# Deep Learning for Recommender Systems: Joint Learning of Similarity and Preference

Master Thesis of

Marcel Kurovski

in Industrial Engineering and Management
at the Department of Economics and Management
Institute for Applied Informatics and Formal Description Methods

| | |
|---|---|
| Reviewer: | Prof. Dr. York Sure-Vetter |
| Advisor: | M.Sc. Steffen Thoma |
| External advisor: | Dr. Florian Wilhelm (inovex GmbH) |

March 27th 2017 – September 27th 2017

Karlsruher Institut für Technologie
Fakultät für Wirtschaftswissenschaften
Kaiserstraße 89
76133 Karlsruhe

für meine Mutter
aus tiefstem Dank

*for my mother
from deepest gratitude*

# Abstract

Recommender Systems are filtering engines that aggregate opinions to support decision making processes with personalized suggestions. They are widely used and influence the daily life of almost everyone in different domains like e-commerce, social media, or entertainment. The efficient generation of relevant recommendations in large-scale systems is a very complex task. In order to provide personalization, engines and algorithms need to capture users' varying tastes and find mostly nonlinear dependencies between them and a multitude of items. Enormous data sparsity and ambitious real-time requirements further complicate this challenge. Meanwhile, the pervasive application of deep learning proves to solve complex tasks like object or speech recognition where traditional machine learning failed or showed mediocre performance. Despite this fact, research on the application of deep learning on recommender systems is still in its infancy.

My master thesis provides an overview of the state of art in deep learning for recommender systems. I illustrate the successful application of different deep neural network types in various domains, point out the challenges and derive trends. Furthermore, I combine latest research to build a recommender system based on deep learning for another domain: the recommendation of vehicles on a large online market. I extend the wide and deep learning approach proposed by Google. Therefore, I augment the optimization criterion with embedding similarity besides preference. I develop strategies for efficient candidate generation applying Locally Optimized Product Quantization on user and item embeddings. The proposed approach outperforms collaborative filtering and hybrid collaborative-content-based filtering in terms of relevance.

# Zusammenfassung

Empfehlungssysteme aggregieren Meinungen, um Entscheidungsprozesse durch personalisierte Empfehlungen zu unterstützen und werden dem Gebiet des Information Filtering zugeordnet. Sie sind weit verbreitet und beeinflussen den Alltag nahezu jedes Menschen bspw. im Online-Handel, bei Social Media oder Unterhaltungsdiensten. In großen Systemen stellt die effiziente Generierung relevanter Empfehlungen eine sehr komplexe Aufgabe dar. Zur Personalisierung müssen die Systeme und Algorithmen zwischen einer Vielzahl von Nutzerinteressen und einem oft ebenso großen Korpus von Items meist nichtlineare Abhängigkeiten finden. Die enorme Spärlichkeit beobachteter Verbindungen zwischen Nutzern und Items sowie Echtzeitanforderungen erschweren diese Aufgabe. Parallel hierzu werden mit der Verbreitung von Deep Learning komplexe Aufgaben wie bspw. Objekt- oder Spracherkennung weitaus besser als durch traditionelles Machine Learning gelöst. Die Erforschung von Deep Learning in Empfehlungssystemen ist hingegen noch wenig fortgeschritten.

In meiner Masterthesis gebe ich einen Überblick über den aktuellen Stand der Technik zu Deep Learning für Empfehlungssysteme. Ich beschreibe hierzu die erfolgreiche Anwendung tiefer neuronaler Netze unterschiedlicher Typen in verschiedenen Domänen und verweise auf Herausforderungen sowie Trends. Außerdem kombiniere ich neueste wissenschaftliche Erkenntnisse, um ein Deep Learning basiertes Empfehlungssystem für einen weiteren Bereich zu entwickeln: die Empfehlung von Fahrzeugen auf einem großen Online-Fahrzeugmarkt. Ich erweitere den Wide and Deep Learning Ansatz von Google. Hierzu ergänze ich das präferenzorientierte Optimierungskriterium um Einbettungsähnlichkeit. Ich entwickle zudem Strategien für effiziente Kandidatengenerierung durch die Anwendung lokal optimierter Produktquantifizierung auf Einbettungen von Nutzern und Items. Der vorgeschlagene Ansatz verbessert die Relevanz von Empfehlungen gegenüber kollaborativem und hybridem kollaborativ-inhaltsbasierten Filtern.

# Acknowledgements

First and foremost, I have to thank the inovex GmbH who gave me the opportunity to work on this exciting topic with practical relevance and scientific rigor. Both were the product of many fruitful discussions with my dedicated supervisor, Dr. Florian Wilhelm, whom I am deeply grateful. I also thank my colleagues Alexandra Wörner, Zoran Cupic and Mara Hermann, who provided interest and insights to stimulate my work. The pleasant working atmosphere, a technology-enthusiastic culture and deep expertise made inovex an excellent place to examine deep learning and its application. Therefore, I also thank Dr. Lars Perchalla who welcomed and supported me in this great community.

I also want to express my gratitude to Prof. Dr. Sure-Vetter and Steffen Thoma from the Institute for Applied Informatics and Formal Description Methods at the Karlsruhe Institute of Technology (KIT) for their kind supervision and collaboration.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| AE | Autoencoder |
| ALS | Alternating Least Squares |
| ANN | Artificial Neural Network |
| AUC | Area Under the ROC curve |
| BPR | Bayesian Personalized Ranking |
| CBF | Content-based Filtering |
| CDL | Collaborative Deep Learning |
| CF | Collaborative Filtering |
| CNN | Convolutional Neural Network |
| CTR | Collaborative Topic Regression |
| DL | Deep Learning |
| DLRS | Deep Learning based Recommender System |
| DNN | Deep Neural Network |
| ELU | Exponential Linear Unit |
| GD | Gradient Descent |
| GRU | Gated Recurrent Unit |
| IDC | International Data Corporation |
| IR | Information Retrieval |
| KD | Knowledge Discovery |
| kNN | k-Nearest Neighbors |
| LOPQ | Local Optimized Product Quantization |
| LSTM | Long Short-Term Memory |
| MAP | Mean Average Precision |
| MAR | Mean Average Recall |
| MBGD | Mini Batch Gradient Descent |
| MF | Matrix Factorization |
| ML | Machine Learning |
| NLP | Natural Language Processing |
| RBM | Restricted Boltzmann Machine |
| ReLU | Rectified Linear Unit |
| RMSE | Root Mean Squared Error |
| RNN | Recurrent Neural Network |
| ROC | Receiver Operator Characteristic |
| RS | Recommender System |
| SDAE | Stacked Denoising Autoencoder |
| SGD | Stochastic Gradient Descent |
| SVD | Singular Value Decomposition |
| WARP | Weighted Approximate-Rank Pairwise |

# 1. Introduction

*"We are leaving the age of information and entering the age of recommendation."*

Chris Anderson (The Long Tail)

## 1.1. Motivation

The age of information is changing humankind. It has brought new ways of communication and information with unprecedented velocity, volume and variety getting rid of local and temporal boundaries. People around the world are connected with each other, information is instantly spread across the globe and the overall knowledge of our species is easily accessible through the world wide web. The internet has become a huge marketplace and provides a myriad of ways to entertain ourselves. However, this progress also brings drawbacks. One of them takes an increasing toll on us. The generated information increases exponentially. A study by the International Data Corporation (IDC) depicts the annual size of the global datasphere as shown in Figure 1.1. The IDC estimates the total size to be approximately 20 Zettabytes[1] and predicts it to become 160 ZB by the end of 2025. [RGR17] Although data does not equal information, it becomes information by semantics. Consequentially, assuming the available information to multiply in the upcoming years seems reasonable. This information appeals our minds.

As a result, we face a fast increasing demand for information processing as the prerequisite to make decisions and finally enjoy the results of our decisions or move forward and evolve. [SVV99] coins this problem as *Information Overload*, a term defined by Milford & Perry in 1977 that "occurs when the amount of input to a system exceeds its processing capacity." Assuming that the capacity of the human information processing system, namely our brain, does not increase proportionally to the growth of available information, it becomes relatively inferior. This leaves us with a fundamental challenge. The neuroscientist Daniel Levitin describes these disadvantageous effects of information and resulting decision overload. As technology and information interfere with some fundamental rules of how our brains works, he subsumes that we loose efficiency and our ability to deeply focus. [Lev15] In 2008 Clay Shirky, who is a professor at New York University and studies the effects of the internet on society, said at the Web 2.0 Expo in New York: "It's not information overload. It's filter failure." [Asa09] His constructive response frames

[1] 1 Zettabyte (ZB) = 1 trillion Gigabyte (GB)

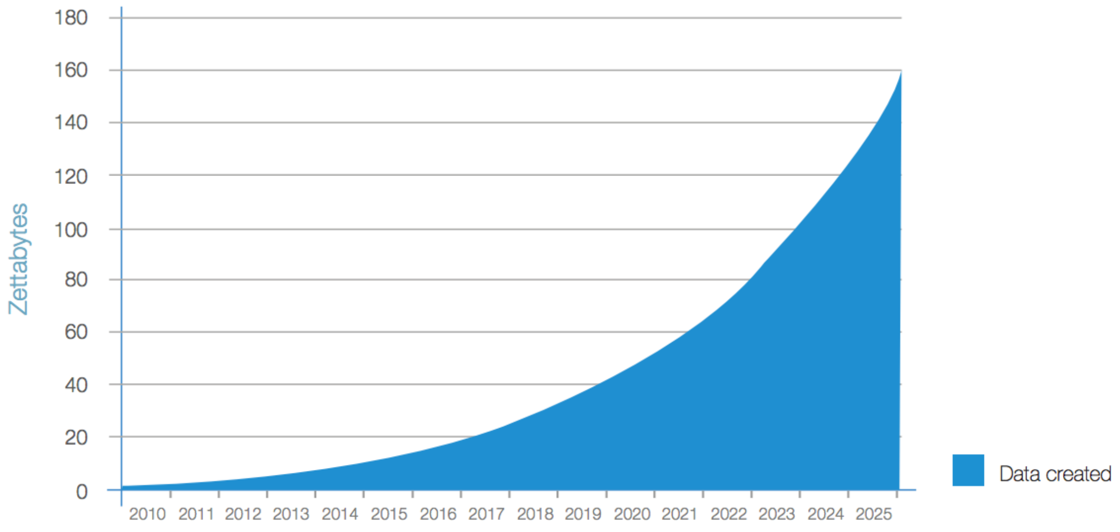

Figure 1.1.: Annual Size of the Global Datasphere [RGR17]

the challenge to address for managing an exploding amount of information: improving information filtering to facilitate decision making and regain our focus. More than ever before we need to filter information that draws on our limited cognitive bandwidth. We need to separate the useful and interesting from the spam and useless that compete for our scarce and valuable attention. Digitization of human life brought advancements, but it also takes its toll. But these advancements also include ways to deal with the toll. They can assist humans in separating the relevant from the irrelevant to safeguard our attention. These systems are called *Recommender Systems*.

We need to profit from increasing information while keeping information processing at a manageable level. This is the only way to save cognitive bandwidth for the decision making part and consecutive merits and joys. Recommender systems can help to address this issue as they filter information to distill them to the relevant. Thereby they facilitate our decision-making process and alleviate temporal and cognitive burden coming with it. As information processing is subjective and involves personal interests, needs and values, recommender systems personalize our consumption of information. They are already widely used in practice and surround almost everyone in daily life. E-commerce, social media, or entertainment are some prominent areas for recommender systems. To personalize content Amazon, Facebook, Spotify, Netflix and many more use recommender systems and machine learning. Thus, Amazon integrated recommendations in nearly every part of the purchasing process. This was perceived as the main driver when they reported a 29% sales increase within a single year in 2012. [Man12] Netflix reported similar success as almost 80% of content consumption stems from recommendations making them an integral part of the whole platform. [McA16] As recommender systems are mainly fueled by machine learning, we need to incorporate another huge development in this field: *Deep Learning*.

Deep learning currently experiences a great hype, but also shows tremendous contributions in different domains. The development of deep learning algorithms was encouraged by their capability to solve complex tasks in artificial intelligence such as object or speech recognition where traditional machine learning algorithms failed or showed mediocre performance. Object recognition surpassed human performance in 2015 regarding the ImageNet dataset. [IS15; HZRS15] With the release of WaveNet, Google made a big step towards generating speech that mimics the natural sound of human voice reducing the gap by 50%. [ODZS16] The advancements show also practical applications, for example in medicine for skin cancer detection with a derma-tologist-level classification of skin cancer with deep neural networks published in the Nature magazine in January 2017. [EKNK17] Humans are beaten in Chess, Go and No-Limit Texas-Hold'em Poker - games that involve a huge complexity of decision-making. These breakthroughs fuel expectations in the application of deep learning for recommender systems.

Deep learning for recommender systems is an emerging but still young research area demonstrating first promising results. The power of deep learning creates new ways to capture non-linearities in user behavior for personalization and to deal with sparse data in large-scale systems. YouTube, Spotify, and Microsoft have just recently reported major improvements by enhancing their recommendation engines with deep learning. Thus, boosting the capacity of recommender systems with deep learning, we could relieve part of the information age's burden by technology that came along with it. Recommendations may guide decision-making towards what suits best and is in line with our needs and interests. Using deep learning to leverage information will address this issue of high relevance and large impact. Therefore, this master thesis seeks to provide an overview of the current state of art as well as to develop and evaluate a recommender system based on deep learning for vehicle recommendations on a large online market.

## 1.2. Research Question and Objective

The thesis provides two research objectives associated with respective questions. First, I will provide a structured overview of current research on deep learning for recommender systems. Second, I will develop and implement a deep learning based recommender system for vehicle recommendations to demonstrate its superiority over conventional recommendation approaches. I will evaluate this superiority terms of recommendation relevance. Turning these objectives into questions yields the following that guide my investigation:

1. What is the current state of art in the application of deep learning for recommender systems?

2. How much does a deep learning based recommender outperform classical information retrieval approaches in terms of recommendation relevance?

## 1.3. Course of Investigation

In order to answer my research questions, I split my course of investigation into six chapters. In **chapter 2** I will provide an overview of the theoretical concepts recommender systems and deep learning are based on. This also includes a brief summary of applied technologies to transfer theory into practice. **Chapter 3** depicts the state of art in deep learning for recommender systems and distinguishes between rather practical and rather theoretical works, surveys and summaries as well as other related research. The first research question will be answered herein. In **chapter 4** I will describe my own end-to-end approach for a vehicle recommendation engine that is powered by deep learning. Consequentially, **chapter 5** illustrates and describes and discusses the results of my approach compared to conventional methods. Thereafter, **chapter 6** serves to conclude on these results leading to position my approach within the existing research. Chapter 4 to 6 serve to answer my second research question. Finally, **chapter 7** shows an outlook on the further path of research and application in the field of deep learning based recommender systems.

# 2. Theoretical Foundations

*"He who loves practice without theory is like the sailor who boards ship without a rudder and compass and never knows where he may cast."*

Leonardo da Vinci

This chapter introduces the theoretical foundations of recommender systems as well as deep learning and presents technologies critical for my implementation. Due to the inherent complexity and large theoretical corpus, I will limit this chapter to the most fundamental and directly relevant topics. For deeper discourse, I recommend consulting *Deep Learning by Ian Goodfellow, Yoshua Bengio and Aaron Courville* [GBC16] and *The Elements of Statistical Learning by Trevor Hastie, Robert Tibshirani and Jerome Friedman* [HTF09] for deep learning as well as *Recommender Systems edited by Charu Aggarwal* [Agg16] and *Recommender Systems Handbook edited by Francesco Ricci, Lior Rokach and Bracha Shapira* [RRS15] for recommender systems. Regarding technologies used herein I advise refering to respective GitHub documentations mentioned in footnotes or the publications associated with each technology, which are referenced and found in the bibliography. Nevertheless, *Hands-On Machine Learning with Scikit-Learn & TensorFlow by Aurélien Gerón* [Gér17] is in general highly recommendable from the implementation point of view.

## 2.1. Recommender Systems

The following part starts with a formal definition of recommender systems (RSs) and differentiates approaches and perspectives. It presents the two fundamental approaches to RSs: collaborative and content-based filtering. Within this part, I will also outline the fundamental notations together with a mathematical formalization. Afterwards, I will elaborate on the metrics to evaluate the performance of RSs and conclude with a section on common challenges.

### 2.1.1. Definition and Distinction

RSs or recommendation engines are software tools and techniques that assist and augment the decision making process by aggregating people's opinions and directing them to appropriate recipients. [RV97] [RRS15, p.1] They can also be interpreted as "search ranking systems". [CKHS16]

We distinguish between six different classes of RSs:

- **Collaborative**
- **Content-based**
- Demographic
- Knowledge-based
- Constraint-based
- Community-based

It is possible to blend different classes with each other. The resulting recommender is referred to as a **hybrid RS**. Furthermore, context can be used to augment recommendations. Thus, using additional temporal, local, social or other information accompanying the user-item interaction, we can filter recommendations or integrate the context into the model itself. The thesis will cover collaborative, content-based and hybrid approaches as they are the most common types and for reasons of brevity. For more information on other classes, refer to [RRS15]. We require a few basic terms for a formal explanation of RS classes which are shortly described in the following.

RSs are used in different domains and have the general objective to present the items to a user that have the highest probability of catching his or her interest. This probability can be reformulated as user utility which recommendation's task is to maximize. As we direct user attention towards a list of few items, we increase the likelihood for content consumption, e.g. buying a suggested product, watching a recommended movie, or socializing with proposed people. This raises a need for utility quantification resembling a personalized score $\tilde{x}_{ui}$ for an item given a specific user. It also raises the need to distinguish between *rating* and *ranking* into three categories as proposed by [WHLM16]:

- Pointwise Rating (Prediction)
- Pairwise Ranking
- Listwise Ranking

*Pointwise ratings* only consider single user-item combinations $(u, i)$ and predict the user's rating for the item. Personalized ratings impose item rankings from which we can generate recommendations. [RFGS09] *Pairwise Ranking* looks at item pairs $(u, i_a, i_b)$ to estimate which item is preferred to another and tries to minimize the number of inversions in pairwise rankings. This is even more natural as we are interested in an ordered list of items and far less in explicit scores. *Listwise Ranking* is more eager as it targets to directly generate an optimal list of items without additional pairwise comparisons. By ranking items we recommend the $top - k$ most relevant items for a given user which is more common and natural as rating prediction. Additionally, we must distinguish between prediction and recommendation. Predictions are quantifiable estimates of how much a user likes an item. Recommendations are a set of suggestions a user might like. Thus, a list of $top - k$ recommendations can display predictions, but still is a display of recommendations. [KE17] Nevertheless, all techniques require a certain extent of prior information to learn parameters for a rating or ranking model. This prior information relates to feedback which is covered in the next paragraph.

Feedback is abstract and object to ambiguous interpretation. In RSs we separate into two types of observed user feedback: preference or rejection of items. Preference can be revealed either explicitly or implicitly. In both cases ratings present the quantification of liking or dislike. Explicit feedback ratings indicate *preference*, whereas implicit feedback corresponds to *confidence* that can be more ambiguous than preference. [RRS15, p. 9 sq.] distinguishes into four types of ratings: numeric, ordinal, binary, and unary. Implicit feedback matrices are often unary. This means that we only perceive a positive signal (click, purchase, etc.), but obtain no information for absent ratings which either mean disregard or more often unawareness. The feedback is called implicit as the user's primary action intent is not to signal preference. In contrast, when users take action for the purpose of rating, we call that explicit feedback. Votes or reviews present examples for this clear preference sign. The rating scales for numeric, ordinal, and binary feedback impose an order on like and dislike. Numeric rating scales are often discrete, e.g. 5-star-rating scales, but may also be continuous. Ordinal ratings use categorical values with an underlying order. Binary ratings only distinguish between like and dislike, but explicitly contain dislike signals. These signals are object of ambiguous interpretation in unary rating contexts vanishing within unaware items. In addition, temporal dynamics can also play a significant role as user preferences might degrade over time. [Agg16, p. 10]

Recommendations are seen to be relevant when the user consumes them. We can evaluate this by observing the click-through-ratio (CTR) which is shown in section 2.1.4. Besides this objective there are more operational and technical goals of RSs outlined in 2.1.4.

### 2.1.2. Collaborative Filtering

Collaborative Filtering (CF) is the most common approach for recommendation engines. In order to identify new user-item associations, CF analyzes relationships between users and interdependencies among products, resp. items. [HKV08] Given sets of users $u \in U$ and items $v \in V$ with $m = |U|$ and $n = |V|$, we require a history of interactions that resemble user ratings on specific items $r_{i,j}$. These ratings can be illustrated in an incomplete matrix $R = \{r_{i,j}\}$. Missing entries in $R$ relate to ratings that have not been observed yet, but which might be observed in the future. Since both, $m$ and $n$ are typically large in recommendation contexts and users normally interact with just a relatively small portion of all items, $R$ becomes very sparse. CF turns the recommendation problem into a matrix completion problem. This requires to predict the missing values in $R$ based on known ratings and presenting the $top - k$ entries to the respective user. We distinguish between two techniques to solve the matrix completion task: *memory-based* and *model-based* techniques. They both exploit item and/or user similarity. We conduct user-user CF when using user similarities, and item-item CF vice versa. Both produce similar results, but may differ significantly in terms of efficiency since $m$ is rarely as high as $n$. [Abe16; ERK10; KAPM08]

**Memory-based**, also denoted as **neighborhood-based**, techniques infer ratings based on item or user similarities. Therefore, they perform a weighted average over the user, resp. item, ratings of the $k$ most similar entities. We can use different similarity metrics for this task, among which cosine similarity, or Pearson correlation are the most popular. [HKBR99] This approach is called **k-Nearest Neighbors** (kNN). [HTF09, p.14] In order to generate recommendations for a specific user $x \in U$, we apply a three-step process: first, we calculate the cosine similarity between $x$ and all other users $y \in \{U \setminus x\}$ which is defined as

$$sim(x,y) = cos(x,y) = \frac{\sum\limits_{i \in V_{xy}} r_{x,i} \cdot r_{y,i}}{\sqrt{\sum\limits_{i \in V_{xy}} r_{x,i}^2} \cdot \sqrt{\sum\limits_{i \in V_{xy}} r_{y,i}^2}} \tag{2.1}$$

where $V_{xy}$ denotes the set of items rated by both users x and y. Given those similarities we secondly select a $k \in \mathbb{N}$ and introduce the subset of $k$ users that are most similar to $x$, i.e. that have the $k$ highest $sim(x,y)$, as $N_k(x)$. To estimate the rating $\tilde{r}_{x,j}$ with $j \notin V_x$ we take the similarity-weighted average of users in $N_k(x)$ which rated item $j$ defined as:

$$\tilde{r}_{x,j} = \frac{\sum\limits_{y \in N_k(x) \cap U_j} sim(x,y) \cdot r_{y,j}}{\sum\limits_{y \in N_k(x) \cap U_j} sim(x,y)} \tag{2.2}$$

Finally, we can rank items $j \notin V_x$ given their estimated ratings $\tilde{r}_{x,j}$ and recommend them to user $x$ starting with the highest rated items.

**Model-based** techniques include Naïve Bayes, Clustering, or (probabilistic) Matrix Factorization (MF). MF-based models are also denoted as latent factor models and present the most popular model-based technique. They are based on machine learning algorithms to build models that find patterns in the training data, i.e. past ratings. Thus, they are able to generalize to new data, i.e. they estimate unknown ratings to create recommendations.

Again, we start with a set of users $U$, a set of items $V$ and a sparse rating matrix $R$. We need to estimate missing values in $R$ and by doing so complete the matrix. MF decomposes $R$ into two lower-dimensional matrices $P$ and $Q$ that should resemble $R$ as good as possible. $P$ and $Q$ both map users, respectively items, into a $d$-dimensional embedding space with $d \ll min(m,n)$. Thus, instead of using ratings to represent users and items as in memory-based techniques, we use latent factors that are much fewer and therefore more dense compared to the sparse vectors within $R$. Finally, we use these dense representations - fitted to resemble observed ratings $r_{i,j}$ - to estimate unobserved ratings $\tilde{r}_{i,j}$.

Thus, the target of MF is to *reconstruct* $R$ using the matrix multiply of dense user and item representations:

$$R_{m \times n} \approx \tilde{R} = P_{m \times d} \times Q_{n \times d}^T \tag{2.3}$$

To quantify the *reconstruction error* we use the root mean squared error (RMSE) between original and reconstructed ratings:

$$\begin{aligned} RMSE &= \sqrt{\frac{1}{|S|} \sum_{(i,j) \in S} (r_{ij} - \tilde{r}_{ij})^2} \\ &= \sqrt{\frac{1}{|S|} \sum_{(i,j) \in S} (r_{ij} - p_i q_j^T)^2} \end{aligned} \tag{2.4}$$

$$\begin{bmatrix} r_{1,1} & r_{1,2} & \dots & r_{1,n} \\ r_{2,1} & \ddots & \ddots & r_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ r_{m,1} & \dots & \dots & r_{m,n} \end{bmatrix} \approx \begin{bmatrix} p_{1,1} & \dots & p_{1,d} \\ \vdots & \ddots & \vdots \\ \vdots & \ddots & \vdots \\ p_{m,1} & \dots & p_{m,d} \end{bmatrix} \times \begin{bmatrix} q_{1,1} & \dots & \dots & q_{n,1} \\ \vdots & \ddots & \ddots & \vdots \\ q_{1,d} & \dots & \dots & q_{n,d} \end{bmatrix}$$

Figure 2.1.: Factorization of Ratings' Matrix R by P and Q

where $S$ denotes the set of user-item tuples for all ratings we observed. We now have to initialize and adapt the respective latent factor vectors $p_i, q_j \in \mathbb{R}^d$ minimizing the RMSE. This resembles an optimization problem which we can address by different algorithms: Stochastic Gradient Descent (SGD), Bias SGD, Alternating Least Squares (ALS), or weighted ALS. [Abe16]

SGD is a standard algorithm used for these kinds of problems and works by gradually adapting the latent factor variables in order to minimize RMSE, which is a differentiable function. This allows us to calculate the partial derivatives for either the user or item embedding variables. The resulting gradient points into the direction of a local or global minimum. We guide the loss function towards its minimum by multiplying the gradient with a predetermined *learning rate* $\alpha$ for the latent variable update. The convergence is critically dependent on an appropriate choice for $\alpha$:

$$\begin{aligned} p_i &:= p_i - \alpha \cdot \frac{\partial RMSE}{\partial p_i} \\ q_j &:= q_j - \alpha \cdot \frac{\partial RMSE}{\partial q_j} \end{aligned} \tag{2.5}$$

The formulas above relate to batch gradient descent (BGD) where we first calculate the training error on all training examples (the batch) before updating the variables. SGD as opposed to BGD updates variables after every single example. The error is just given by the squared error between true and estimated rating. Thus, transferring formula 2.5 to SGD and applying the chain rule leads to the following update rule for SGD:

$$\begin{aligned} p_i &:= p_i - \alpha \cdot \frac{\partial SE}{\partial p_i} = p_i + 2 \cdot \alpha \cdot E \cdot q_j \\ q_j &:= p_j - \alpha \cdot \frac{\partial SE}{\partial q_j} = q_j + 2 \cdot \alpha \cdot E \cdot p_i \end{aligned} \tag{2.6}$$

*SE* describes the squared error and *E* the sole error between true value and prediction as part of the RMSE formula.

We can extend the model-based MF approach with additional information, e.g. temporal effects, confidence levels, or regularization. Adding a regularization term can avoid overfitting the training data and provide better generalizations, i.e. rating estimations. The following equation shows the RMSE augmented by regularization which is controlled using a constant $\lambda \in \mathbb{R}^+$:

$$RMSE = \sqrt{\frac{1}{|S|} \sum_{(i,j)\in S} (r_{ij} - p_i q_j^T)^2 + \lambda(\|p_i\|^2 + \|q_j\|^2)} \tag{2.7}$$

Since users and items can have significant biases, e.g. very critical users with a comparatively low rating average or hyped items with a comparatively high rating average, distortions can arise and negatively affect the model's predictive capacity. This behavior is anticipated by adding user- and item-specific bias terms $b_i, b_j \in \mathbb{R}$ to the estimated rating which then turns into the following:

$$\tilde{r_{ij}} = p_i q_j^T + b_i + b_j \tag{2.8}$$

Another option is to make latent factors and biases time-dependent and thus to account for potential temporal distortions, e.g. item popularity:

$$\tilde{r_{ij}} = p_i q_j^T(t) + b_i(t) + b_j(t) \tag{2.9}$$

The appendix includes a simple example for MF to illustrate the mechanisms. It can be found in A.1.

### 2.1.3. Content-based Filtering

Content-based Filtering (CBF) presents another well-established approach to recommendations where the system learns to recommend items based on the similarity of attributes. These attributes are part of user profiles and item representations and can be simple keywords or concept-based representations of items and user profiles. They actually present content which is mapped against each other as the basis for recommendations. User content can relate to preferences or interests, whereas item content can be textual or consist of item-associated metadata. As a result of matching item attributes with attributes of target user profiles, we get relevance scores considered as the preference level a user has for a specific item. [RRS15, p. 11 sq., p. 119 sq.]

[RRS15] present a 3-step recommendation process for CBF which is illustrated in Figure 2.2. Content analyzer, profile learner, and filtering component are the main parts of the system. The content analyzer provides structured content for the following steps. The profile learner constructs a model to generate user profiles based on past interactions. It leverages probabilistic models, relevance feedback from users, or kNN doing so. Finally, the filtering component matches user profiles against item representations to produce a binary or continuous relative judgment. These item judgments are ordered in a ranked list of items that are likely to be interesting for a specific user.

User profiles combine the rating behavior of users with the content of rated items being agnostic to the rating behavior of other users. Item descriptions that are labeled with ratings (either implicit or explicit) are used as training data to obtain a user representation. This introduces the capability of recommending new items, however it does not provide generalization to new users. [Agg16, p. 14 sq.]

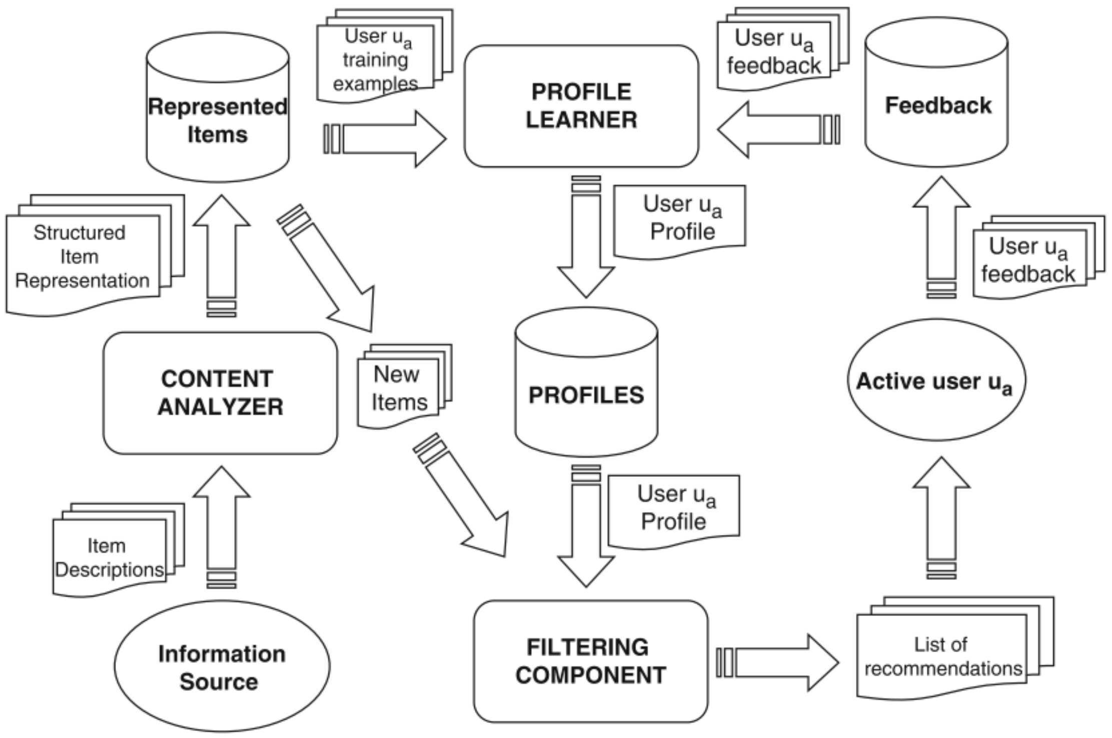

Figure 2.2.: High-level Architecture of a Content-based Recommender [RRS15, p. 121]

## 2.1.4. Evaluation

This section presents the reasoning, objectives and metrics for evaluating the performance of RSs as well as the design of such an evaluation.

### 2.1.4.1. Evaluation Reasoning

We need to evaluate RS to get an objective assessment of their capability to assist and augment the decision making process which is the task they are designed for. The definition also mentions *appropriateness* of targeted users. Thus, we need to achieve transparency on how well RSs assist and augment decision making as well as how *appropriate* recommendations are. Obtaining quantifiable results on the performance of RSs allows us to track the impact of model adaptions towards improving a model. It also enables us to benchmark RSs with each other. Therefore, evaluation presents a quintessential part when designing, implementing and improving RSs.

Online and offline evaluation as well as user studies are the three primary types of evaluation of RSs. **Online evaluation** measures user reactions to displayed recommendations. The participants are typically in a fully deployed commercial system and behave in a natural way as they interact with the live system which results in a lower evaluation bias. The typical online metric is the conversion rate or CTR which measures the proportion of recommendations that are selected by the user. It is possible to extend the view on the conversion rate by adding monetary aspects. Thus, one might take into account the costs of recommendations and the expected profit of a successful recommendation to anticipate a varying relative item importance. A/B-Testing is a typical instrument to conduct online evaluation. The users are separated into two groups A and B of which one serves as baseline and the other is confronted with the new system to evaluate changes in behavior, e.g. explicated by differing conversion rates on statistically significant user groups. **Offline evaluation** is the most common and practical type in which historical data are leveraged. One might also use externally available datasets from various domains, e.g. Netflix Price, MovieLens, or XING RecSys 2017 data, in order to assess RSs. Offline typically comes before online evaluation to pre-test algorithms prior to user testing as it does not affect a live system and takes less time. In online evaluation we can directly measure how users respond, but bear the risk that tests are inferior to the standard system and thus may repel users. However, since online and offline evaluation do not necessarily correlate, a thorough evaluation must contain both for the sake of reliable evaluation results. Finally, there are **user studies** that can be conducted in an online or offline lab where predetermined users are confronted with a system and requested to deliver detailed feedback. Due to a higher evaluation bias in this artificial context and the high costs this type is less relevant. [ERK10, p. 114 sqq.] [Agg16, p. 225 sqq.]

#### 2.1.4.2. Evaluation Objectives

Evaluation always depends on an objective against which to evaluate a system. The overall objective of a RS is to increase conversion rates. There is a multitude of objectives for RSs which have a long- or short-term effect on this target: accuracy, coverage, confidence together with trust, novelty, serendipity, diversity, robustness as well as stability, and scalability. These goals partially require trade-offs and differ in the degree up to which they can be measured objectively or refer to a rather subjective user experience. Many RSs depend on rankings of a list of $top - k$ items rather than ratings which is particularly the case for implicit feedback environments. This also influences the considerations on the most important goal of recommender systems that is **accuracy**, also denoted as **relevance**. Below, I will elaborate briefly on most of these goals and cover accuracy and scalability in more detail as they are essential for answering the research question.

**Accuracy or relevance** can further be divided into two subdomains that are accuracy of estimated ratings and accuracy of estimated rankings. The metrics used for quantifying those domains are different. To quantify the accuracy of ratings as numeric quantities, we normally calculate the absolute or squared difference between true and predicted rating using mean absolute error (MAE), mean squared error (MSE), or the RMSE (see specification in chapter 2.1.2). The subdomain of ranking accuracy can be further split into three areas: rank-correlation metrics, utility-based measures, and the receiver operator characteristic (ROC). Both latter are particularly relevant for implicit feedback datasets and associated with respective metrics in a later section.

**User-space or item-space coverage** considers the proportion of items and users for which we can actually generate recommendations. In the context of CBF we can interpret user-space as the amount of user profiles that can be built upon existing interaction history.

**Confidence and trust** describe the degree of certainty the system has towards its recommendations and how likely users are to believe these recommendations. "While confidence measures the system's faith in the recommendation, trust measures the user's faith in the evaluation." [Agg16, p. 232]

**Novelty, serendipity and diversity** are likewise but pose different notions. Novelty describes the degree up to which users receive previously unknown items within their taste profile, whereas serendipity is stronger in the sense of delivering rather unexpected recommendations which may augment or change a user's taste profile and help to discover. Diversity relates to the list of $k$ user-specific recommendations itself and demands these items to be dissimilar such that the user does not perceive them as same items.

**Robustness and stability** consider the resilience of recommendations against fake ratings or negative influences from implicit feedback.

**Scalability** becomes increasingly important in the presence of Big Data (see chapter 2.2.1). As the amount of collected implicit and explicit feedback rises exponentially it becomes more crucial to consider efficiency aspects when turning these data into recommendations. Scalability can be divided into three areas: training time, prediction time, and memory requirements. Training models and choosing the right one as well as retraining existing models is normally very time-consuming, e.g. MF learning latent factors for users and items, or deep learning adapting weights during training. Compared to prediction time training is less critical since its mostly done offline and decoupled from direct user requests. Prediction time regards the time it takes to infer recommendations when users demand them. Since latency is critical to user experience, we tackle low inference times. Memory deals with the allocation of data and is restricted by disk, main or processor memory. We often face a trade-off between memory and processing times to achieve a satisfying level of inference times. [Agg16, p. 229 sqq.]

RS can target additional goals like customer satisfaction, recommendation explanation, loyalty, etc. which are not covered here. [Agg16, p. 3 sqq.]

#### 2.1.4.3. Evaluation Design

The design of evaluation focuses on offline accuracy based on historical datasets. It is similar to the frameworks used in classification and regression tasks. Overall, we need to separate the set of observed ratings $S$ into training and evaluation datasets. Evaluation data can be further divided into validation and testing data. First, we use the training data in order to train one or several models. Using validation data enables us to select the best performing model or tune the model parameters. Finally, testing data is used to give us an estimate of the generalizability of the model, i.e. how well it performs on new, previously unseen data. For example, the well-known MovieLens 100K Dataset [HK15] comprises ratings of $m = 943$ users on $n = 1{,}682$ items. It is split into a training dataset containing 19,048 ratings and a test datasets with 2,153 rating which roughly relates to a 90/10-split.

#### 2.1.4.4. Evaluation Metrics

Finally, we need metrics to quantify the degree up to which we achieve certain system goals. The following section introduces into the metrics used to measure accuracy. We start with the confusion matrix to develop these metrics from:

The **confusion matrix** as shown in Table 2.1 is a fundamental concept in Information Retrieval (IR) and separates into retrieved and relevant information. In RSs *retrieved* information relates to the list of $k$ recommendations that are generated for user display. *Relevant* information regards those items that are actually preferred by a user. $TP$ stands for *true positives*, $FP$ for *false positives*, $FN$ for *false negatives* and $TN$ means *true negatives*. [Faw06]

Table 2.1.: Confusion Matrix

| | relevant | ¬ relevant |
|---|---|---|
| retrieved | TP | FP |
| ¬ retrieved | FN | TN |

**Precision** and **Recall** are two metrics that derive from the confusion matrix. Precision is the percentage of retrieved items that are relevant, whereas recall is the percentage of relevant items that are retrieved. [KE17] Since we deal with lists of $k$ items, retrieval is limited to $k$ ($k = |retrieved_k|$). Thus, we usually deal with Precision@k and Recall@k.

$$Precision@k = \frac{retrieved_k \cap relevant}{k} \tag{2.10}$$

$$Recall@k = \frac{retrieved_k \cap relevant}{|relevant|} \tag{2.11}$$

We can determine these metrics for each user where the average over all users yields the mean average precision @ $k$($MAP@k$), or mean average recall @ $k$ ($MAR@k$).

The $F_1$**-Score** combines both metrics due to their inherent trade-offs (refer to [Faw06]) and resembles a weighted average of precision and recall:

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{2.12}$$

The **reciprocal rank** defines the inverse of the rank of the first relevant retrieval in the list of $k$ items. If there is no relevant item in the list, the reciprocal rank is undefined, but normally set to zero in that case. Averaging over all users, respectively queries $Q$, leads to the **mean reciprocal rank (MRR)** [Agg16, p. 240 sqq.]:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \tag{2.13}$$

The **Receiver Operator Characteristics (ROC)** is a graph to visualize and compare classifiers in terms of their classification quality. We need to distinguish between discrete and probabilistic classifiers. For example, in the case of binary classification a discrete classifier just outputs one of the target labels, whereas for a probabilistic classifier there is a probability of belonging to a certain class. In the latter case a predetermined threshold guides this probability to the respective class. ROC compares **true positive rate (TPR)** on the y-axis with **false positive rate (FPR)** on the x-axis. TPR is equivalent to precision, whereas FPR is defined as follows:

$$FPR = \frac{FP}{FP + TN} \tag{2.14}$$

TPR and FPR are both defined for the interval $[0, 1]$ and span the ROC space in which a discrete classifier resembles a single point and where a probabilistic classifier resembles a collection of points as the threshold between the two classes is varied. These multiple points for the same but differently discriminating classifier constitute a curve which is called ROC curve. Please refer to Figure 2.3 for an illustration.

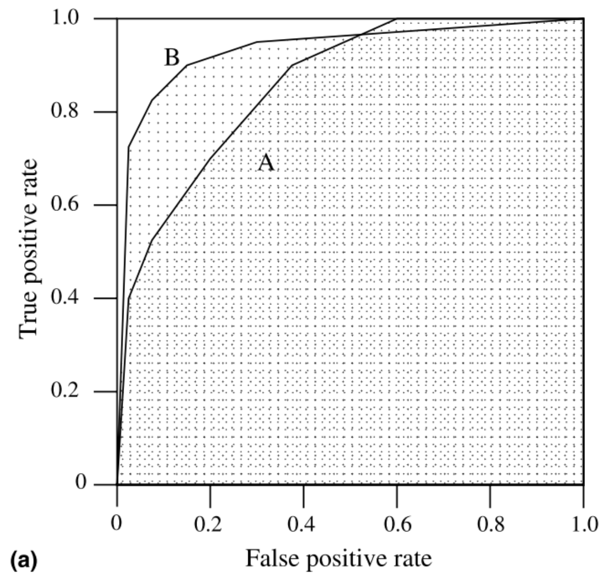

Figure 2.3.: ROC Curves and AUC for two Probabilistic Classifiers [Faw06]

The **Area Under the ROC Curve (AUC)** is the space between ROC curve and the diagram thresholds and is also defined for $[0, 1]$. The perfect score is 1 and a random classifier yields a score of 0.5. Thus, we can evaluate the performance of a classifier by considering its AUC score which also enables us to compare several classifier with each other, e.g. classifier B is better compared to A as illustrated in Figure 2.3.

Finally, we also need to define **classification accuracy** which is the share of all training examples that were classified correctly, i.e. either positively or negatively:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{2.15}$$

These are the most relevant metrics to evaluate accuracy/relevance of recommendations. They are used across different papers relevant for this thesis: [CAS16; Kul15; ESH15; HKBT15; QKHC17] report on MRR, MAP/P@k, and MAR/R@k, [ODS13] evaluate with MAP, and AUC, [CKHS16] use the AUC.

### 2.1.5. Challenges

Many difficulties arise when developing and operating RSs. Therefore, it is inevitable to remember them and design systems in which they become mitigated or at least tolerable. I will not present an exhaustive list, but elaborate on three important ones: Sparsity, Cold-Start, and Scalability

**Sparsity** describes the issue that a large part of potential user-item interaction resembled by the interaction matrix $R$ is unknown. Its counterpart is density which is defined as the ratio between known entries and the size of the user-item interaction matrix. For example, regarding the MovieLens 100k dataset [HK15] we have $m = 943$ users and $n = 1{,}682$ items, which are linked by 100,000 unique ratings on a scale from 1 to 5. Thus, the rating matrix has a size of $m \cdot n = 943 \cdot 1{,}682 = 1{,}586{,}126$. Since 100,000 ratings are provided, the density is $\frac{100{,}000}{m \cdot n} = 0.0630 = 6.3\%$. This leads to a $sparsity = 1 - density = 1 - 0.0630 = 0.9370 = 93.7\%$. Having no information on most of all potential interactions is a common problem in RSs. It is the very starting point of why we need to make predictions on ratings and build our rankings from them. It does not only pose uncertainty, but also computational complexity. We partially alleviate the sparsity problem by matrix factorization methods that convert sparse into dense representations of users and items.

**Cold-Start** can be divided into user or item cold-start and relates to users, respectively items, with no interactions. [WHCZ17] introduce *incomplete cold start (ICS)* to denote entities with too few instead of none interactions to be significant yet. In CF cold-start presents a crucial problem since interactions are the only source of information to infer preferences upon. Therefore, in CF new users or new items can not be recommended or receive personalized recommendations unless the system experiences interactions from them. Only then, there is a chance to generate recommendations. In CBF cold-start is less a problem in terms of items as they enter the system with their features set. Regarding users, the problem is still difficult as the user profile learner has no data, i.e. user-item interactions, on which it can infer user profile features on. This CBF user cold-start situation is often mitigated through the existence of a default profile that contains popular features, however it is a mediocre approach due to the lack of personalization.

**Scalability** is a concern inherently caused by sparsity. Real-world RSs often deal with millions of items and more than a billion of users [CAS16; ESH15; CKHS16]. The classical IR dichotomy is a attempt to fulfill low inference latencies, e.g. 10 milliseconds. [CAS16]

Nevertheless, candidate generation needs to balance between efficiency and quality. This demands for efficient algorithms and parallel structures for RSs to scale well and achieve high quality in limited time.

## 2.2. Deep Learning

The following section introduces Deep Learning (DL) starting with definitions and distinctions in the superordinate field of Artificial Intelligence (AI). I will continue with the transition from artificial neural networks (ANNs) to deep neural networks (DNNs) as the building blocks of DL. Afterwards, I will provide a formal mathematical introduction of what brings DNNs to life, i.e. to solve complex problems by learning patterns in data. Finally, I will briefly outline some challenges DL scientists and practitioners face.

### 2.2.1. Definition and Distinction

The field of AI involves many different topics like Big Data, Data Science, Knowledge Discovery, Data Mining, machine learning and of course DL itself. The following section puts them into a coherent context and provides clear definitions.

**Artificial Intelligence (AI)** is concerned with machines that are thinking and acting both rationally and humanly. A machine needs to perform natural language processing (NLP), knowledge representation, automated reasoning, machine learning, computer vision, and robotics to achieve most of what is connected to AI. Nevertheless, the term is too broadly used to give a single, exhaustive answer to its definition. [RN09, p. 1 sqq.] Modern AI is mostly concerned with the question whether and how machines can exhibit intelligence and outperform human beings. The concept that they already do this for specific tasks and thus, within a narrow scope, is framed with the term **narrow AI**. However, the capability of performing general tasks and quickly adapting to new problems involves multitask learning and carries the notion of **general AI**. Latter is not present yet, but expected to evolve.[Pet17] [GBC16, p. 2] describe the transition from informal (human centered) knowledge into the computer as one of the key challenges in AI. This also requires a formal definition of the multitude of general tasks humans perform intuitively on a daily basis. Although, solving formalized problems is easy for machines, formalizing problems we as humans intuitively solve is among the most difficult tasks for us.

#### Machine Learning (ML)

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." [Mit97, p. 2]

ML comprises three core concepts: tasks, performance, and experience. We can understand their meanings and interconnections with the help of an example from fraud detection. The task T can be formulated as a binary classification problem in which we need to distinguish transactions into fraudulent and non-fraudulent. The performance P of solving this task is measured by accuracy, i.e. the percentage of transactions that were correctly classified as fraudulent or non-fraudulent. As the fraud detection system works, it collects data on transactions together with features like involved entities, amounts, etc. which are associated with the class label and augmented by tracking the outcome. These data resemble experience E which helps a machine to learn which classifications were correct and which classifications, especially which features, should be reconsidered in order to improve classification accuracy. Thus, the machine learns from experience to solve its task better which is quantified by the performance measure. Doing so it shows the capability to acquire useful knowledge by extracting patterns from data. [GBC16, p. 2 sq.]

**Representation Learning** addresses the shortcoming of classical ML requiring features that are handcrafted by humans. The task to disentangle the relevant factors for output variation from the irrelevant imposes a performance constraint as the machine can just become as good as human feature engineering allows it to become. Representation Learning goes beyond this and creates features from the raw representations of data that are shown to the algorithm. This resolves the dependence on pre-formulated representations and allows for better performance. Adding additional layers of more abstract features then presents the shift to DL.

**Deep Learning (DL)** is "a scalable way to train nonlinear models on large datasets" (learn complicated functions in high-dimensional space) and was encouraged by the failure of traditional ML algorithms to solve complex AI tasks such as speech or object recognition. [GBC16, p. 153] It not only learns representations from unstructured information, but also it constructs a hierarchy of concepts, where the term concept is used synonymously to representation. This hierarchy starts with simple information that are mapped to simple features as the building blocks for rather complicated concepts. Thus, the constraint of predefined representations is removed which leads to a machine that learns simple and complex representations by itself. The resulting hierarchy of concepts, be it the concepts or the computation steps, coined the notion of depth in learning. As a result, DL replaces the manual feature engineering by humans as in traditional ML. It can be regarded as the "study of models that either involve a greater amount of composition of learned functions or learned concepts than traditional ML does." [GBC16, p. 8] But there is still no consensus on what qualifies DL to be actually 'deep' as the following quote shows:

> Because it is not always clear which of these two views—the depth of the computational graph, or the depth of the probabilistic modeling graph—is most relevant, and because different people choose different sets of smallest elements from which to construct their graphs, there is no single correct value for the depth of an architecture, just as there is no single correct value for the length of a computer program. Nor is there a consensus about how much depth a model requires to qualify as "deep". [GBC16, p. 8]

Figure 2.4 shows how DL is incorporated by ML that goes beyond its capabilities. It has already proven its usefulness in various domains ranging from computer vision to search engines and online advertising. Especially in recent years, DL experienced a tremendous leap forward that is mainly fueled by three developments [GBC16, p. 11]:

- Increasing availability of data in terms of volume, velocity, and variety (Big Data)
- Improving software and hardware infrastructure through software libraries, the application of GPUs and the current advent of Tensor Processing Units (TPUs)
- Spreading application domains with DL outperforming existing solutions

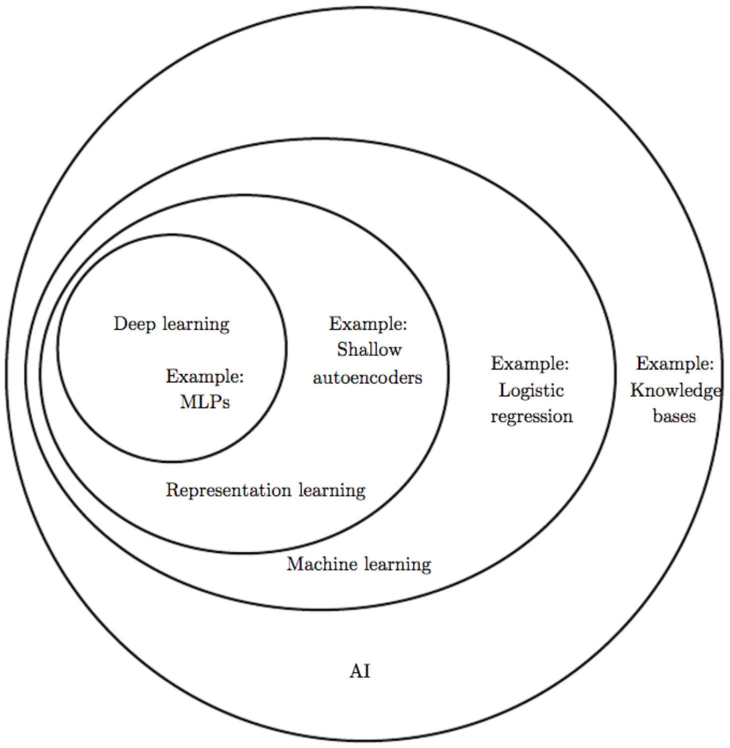

Figure 2.4.: Hierarchy of AI, ML and DL [GBC16, p. 9]

**Knowledge Discovery (KD)** in Databases defines a process to obtain knowledge from data involving five successive steps also shown in Figure 2.5: Selection, Preprocessing, Transformation, Data Mining, and Interpretation/Evaluation. Similar to DL this does not start with features, but with low-level data, where DL partially selects, preprocesses and transforms data in an automated manner. **Data Mining** is considered as just a part of it and refers to the "application of specific algorithms for extracting patterns" from transformed data. [FPS96]

**Data Science** can be described as the "study of generalizable extraction of knowledge from data" involving disciplines such as mathematics, ML, AI, statistics, databases, and optimization with the task to formulate problems and effectively solve them. [Dha13] The term goes beyond KD as the traditional database models change a lot by the massive increase of heterogeneous and unstructured data which requires tools from further domains to integrate and interpret them.

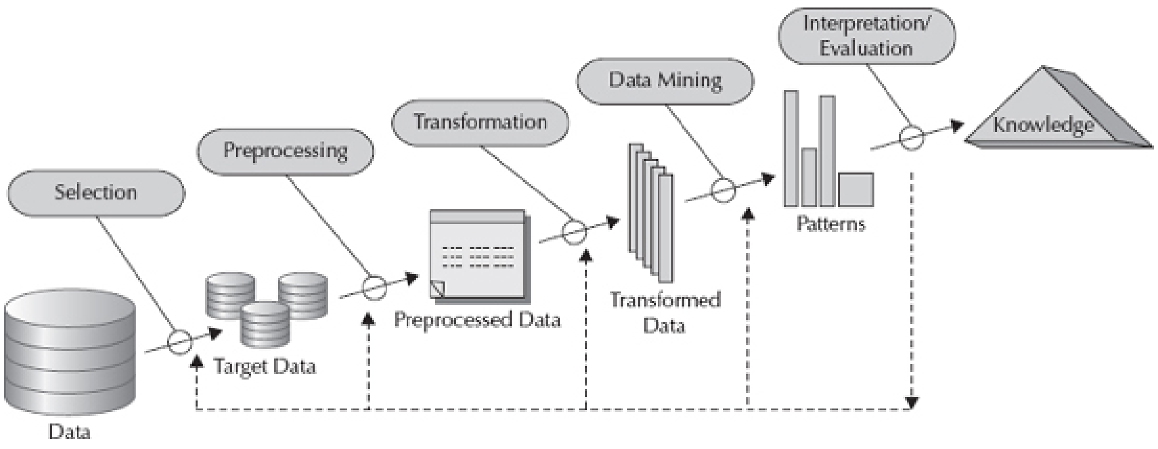

Figure 2.5.: Process of Knowledge Discovery in Databases (KDD) [FPS96]

**Big Data** is "the storage and analysis of large and/or complex datasets using a series of techniques including, but not limited to: NoSQL, MapReduce and machine learning." [WB13] This definition was an attempt to combine different definitions of Big Data among which Gartner provided the most prominent one as a combination of four V's: volume, velocity, variety, and veracity. Volume relates to the increasing size, velocity to the fast real-time data exchange rates, variety to the wide range of formats and representations of data and veracity to trust and uncertainty associated with data.

In summary, we consider DL as a specific kind of ML and thus part of AI. DL has an overlap with the KD process and thus turns data, almost Big Data, into knowledge or actions. Data Science describes the overall domain in which this process happens.

### 2.2.2. Machine Learning

The ML application domains can be broadly separated into two categories: supervised and unsupervised problems. Supervised problems are characterized by providing dependent and independent variables. The task is to learn a function about the mapping from independent (input) to dependent (output) variables. The data already contains supervised learning signals, i.e. output values, and we need to train a model that is capable of estimating an output based on provided input. Regression and classification present typical tasks of supervised learning. Unsupervised learning is characterized by the lack of those supervision signals and needs to find patterns in the input data. Learning representations is a typical unsupervised learning task and presents a central topic in DL. It separates into three different types: lower dimensional representations (dimensionality reduction techniques), sparse representations, and independent representations. [GBC16, p. 147]

A DL algorithm comprises four components: a cost function, an optimization criterion, a model and a dataset. This framework helps to structure our thinking when dealing with ML algorithms or designing those. The learning process of such algorithms can be divided into training, evaluation, and inference. **Training** relates to the anticipation of the underlying relations between input and output by a model in order to optimize the criterion, i.e. reducing the training error. **Evaluation** quantifies how well the model performs on previously unseen, new data and thus estimates its generalization error. **Inference** is the application of a trained and evaluated model in a live system where humans or machines infer predictions by providing respective input.

During training and evaluation of a DL algorithm we need to make a trade-off between performing well on training data and performing well on test data. The first relates to a reduction of bias, whereas the second relates to the reduction of variance. Training a model always involves the risk of overfitting the training data given enough data and sufficient model capacity. The capacity, also denoted as model complexity, regards the ability of the model to capture the underlying function between input and output. Figure 2.6 illustrates the trade-off between both regimes.

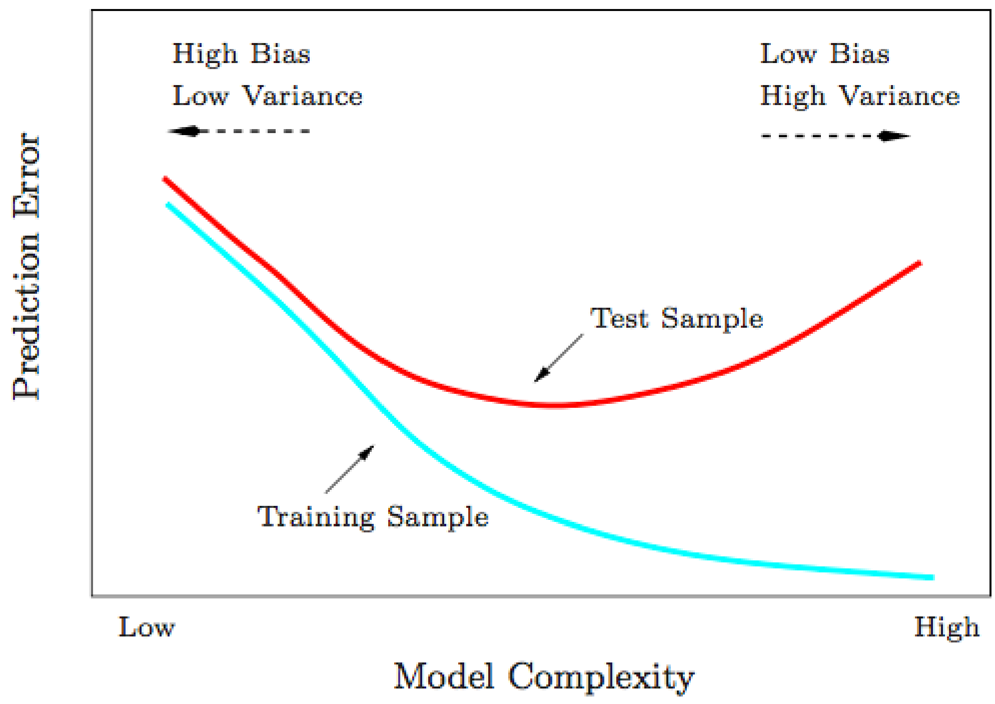

Figure 2.6.: Bias-Variance Tradeoff and Model Capacity [HTF09]

Furthermore, ML models have different types of parameters that are model parameters and hyperparameters. Parameters are the object of the fitting process in which they are adapted in order to satisfy the optimization criterion. Hyperparameters are adaptions of the fundamental parametric model. The given dataset is usually divided into a train, validation, and test set. The train set is used to fit the model (parameters), the validation set guides the selection of hyperparameters, and we estimate the generalization error using a test set.

### 2.2.3. Artificial and Deep Neural Networks

We can perceive an Artificial Neural Network (ANN) as a computational model that acts as a "function approximation machine" [GBC16, p. 169] and which is inspired by biological learning. This section covers the components of a DL algorithm which is based on neural networks. Therefore, I use the framework mentioned before to give a general understanding of the algorithm components that are dataset, model, cost function, and optimization criterion. For brevity, I will focus on the set of supervised learning tasks. The framework used and its components are related to [GBC16, p. 153 sqq.] and [Hea15, p. 1 sqq.].

#### 2.2.3.1. Dataset

The **dataset** is also described as **design matrix** X and consists of individual examples $\mathbf{x_i}, i \in 1, \ldots, m$ where each has $n$ values regarding the independent variables. Thus, each example resembles a row vector in the design matrix. In supervised settings the dataset also contains a **target** Y which is a matrix with values for the dependent variable(s) for all examples.

#### 2.2.3.2. Model

The following model description focuses on feed-forward neural networks which present the most common type of ANNs. The model describes architecture and operations of a network. We start with the Perceptron as the most basic ANN and augment it with multiple outputs concluding with a deep neural network (DNN).

**Neurons** or **units** are the fundamental building blocks of ANNs. They are connected to other neurons in order to receive, process and send information. Figure 2.7 shows the structure and operations of a single processing unit (Perceptron) which is a graphical representation of a mapping function from an input to an output space:

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^k \tag{2.16}$$

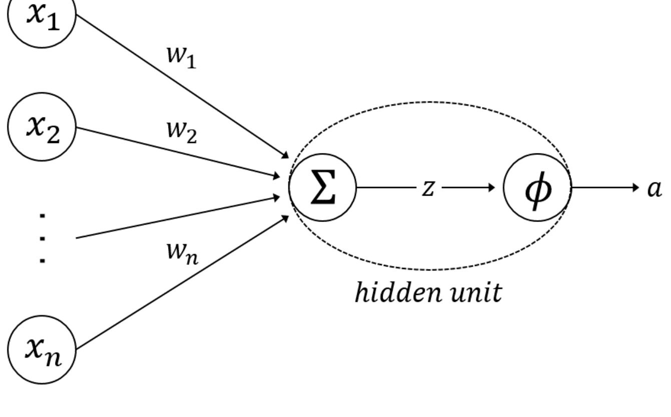

Figure 2.7.: Simple Perceptron Model

The input units $x_1, x_2, ..., x_n$ are placeholders for real numbers that are fed into the network. The connections between inputs (left) and the hidden unit are associated with input-specific weights which are real numbers as well. Two operations constitute the hidden unit. The first operation calculates the **logit** $\boldsymbol{z}$ as the sum over all inputs multiplied with their weights. The second operation performs an **activation function** $\boldsymbol{\phi}$ on the logit and outputs the unit's **activation** value $\boldsymbol{a}$.

$$a = f(\boldsymbol{x}, \boldsymbol{w}) = \phi(z) = \phi\big(\sum_i w_i x_i\big) \tag{2.17}$$

Within a feed-forward network information only flows from input to output. Therefore, we can associate it with a directed acyclic graph that resembles a hierarchy of operations to yield an output.

We can further extend this basic model by adding more hidden units. Figure 2.8 shows a network with three hidden units each being connected to all input units. Consequently, the weight vector $\boldsymbol{w}$ of the previous model turns into a weight matrix $W^{(1)}$ with dimensions $3 \times n$ that contains the weight values for all connections.

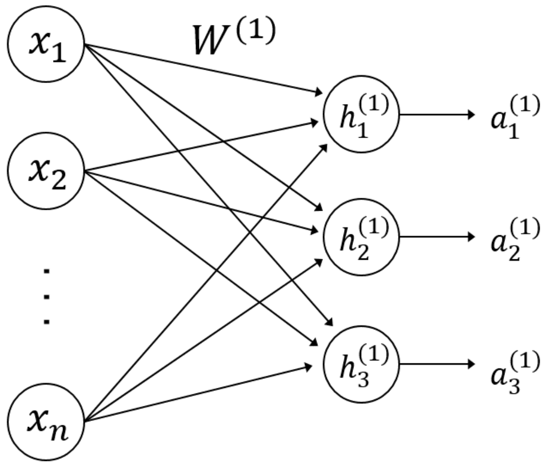

Figure 2.8.: Single Layer Neural Network with Multiple Outputs

The **units** are usually grouped into **layers** and two or more layers constitute a **network**. We distinguish between input, hidden and output layers. The input layer contains all input units, whereas the output layer contains all output units. In between input and output layer there may be several hidden layers that add more computations and thus further complexity to the network. They are called *hidden* as they are not directly exposed to input or output. Hidden unit activations are denoted with $h_i^{(l)}$ replacing $a^{(l)}$ for $l \in 1, \ldots, L$ and subsume the aggregation and activation operation where index $l$ stands for the network layer a unit belongs to and $i$ is the index of the unit within its layer. $L$ is the total number of layers despite the input layer. The input layer can be interpreted as $\mathbf{x} = h^{(0)}$. The output layer is given by $\mathbf{y} = h^{(L)} = a^{(L)}$. Besides the depth $L$ of a network each layer has a certain width $n_l$ which is number of units layer $l$ contains. Therefore, the width of the input layer is usually equal to the number of features $n$ as given by $X$.

Each hidden and output layer is associated with a real-valued weight matrix $W^{(l)}$. As $W^{(l)}$ contains the weight values for connections between all units of the previous layer $l-1$ and all units of the current layer $l$ its dimensions are $n_n \times n_{l-1}$. In addition, we normally add a bias $b^{(l)}$ to each logit in layer $l$ prior to their activation:

$$W^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$$
$$b^{(l)} \in \mathbb{R}^{n_l}$$

As a result, the final formulation for logits and activations becomes:

$$z^{(l)} = W^{(l)}\mathbf{a}^{(\mathbf{l-1})} + b^{(l)} \tag{2.18}$$

$$a^{(l)} = \phi(z^{(l)}) \tag{2.19}$$

By adding further hidden layers, we can increase the depth of the network to get DNNs. Figure 2.9 shows a network with three hidden layers each containing four units. The case of $L = 3 + 1 = 4$ can model more complex tasks due to the hierarchy of concepts that may be learned within each layer.

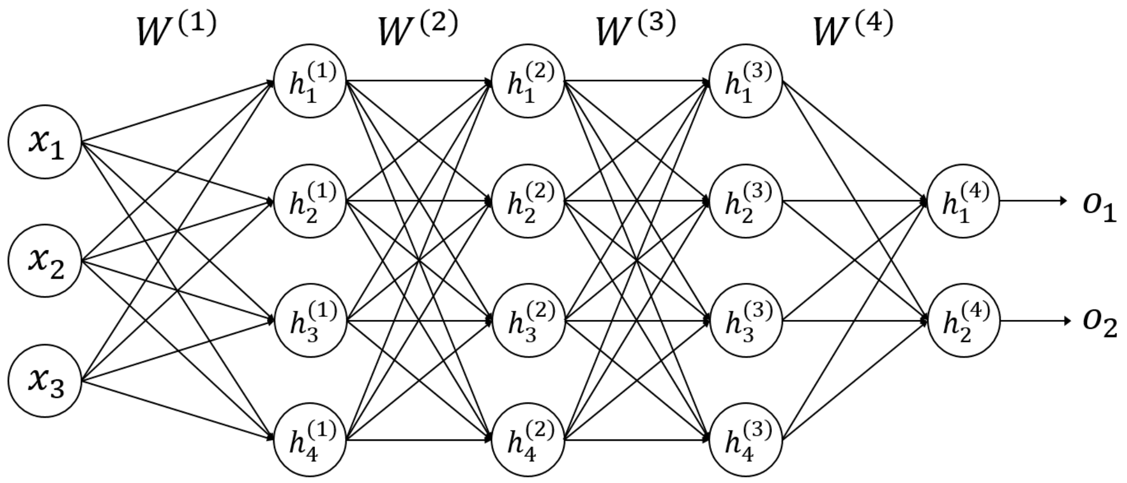

Figure 2.9.: Deep Neural Network with 3 Hidden Layers

Depth and proper activation functions can increase model complexity and introduce nonlinearities. Activation functions operate unit-wise on logits and label the respective unit with the type of operation they perform. There are linear, piecewise linear and nonlinear groups for activation functions. Sigmoid, Rectified Linear Unit (ReLU), and Exponential Linear Unit (ELU) are among the most relevant for neural networks and will be defined as follows and illustrated in Figure 2.10.

The sigmoid or logistic activation is a continuous, differentiable function with an easy derivative and presents a standard choice in DNNs, but suffers from the vanishing gradient problem as described in 2.2.5:

$$\phi(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.20}$$

Problems with the sigmoid activation motivated the development of the ReLU, especially within convolutional neural networks (CNNs) which are naturally very deep. As ReLU is not differentiable for $z = 0$ the derivatives are defined piecewise being 0 for $z < 0$ and 1 for $z > 0$ which drastically reduces gradient computation effort:

$$\phi(z) = ReLU(z) = \max\{0, z\} \tag{2.21}$$

ReLU has different extensions like leaky or the more general parametric ReLU that also yield a nonzero output for $z < 0$. More recently, researchers presented the Exponential Linear Unit (ELU) as another similar type of linear unit that performs better as the widespread ReLU. For $z \geq 0$ it has the same output as ReLU, but for values below zero it still outputs a small negative value involving the exponential function. Thus, also negative signals can be created with a limited impact [CUH16]:

$$\phi(z) = ELU(z) = \begin{cases} e^z - 1 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

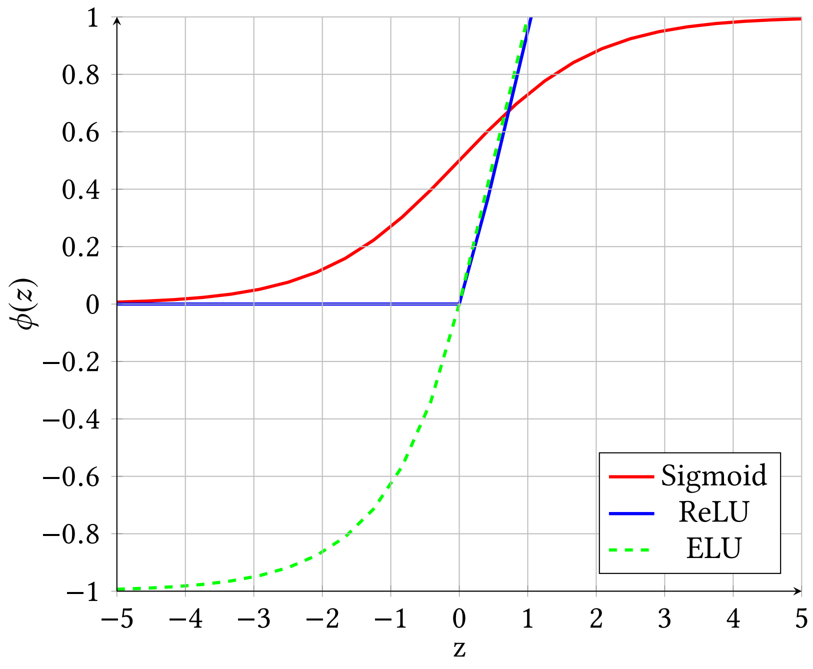

Figure 2.10.: Sigmoid, ReLU and ELU Activation Functions

Despite these activation functions another more distinctive type is worth noting. **Softmax units** impose a probability distribution as they consider all units of a layer with respect to the individual unit. This makes them suitable for classification tasks:

$$softmax(z^{(k)})_i = \frac{\exp(z_i^{(k)})}{\sum_j exp(z_j^{(k)})} \tag{2.22}$$

The fundamental task of training a neural network is to adapt its weights in order to optimize some optimization criterion, e.g. minimize a loss function. This requires choosing a certain architecture to get a model capacity appropriate for the optimization task. Therefore, developing feed-forward networks involves several design decisions. We need to determine the number of layers and thus the network's depth. Besides, we have to specify the number of units each layer comprises as well as the type of units. Finally, we need to decide how layers are connected with each other. All of these concerns can be referred to as *architecture engineering*.

#### 2.2.3.3. Cost Function

As the model approximates a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ with $\hat{y} = f(x; \theta)$, we can quantify the deviation between this estimation and the true value $y$. This deviation is denoted as cost, error or loss and object to some optimization criterion. We denote the cost function with $J(\theta)$ where $\theta$ refers to the weights $W$ and biases $b$ we choose for our model and which we want to adapt.

$$\theta = (\mathbf{W}, \mathbf{b}) = ((W^{(1)}, W^{(2)}, ..., W^{(L)}), (b^{(1)}, b^{(2)}, ..., b^{(L)})) \tag{2.23}$$

We compute the cost on a set of examples $\{(x^{(1)}, y(1)), ..., (x^{(m)}, y^{(m)})\}$ where $x^{(i)}$ and $y^{(i)}$ can be tensors of arbitrary dimensions. Generally, $x^{(i)}$ is a vector containing $n$ feature values where $y^{(i)}$ is a scalar, e.g. class label (classification) or real value (regression). Cross-entropy loss or MSE are the most common cost functions. MSE was already shown as part of the RMSE in chapter 2.1.2. The cross-entropy loss for a binary classification problem with $y^{(i)} \in \{0, 1\}$ is depicted below. It is an appropriate differentiable surrogate for the classification error that provides large gradients which are beneficial for fast convergence:

$$J(x, y; \theta) = -\Big[\sum_{i=1}^{m} y^{(i)} \log f(x^{(i)}; \theta) + (1 - y^{(i)}) \log(1 - f(x^{(i)}, \theta))\Big] \tag{2.24}$$

#### 2.2.3.4. Optimization Criterion

The optimization criterion directs the training process towards a certain direction. Given a cost function that measures the deviation between approximate estimate and true value, our goal is to adapt the network parameters in order to minimize the deviation. Thus, minimization of error, loss, respectively cost is the criterion that guides the training process:

$$\min_{\theta} J(x, y; \theta) \tag{2.25}$$

We need to distinguish between closed optimization which is possible in linear contexts and nonlinear optimization that is typically done by gradient-based techniques like SGD.

### 2.2.4. Training and Regularization

In this section I will cover the training process for a feed-forward neural network that consists of three sequential steps: forward propagation, backward propagation, and gradient descent. Those three steps constitute a training step or an iteration and can be conducted for one (SGD) or many training examples concurrently. It is important to note that backpropagation is just a single step within a training iteration and that it does not constitute the whole learning routine by itself. Across the iterations, we need to track the development of the cost function for the training and evaluation set to obtain training and generalization error.

#### 2.2.4.1. Forward Propagation

Forward propagation is used to obtain an output from a given input. Therefore, the input is propagated forward through the network and thus undertakes transformations and activations to finally deliver an approximate estimate of the target. The forward propagation can be either described recursively starting from the output $\hat{y}$ or starting from the input:

$$\begin{aligned}
1^{st} \text{ hidden layer: } h^{(1)} &= \phi^{(1)}(W^{(1)}x + b^{(1)}) \\
2^{nd} \text{ hidden layer: } h^{(2)} &= \phi^{(2)}(W^{(2)}h^{(1)} + b^{(2)}) \\
&\dots \\
L^{th} \text{ hidden layer: } h^{(L)} &= \hat{y} = \phi^{(L)}(W^{(L)}h^{(L-1)} + b^{(L)})
\end{aligned}$$

The second equation below shows the recursive resolution of $\hat{y}$ until we arrive at the input layer:

$$\begin{aligned}
\hat{y} &= f(x;\theta) = f(x;W,b) \\
&= f(x;W^{(1)},b^{(1)},\dots,W^{(L)},b^{(L)}) \\
&= \phi^{(L)}(W^{(L)}h^{(L-1)} + b^{(L)}) \\
&= \phi^{(L)}(W^{(L)}\phi^{(L-1)}(W^{(L-1)}h^{(L-2)} + b^{(L-1)}) + b^{(L)}) \\
&= \dots \\
&= \phi^{(L)}(W^{(L)\top} \dots \phi^{(1)}(W^{(1)}x + b^{(1)}) \cdots + b^{(L)})
\end{aligned}$$

After computing $\hat{y}$, we can use it to calculate $J(\theta)$ based on the predefined cost function. Backpropagation requires this value for estimating the relative influence of each weight and bias on the final error.

#### 2.2.4.2. Backward Propagation

Backward propagation, backpropagation or backprop involves measuring the cost and propagating the gradient of the cost with respect to weights backwards through the network. This is required to incrementally adapt them using gradient-based techniques. It uses $J(\theta)$ to compute the gradients for each weight and bias along the neural network. Thus, backpropagation needs to calculate:

$$\nabla_\theta J(\theta) = \frac{\partial J(\theta)}{\partial \theta} \tag{2.26}$$

By applying the chain rule of calculus [GBC16, p. 204 sqq.] we can recursively resolve gradients going backwards through the network. It allows us to compute unknown derivatives based on known derivatives. E.g. for $z = f(g(x))$ and $y = g(x)$:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} \tag{2.27}$$

Figure 2.11.: Chain Rule of Calculus

Finally, we calculate $\frac{\partial J(\theta)}{\partial W^{(L)}}$ and $\frac{\partial J(\theta)}{\partial b^{(L)}}$ for all weight matrices and bias vectors in $\theta$. For $W^{(L)}$ we thus receive a Jacobian matrix and a vector of gradients for $b^{(L)}$. [GBC16, p. 204 sqq.] [NNFM17]

#### 2.2.4.3. Gradient Descent

Backpropagation and gradient descent are the core of training DNNs. We can vary the scope of gradient descent by changing the amount of data used for a single forward and backprop pass. We introduced the two extreme cases, SGD and BGD, in section 2.1.2. However, both cases have minor relevance for DL compared to **Mini-batch Gradient Descent (MBGD)** where we choose a batch size $m' : 1 < m' < m$. Thus, for every training iteration, we randomly choose $m'$ examples from the (shuffled) training set without replacement. MBGD is the more common technique as it is less memory-intensive and generally provides good convergence given sufficient batch sizes. Afterwards, we calculate the gradients for the loss obtained from this mini batch. We then use the gradients as part of the update rule for weights $W_{ij}^{(l)}$ connecting unit j from layer $l-1$ with unit i from layer $l$ and biases $b_i^{(l)}$ for unit i on layer $l$:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \cdot \frac{\partial J(\theta)}{\partial W_{ij}^{(l)}} \tag{2.28}$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \cdot \frac{\partial J(\theta)}{\partial b_i^{(l)}} \tag{2.29}$$

We generally iterate over several epochs. One epoch is a single full pass over the whole dataset. Thus, we approximately conduct MBGD $\frac{m}{m'}$ iterations per epoch.

#### 2.2.4.4. Regularization

Regularization is a strategy to prevent a learning algorithm from overfitting. We consider different regularization techniques like weight decay, dropout, batch normalization, early stopping, etc. for neural networks of which the first two are briefly described according to [GBC16, p. 228 sqq.].

**Weight Decay** is a widely used technique that intends to keep all weights and biases in $\theta$ small and thus keep them from specializing through choice of extreme and specific values. Therefore, we augment the cost function with a weight decay term that can be viewed as a penalty. We usually use a sum of euclidean norms of all weights and biases that is multiplied with a $\lambda \in \mathbb{R}^+$ that acts as a control parameter to put the penalty term into a moderate relation with respect to the original cost term.

**Dropout** is a neural network specific regularization technique designed to improve generalization by constraining a network during training. Therefore, we first select network layers to apply dropout to. Second, we determine a drop probability for each dropout layer. This is the likelihood for each dropout layer unit to be silenced, i.e. set to zero for a single training iteration. As a result, the restricted network needs to counterbalance the resulting information loss. Counterbalancing means to adapt weights for units such that each becomes less dependent on the dropped units. This limits codependency development between units and finally reduces overfitting. The unit dropping selection changes after each training iteration. For evaluation and inference, we deactivate dropout to exploit the capacity of the full yet better generalizing network.

### 2.2.5. Challenges

DL algorithms also pose some challenges designing and training them. Architecture design, exploding or vanishing gradients, overfitting, long training times, or their low interpretability and thus slight black-box behavior are just a few of them.

**Architecture design** involves the decisions on the hyperparameters that constitute the structure of the neural network. They refer to the number of layers, the number of units per layer, the activation functions, application and degree of dropout, and so forth. As there are many variations of the parameters, the space to search in for the best setting grows exponentially. Since neural networks also normally take considerable training time, classical grid search has limited applicability. [ZL16] illustrated this problem and used Reinforcement Learning to find the best neural network architecture. However, their approach was more efficient and flexible as grid search, they used 800 GPUs to train the model which still presents tremendous computing power involved. Thus, the relief of feature engineering in traditional ML takes its toll considering architecture search in DL.

Moreover, with the nonlinearity of activation functions like sigmoid or tangens hyperbolicus there is the problem of **vanishing or exploding gradients** where gradients become very small or very large due to long multiplicative chains of either small or big gradients during backprop. As a result, they are squashed to the flat regions of the activation functions and get stuck as the gradients of following backprop iterations keep them there. [Gro17] Gradient clipping or the advent of ReLUs helped to alleviate this problem, but introduced the problem of *dying RelUs* [SE15] which describes units that once set to zero always output zero and thus *die* as they no longer transmit information. Adaptions to ReLUs, like leaky ReLUs, which still output small negative signals for values below zero, or the introduction of ELUs helped to alleviate this problem. Nevertheless, ReLUs are still quite popular.

Furthermore, the greater capacity of DNNs to capture nonlinear dependencies has a downside regarding their **interpretability** by humans. Thus, they are sometimes referred to as black boxes as the values in hidden layers are hardly to interpret in order to understand the concepts that are hierarchically build by the network to solve a problem. At least for CNNs that are used in computer vision tasks, hidden layers allow more interpretations. Each hidden CNN layer learns another abstraction level for the final reconstruction of images from their values. Nevertheless, this is just a single domain of networks allowing some insights.

## 2.3. Technologies

This part introduces software and hardware involved in RSs and DL, in particular software libraries like TensorFlow and LightFM that are crucial for my approach. I will also briefly present some more assistive Python libraries and finish with a short paragraph on hardware beneficial for DL.

### 2.3.1. LightFM

LightFM[1] is a lightweight Python library that allows for CF, CBF and hybrid recommendation by implementing several recommendation algorithms. Its centerpiece is a parallelized implementation of MF taking advantage of multi-core architectures. Furthermore, it provides four different evaluation metrics from those presented in 2.1.4.4: Precision@k, Recall@k, AUC, and Reciprocal Rank. LightFM also supports four different loss functions: *logistic, bpr, warp, warp-kos.* Weighted Approximate-Rank Pairwise (WARP) loss was introduced in [WBU11] and presents the most appropriate choice for the optimization of Precision@k ($P@k$) for unary rating scenarios with SGD. The WARP optimization algorithm picks negative examples for a positive and compares their scores. This process repeats up to a rank violation (or up to some maximum number of negative samples) and conducts a gradient step hereafter.

Applying LightFM to recommendations usually follows three steps: First, one needs to initialize the model and optionally change the default parameters for cost function, optimizer, etc. Second, we need to fit the model to some training data. Subsequently, we can make predictions or evaluate the model on a test set using the variety of implemented metrics. [Kul15]

### 2.3.2. TensorFlow

This section presents an overview on what TensorFlow is, how it works and why it qualifies for this research. Therefore, I also refer to the TensorFlow white paper as the source for this short introduction and for further details. [AABB16]

TensorFlow[2] is an open-source ML software library from Google. It was developed by the Google Brain team to replace DistBelief, which was used previously and only internally. Google made it available as of November 2015 under an Apache 2.0 license and already integrated it into many of its products. The software library is originally written in C++, but provides front-ends for both, C++ and Python. Extensions to other programming languages are planned. TensorFlow is used in a broad range of applications from speech recognition to computer vision.

The name *Tensor-Flow* resembles how it works: *Tensors*, which are multidimensional data arrays, *flow* through a set of nodes in a directed graph that perform operations on

[1]https://github.com/lyst/lightfm
[2]https://www.tensorflow.org/

those tensors. Working with TensorFlow can be separated into graph definition and execution. Within graph definition, we define placeholders, variables and operations to implement a machine learning model. The underlying tensor element types are modeled within this phase or inferred during the second. The definitions yield a directed graph which is then executed in order to train a model or to make inferences from an existing model. In addition, we can add control dependencies which are nodes that control operations.

For better illustration, Figure 2.12 shows a simple computation graph taken from Google's TensorFlow white paper. Three colors distinguish the type of nodes: $x$ is a placeholder tensor that receives values during the execution phase, $W$ and $\mathbf{b}$ are tensors resembling variables and initialized with predetermined values. Normally they are set to be trainable which means that their values can be adapted by graph operations to fulfill some criteria, e.g. minimizing a loss function. The remaining purple nodes stand for operations like matrix multiplication or a ReLU-function. Directed edges specify the order of operations. Upon execution, $x$ is fed an array of values that is multiplied with the weight matrix. The result of this operation is the output of the *MatMul*-node. *Add* calculates the sum of its inputs, which are **b** and the output of **MatMul**. Thus, data in the form of tensors flows through the graph passing its edges and adapted by node operations.

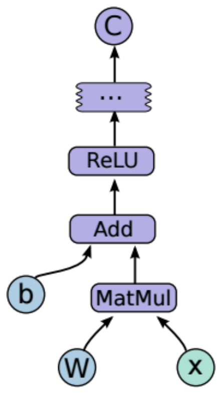

Figure 2.12.: Example TensorFlow Computation Graph [AABB16]

TensorFlow is just one out of a multitude of DL software libraries like Theano, PyLearn2, Torch, PyTorch, Caffe, DeepLearning4j, or MXNet that are used in commerce and research. [GBC16, p. 25] Based on the number of GitHub forks, TensorFlow is currently the most popular DL framework. This popularity constitutes broad support and documentation as one of its major advantages. Furthermore, it poses flexibility and scalability together with its dedicated visualization routine called TensorBoard. It facilitates production by bringing the gap between model and productive operation with TensorFlow serving. Additionally, Google already integrated Keras, another library that builds a layer on top of TensorFlow and thus enables higher abstraction levels and makes model building even easier.

### 2.3.3. Locally Optimized Product Quantization

Locally Optimized Product Quantization (LOPQ) allows for efficient approximate nearest neighbor search by generating low-dimensional integer-valued encodings for high-dimensional data. Given a large $m \times n$ matrix $X$ of high-dimensional real-valued representations it conducts a hierarchical clustering with two major steps. First, it performs product quantization by splitting for each row vector in $X$ splitting it into subvectors that are clustered separately. The resulting cluster assignments establish coarse codes. Second, it applies local optimization meaning that it clusters the residuals between each example split and the cluster centroid. These cluster assignments are the fine codes. Preserving the underlying geometric distances with a tuple of integer-valued coarse and fine codes is a much more efficient representation that easily fits into memory and allows for fast search.
Using LOPQ, we can represent item and user feature vectors in RSs in a way that enables large scale approximate nearest neighbor search. This is necessary to quickly generate candidates without the large computational effort of calculating distances for the original representations. LOPQ was open-sourced by Yahoo and provides implementations for Python and Spark.[3] [KA14]

### 2.3.4. Complementary Python Libraries

The implementation required further Python libraries of which the most important are briefly presented here.

**SciPy** refers to a collection of open-source scientific computation libraries in Python among which NumPy and pandas have considerable importance for the implementation of this thesis and smoothly integrate with packages on higher abstraction layers, like TensorFlow, LightFM, and LOPQ.

**NumPy** is a broadly used package for scientific computing with Python. It allows for extensive linear algebra computations involving tensors of arbitrary dimensions as fundamental data type.[4]

**Pandas** is based on NumPy and introduces database-like structures and operations with DataFrames and Series objects as its main data types.[5]

**Scikit-Learn**[6] is the reference ML library for Python and integrated with NumPy, SciPY, and matplotlib. [PVGM11]

**Runstats** is another open-source library for Python that provides an implementation for calculating online summary statistics and linear regressions in one pass without run-

[3]https://github.com/yahoo/lopq
[4]http://www.numpy.org/
[5]http://pandas.pydata.org/
[6]http://scikit-learn.org/stable/

ning the computation again on all samples that constitute the statistics or regression. This allows for higher efficiency and fewer memory requirements in many live systems.[7]

### 2.3.5. Hardware

Hardware concerns are mainly focused on the increase and parallelization of computation power. The shift to using graphics processing units (GPUs) instead of central processing units (CPUs) has enabled a new speed of development in deep learning. [Hea15, p.176 sqq.] Nevertheless, a new shift might be just at the beginning, from using GPUs to tensor processing units (TPUs) which were just recently announced by Google and made available for their Cloud platform. [DYP17]

[7]https://pypi.python.org/pypi/runstats/

# 3. State of the Art

*"Only animals that calculate probabilities correctly leave offspring behind."*

Yuval Noah Harari
Homo Deus:
A Brief History of Tomorrow

The recent two years have shown increasing research activity in applying DL on RSs. Researcher tried different neural network architectures like autoencoders (AEs) [WWY15; SMG16; VKW17], recurrent neural networks (RNNs) [BBM16; HKBT15; QKHC17], or CNNs [ODS13], standard feed-forward [CAS16], or wide and deep architectures [CKHS16; GTYL17]. Nevertheless, work on this intersection is still in its infancy or receives little attention [Zhe16; BBM16; CAS16]. In the following, I will provide a selection of approaches with distinct architectures and point out successes and shortcomings as well as commonalities and specialities. The list is not holistic and aims to provide a first impression on potential approaches. Papers can be categorized into application-oriented papers, theory-oriented papers, surveys and summaries, and those with potentially relevant contributions that do not explicitly target the intersection of DL and RSs.

## 3.1. Application-oriented Research

The model that won the first Netflix Progress Prize in 2007 can be identified as the first popularized attempt in applying deep learning to recommender systems. [AB12] In this case the winning team submitted a linear blend of algorithms including a Restricted Boltzmann Machine (RBM). [SMH07] present this first CF engine based on neural networks. In this case the authors used different RBMs to generate respective movie recommendations. The Netflix dataset provides over 100 million dated ratings on a scale from 1 to 5 regarding 17,770 items and 480,189 users. The prize of 1 million dollar was awarded to the team that could improve the rating score RMSE of Netflix' own solution by at least 10%. Combining the RBM approach with Singular Value Decomposition (SVD), which is basically MF, the authors perform slightly better than with SVD alone. The RBM captures nonlinearities which are not anticipated by SVD. The authors claim that sparsity creating a non-convex problem was insufficiently addressed by previous naïve approaches.

### 3.1.1. Standard Deep Feed-Forward Networks

[CAS16] use two separate feed-forward DNNs for candidate generation and ranking to recommend videos on YouTube (see Figure 3.1 for overall structure). Using CF (activity history as input) video candidates are selected. These are then ranked using another neural network with other candidate and video features. This model propagates the input from a wide input layer with embedded video watches, search tokens as well as geographic embeddings and uses ReLU activations for training. The architecture follows a *tower pattern* with the number of units halving every consecutive layer. The approach allows to incorporate different ways of candidate generation. The candidate generation network yields a dot product space for efficient nearest neighbor search and is trained with respect to high precision. This contributes a lot to fulfill the latency requirements of 10 milliseconds as in [CKHS16] as well. Candidate ranking is rather special as it is designed as an extreme multiclass classification problem to classify video watches at a specific time using a softmax classifier over millions of items. As well as for Google Play [CKHS16], rich sets of item and user features transformed into high-dimensional user-context and video embeddings fuel the neural architecture and are jointly trained with the other model parameters. The pure supervised training further requires negative sampling and uses only implicit user feedback being positive for completed video views. The online evaluation focused on watch time to prevent distortions caused by *clickbaits*, but results are not reported. Nevertheless, offline results show an approximate 17% improvement of weighted per-user loss outperforming previous methods. The authors point out the considerable feature engineering effort despite neural networks' capability of capturing those inherently.

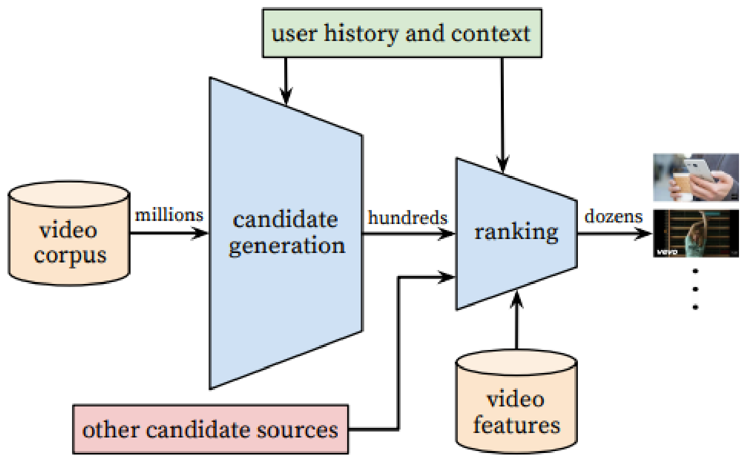

Figure 3.1.: Two-Stage Recommendation System Architecture [CAS16]

[ESH15] use multiple DNNs for different domains and combine their outputs to provide better recommendations in every separate domain. Their approach builds upon Deep Structure Semantic Models (DSSMs) that map users' and items' "high-dimensional sparse features into low-dimensional dense features in a joint semantic space". The generic

model shown in Figure 3.2 is applied to three Microsoft domains: Bing News, Windows AppStore and XBox TV. Thus, it encompasses not only different domains, but also different item modalities (news, apps, and movies). The authors mainly use the textual item descriptions as well as user search and browsing history to extract rich features. In order to increase recommendation quality across multiple domains, users and their preferred item representations are mapped into a shared latent space. User and item domain feature sizes range from 3.5 million (user features) over 100,000 (News) to 50,000 (both Apps and Movie/TV). First, they create tri-grams as textual representation form. In a second step, they separately map those representations into user and item views using a word hashing layer. This reduces the amount of possible tri-grams to a layer representing the 30,000 most common ones. Finally, two layers of each 300 neurons and an output layer of 128 neurons (shared semantic space) conduct the nonlinear transformation of the input. Similarity is maximized in order to recommend items with maximum semantic similarity to a given user. Thus, the multi-view DNN (MV-DNN) learns a mapping between different domains and matches preferred items cross-domain-wise. To scale well the authors seek to reduce the number of user features while maintaining expressiveness and employ techniques like feature selection, K-means, or local sensitive hashing.
The evaluation compared SV-DNN and MV-DNN to SVD, most frequent-items, Canonical Correlation Analysis (CCA), and Collaborative Topic Regression (CTR) in terms of Precision@1 and MRR. The focus is on sampled subsets where each consists of 10 user-item-pairs with one observed and 9 sampled items. The authors show that even SV-DNN significantly outperforms the best CTR models. This effect is further increased with MV-DNN which leads to a Recall@1 increase of 115% for cold-start and 49% for all users. Results could be reproduced on external data and thus the authors bring their use case to non-proprietary data unlike most publications. They conclude that combining rich user features with the nonlinearity of DNNs captures semantics that can not be accurately modeled by traditional models. Scalability was only reported with respect to training but not inference time which lacks significance and sufficient expressiveness to conclude on.

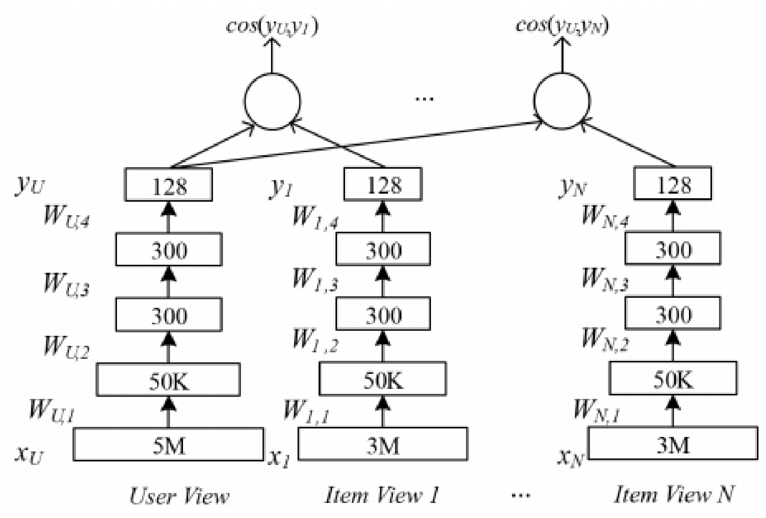

Figure 3.2.: Multi-View Deep Neural Network Model Structure [ESH15]

[GRDB15] present a neural language-based algorithm for personalized product advertising in e-mail clients (Yahoo in this case). It is based on user purchase history extracted from their e-mail receipts. They deal with a rich dataset that consists of 29 million users, 2.1 million unique products from 172 e-commerce websites with 280 million purchases made.
Their approach first creates low-dimensional ($d = 300$) representations of purchased products and secondly, uses a nearest neighbor approach to make recommendation within that embedding space. The prediction step is split into two different ways: product-to-product and user-to-product. The first takes the most recently purchased product and recommends either its top kNN (prod2vec-topK) or the top kNN from top clusters (prod2vec-cluster). The second approach takes into account the whole purchase history of a user for more tailored recommendations, but also introduces a need for more frequent updates.
The model is augmented by a time decayed scoring function for recommendations decaying with the number of days between the current and the day of the neighboring items' purchase. The cold start mitigation strategy proposed is to recommend popular items.

[LZE15] achieve state-of-art performance regarding the recall of user interactions in the music recommendation domain on the Million Song Dataset[1] [BEWL11]. They combine a weighted MF model for CF with a DNN to raise content awareness. This awareness alleviates the cold-start problem and helps to slightly improve pure CF results. The neural network is used to capture nonlinearities of tagging information to generate a high-level content representation used as prior for the latent song representation.

### 3.1.2. Recurrent Neural Networks

[BBM16] apply RNNs to text recommendation and tag prediction. In particular, they use gated recurrent units (GRUs) as a competitive but more efficient successor of long short-term memory (LSTM) units. Since items in different domains often go along with textual descriptions, they generate a latent vector from item-specific text sequences. Doing so, they capture meaning from word order whose lack they criticize as a major shortcoming in bag-of-words or topic models. Consequently, the DNN creates item embeddings for text content partially accompanied by user activity to alleviate the cold-start problem. The final item representation is combined with a user embedding for multi-task learning (recommendation and tag prediction) in a supervised setting to train latent vectors end-to-end on the CF task. Recommendation results measured by Recall@50 and average reciprocal Hit-Rank@10 obtained from scientific paper recommendation (CiteULike) outperform CTR as a state-of-art approach. In terms of tag prediction the results were still competitive. The authors call for exploring additional objectives for multi-task learning with respect to deep architectures and multiple input modalities.

[HKBT15] present a RNN-based approach to session recommendations and defend session-orientation to be more suitable for some contexts. This reduces the burden of

[1]https://labrosa.ee.columbia.edu/millionsong/

user profiling and handles subsequent sessions of the same user independently. Within the same session each consecutive interaction is dependent from all previous. RNNs are capable of handling variable-length data and may store information on the past. Further adaptions, e.g. a new ranking loss function, better suit RNNs for the recommendation task. The actual state of the session can be an item, event history or actual event and serves as input to predict the next event or the likelihood of being the next item. Therefore, the input vector is a one-hot encoding of items the user interacted with during the current session. The authors also propose popularity-based negative sampling by which items are sampled in proportion of their popularity. Thus, if no information is provided on an item, dislike is more probable compared to unawareness as the popularity of the item increases. The authors use two datasets from e-commerce and a video portal to evaluate their approach with respect to MRR@20 and Recall@20. Popularity, similarity-based (kNN) approaches and Bayesian Personalized Ranking MF serve as baselines. On both datasets, they achieve significant performance improvements of 20-30% in accuracy compared to the best baseline approach. Critical for this success was the choice of GRU units and a pairwise ranking loss function.

[DWTS17] propose a RNN-based model to capture the complex and nonlinear dynamics of user and item feature embeddings in a nonparametric fashion. Their approach assumes that features co-evolve over time by mutually influencing each other. Therefore, their DeepCoevolve learns feature embeddings based on interactions to capture the mutual influence. It captures temporal drift, self evolution (features influenced by their prior state), user-item coevolution (item features from interactions influence the user embedding, and vice versa), and interaction features relating to content or context of the interaction. Consequently, recurrent updates of features are performed at specific points in time. Finally, an intensity function takes into account the current embeddings and time since their last update to predict either the next item or the time of the next event. The evaluation on three different datasets shows significantly better results compared to baseline approaches.

[QKHC17] turn the session-based recommender systems with RNN as presented in [HKBT15] into a hierarchical RNN (HRNN). The hierarchy uses single-layer GRUs and consists of two levels, a session level $GRU_{ses}$ and a superordinate user level $GRU_{usr}$. $GRU_{ses}$ models the sequence of items that determines a user-specific session. Therefore, based on the current item ID it predicts scores for each item representing the likelihood of being the next item in the session. $GRU_{usr}$ is used to model the evolution of user interests within and across sessions. This additional level distinguishes HRNN from previous session-based approaches that were solely based on the interactions. The additional leverage of user identifier provides useful information for further session-aware recommendations. The hidden state of $GRU_{usr}$ serves as initialization for the upcoming session $GRU_{ses}$ and is updated by the end of the respective session. The personalization can therefore evolve over multiple consecutive sessions unlike previous approaches while user-level representations remain constant during sessions. The authors also analyze the arising differences when the user-level hidden state can influence every step of the session ($HRNN_{all}$) and not only the first one ($HRNN_{init}$). The training process compares one-hot-encoded items with the score predictions and uses user-parallel mini-batches. To reduce computational complexity, explicit negative sampling is substituted by popularity-based sampling using item IDs from other sessions within the same batch. The evaluation is conducted on a job recommendation dataset (XING) and a YouTube-like platform with distinct outcomes that are attributed to the different underlying user domain behaviors. Nevertheless, the results show a significant improvement over naïve baselines and previous RNN approaches for both, precision and recall. Thus, the mix of session-based recommendations with user profiles confirms its effectiveness with a support for $HRNN_{init}$ over $HRNN_{all}$. However, the authors do not evaluate cold-start items and users as they discard entities with no or few interactions from their analyses. The model structure is shown in Figure 3.3. In the end, the approach supports another view on recommender systems as "a tool that generates recommendation by building and exploiting user profiles." [RRS15, p.9]

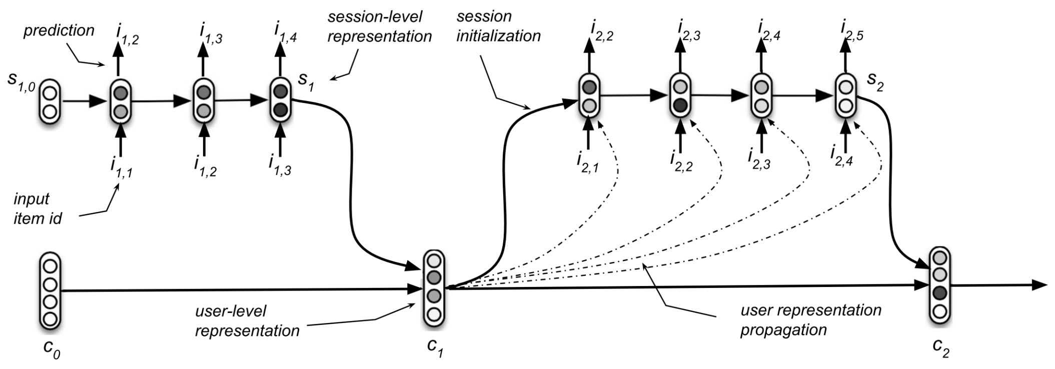
prediction
input item id
session-level representation
session initialization
user-level representation
user representation propagation
$s_{1,0}$
$s_1$
$s_2$
$c_0$
$c_1$
$c_2$
$i_{1,1}$ $i_{1,2}$ $i_{1,3}$ $i_{1,4}$
$i_{2,1}$ $i_{2,2}$ $i_{2,3}$ $i_{2,4}$ $i_{2,5}$

Figure 3.3.: Hierarchical Recurrent Neural Network for Session-Aware Recommendations [QKHC17]

## 3.1.3. Convolutional Neural Networks

[ODS13] use a convolutional neural network (CNN) to music information retrieval. They significantly outperform the traditional CF approach and also alleviate the cold start problem associated with it. The authors use the network to generate latent representations for songs that bridge the semantic gap between song characteristics and their audio signals. It is comprised of two convolutional and two fully connected layers. Experiments are based on a subset of the Million Song dataset with 380,000 songs and a dataset from Last.fm with 500,000 songs. They deal with implicit feedback data as they work with the play counts per song and user. Their weighted MF compares two sources for latent factors. On the one hand, they use a bag-of-words representation, on the other hand, they apply a CNN. As input for the network the authors choose mel-spectograms generated from songs. With respect to $MAP@k$ for $k = 500$ and AUC they achieved large improvements compared to conventional approaches. Besides better accuracy a small qualitative study showed to improve the serendipity likewise diversity of songs with their CNN approach.[2]

[2]Sander Dielemann as one of the authors has provided an interactive blog post summarizing the paper: http://benanne.github.io/2014/08/05/spotify-cnns.html

### 3.1.4. Autoencoders

[WWY15] propose to blend representations learned by a stacked denoising autoencoder (SDAE) with CF for ratings and call their approach collaborative deep learning (CDL) (see Figure 3.4 for overall architecture). Comparing with CTR, which integrates Latent Dirichlet Allocation (LDA) as a topic model with probabilistic MF, they claim it fails in very sparse contexts due to its ineffective latent representations. As representation learning is greatly addressed by DL they employ a SDAE. It learns latent item vectors as a generalized encoding from a bag-of-words representation from raw textual content associated with the items. This is coupled with CF as they argue DL to be inferior capturing similarity and relationship between items. To evaluate their proposed model the authors use three datasets, two with implicit feedback from CiteULike and differing sparsities (0.07% and 0.22%) plus tag-annotations, and the Netflix dataset containing explicit feedback that is augmented by IMDB data to provide textual information through movie plots. The latent representations are compressed from vocabularies of size 8,000 or 20,000 to 50 features. The results are evaluated according to Recall@k for $k \in \{50, 100, 150, 200, 250, 300\}$ and MAP for subsamples of the data and with respect to Collaborative MF, feature-based CF, DeepMusic [ODS13], and CTR. CTR shows to give a strong baseline, but CDL outperforms all approaches across datasets and different train / test set sampling strategies, with even better results for slightly deeper SDAE-architectures. Their reported MAP is almost or more than twice as big as for CTR as the most competitive baseline. They accompany their quantitative evaluation by some qualitative insights and point out the scalability of CDL when it comes to larger datasets.

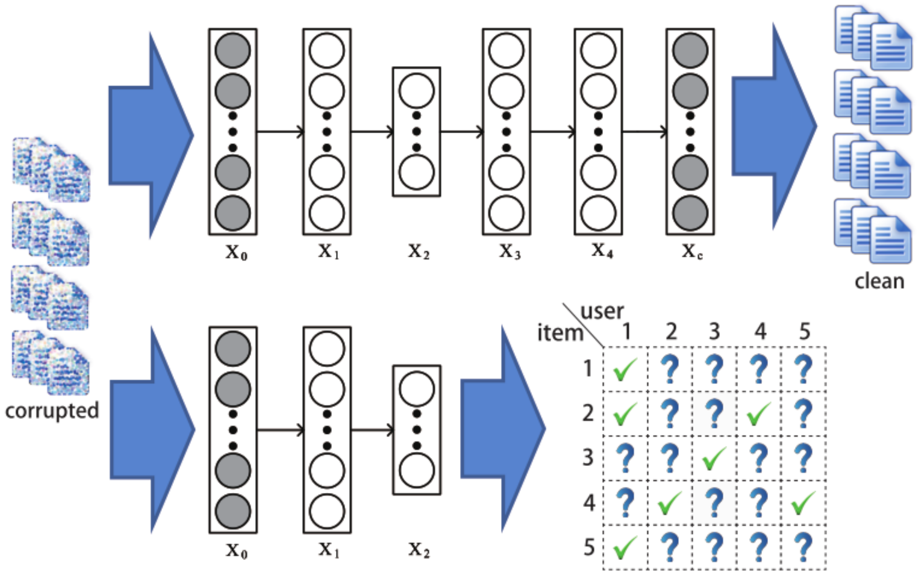

Figure 3.4.: Collaborative Deep Learning Architecture [WWY15]

[WHCZ17] blend time-aware CF with DL using a SDAE as well. Their focus is to alleviate the cold-start problem which together with sparsity presents one of the major challenges for effective recommendations. They further distinguish into complete cold-start (CCS) and incomplete cold-start (ICS). CCS relates to items and users that have no ratings at all, whereas ICS refers to those which have just very few ratings that are insufficient to generate qualitative recommendations. To extend timeSVD++ as the state-of-art time-aware MF-technique they use text content associated with items. This content is subject to a bag-of-words approach that creates vectors as input for the SDAE to create a latent vector. This latent vector is used in conjunction with the CF-inherent rating prediction and thus augments CF by DL-generated content information. Their evaluation is based on the Netflix dataset and movie plots from IMDB as content information. The authors reduce the vocabulary to 20,000 words and generate a compressed latent vector with 50 dimensions. Evaluation is split into CCS and ICS scenarios with the size of latest appearing movies being 100, 200, or 300. They use RMSE as error metric and describe relatively low improvements between 0.37% and 0.47% regarding the ICS-scenario. For CCS the compared techniques were too basic (top of user, top of all, simple average) to call the reported outperformance significant.

[SMG16] present a hybrid approach based on Autoencoders (AEs) that learn nonlinear representations used by CF. The linearity of classic CF approaches is incapable of capturing subtle factors and solve for the cold-start problem. AEs can alleviate this and also integrate side information. The approach is specific compared to other AE-assisted ways as it integrates the matrix within the network for scalability and robustness. The input includes the sparse user/item CF representation and side information on each layer. The AE bottleneck is set between 500-700 units with separated 2-layer-networks that either generate dense representations for users, or items. In order to avoid side information dominating sparse information, its dimensionality is reduced to be lower than the bottleneck size. The authors use *ALS with Weighted-$\lambda$-Regularization* (MF) and *SVDFeature*, feature-based matrix CF (hybrid), as benchmark methods and apply all approaches to three different datasets measuring RMSE. As a result, the item-based CF network *V-CFN* slightly outperforms the user-based *U-CFN* approach with limited influence of side information, however increasing with decreasing count of ratings for entities. Both approaches significantly outperform the benchmarks, in particular when they include additional side information.

### 3.1.5. Other Deep Neural Network Structures

[CKHS16] propose a wide and deep learning architecture for app recommendations at Google Play (see Figure 3.5). [Che16] The authors claim that over-generalization in very sparse and high-rank environments leads to the recommendation of less relevant items. To avoid this shortcoming of deep-only (feed-forward neural network) architectures they augment them with a wide component (generalized linear model). This yields a combination of generalization and memorization targeting recommendation diversity and relevance. The additional memorization component is meant to learn frequent co-occurrences of items or features and to exploit the correlation available in historical data. Furthermore, there is an efficient two-stage recommendation process, which first narrows down the multitude of millions of items to a few hundred candidates before ranking them, which is rather the focus of the W&D model. Both components are jointly trained. User/item embeddings subsume contextual and app impression features (continuous and categorical) to fuel the deep component. DL and the use of embeddings also reduces the burden of feature engineering and increases generalizability turning high-dimensional features into low-dimensional and dense real-valued embeddings. The deep components' layers use ReLUs. The final output is the sigmoid of a weighted average of both wide and deep component and thus leads to joint training of both components. Therefore, it can be interpreted as a probability or preference degree. Their TensorFlow-based implementation shows significant improvements in offline and online evaluation with respect to AUC and online app acquisition (+3.9%) compared to wide- or deep-only architectures, even though, online improvements were greater. Despite the higher quality of recommendations, strict latency requirements around 10 milliseconds could be approximately met with 14ms using a highly parallelized implementation and thus they also prove to be scalable.

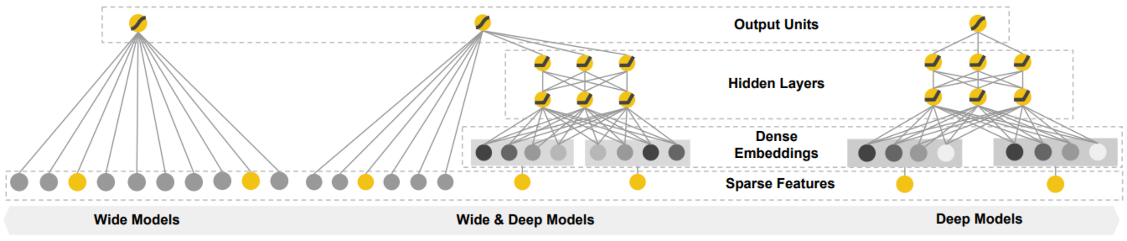

Figure 3.5.: General Wide and Deep Model Structure [CKHS16]

[ZNY17] present their Deep Cooperative Neural Network (DeepCoNN) to learn ratings from textual reviews of users and items. Their model is composed of two separate networks that create hidden, low-dimensional embeddings for items and users as well as a shared layer at the top that combines both representations to predict a rating. The inputs aggregate all reviews written by a user or written for a specific item, respectively. Words are transformed into word embeddings forming a lookup layer. Each separate network involves a CNN-like architecture with a convolution, max-pooling and fully-connected layer. The authors claim scalability, ability for continuous updates and better cold-start behavior as advantages of their approach. The convolutional architecture exploits textual data capturing its semantic meaning, sentiments and respecting word order to improve rating prediction. Thus, it goes beyond the traditional bag of words method provided rating-dependent modeling of users and items. To estimate the final rating the authors apply a factorization machine. They train their model with an learning rate adaptive version of MBGD, use dropout for regularization and MSE as loss function. Experiments concern three datasets with 1 million, 144 millions and 3 millions of reviews. The authors compare with pure rating based, topic modeling based and DL based approaches and show an average improvement of 8.3% compared to the best baseline methods across all three datasets with the best performance on the most dense ($\approx$ 2% density) dataset. Furthermore, they prove the effectiveness of using word embeddings and cooperation between user and item representations by testing different versions of their model. Finally, they show that the MSE reduction compared to MF is even higher for lower review numbers which supports their claim to alleviate the cold-start-problem up to a certain degree.

## 3.2. Theory-oriented Research

[RFGS09] presents a generic optimization criterion for personalized item rankings called *BPR-Opt* and a generic learning algorithm for optimizing models with *BPR-Opt*. The authors use SGD together with bootstrap sampling for learning. The proposed model is evaluated against CF approaches with adaptive kNN and MF.
The optimization criterion takes pairs of items into consideration. This solves the problem of missing feedback for which the standard technique assigns negative values which inhibits future positive predictions. Thus, items with existing implicit feedback are preferred against those with missing feedback and therefore presenting positive entries. Analogously, negative samples are crafted. Solely those pairs with missing feedback for both items are unknown and automatically forming the test dataset. The task is to provide users with a ranked list of items based on their implicit feedback.
The *BPR-Opt* criterion is a derivative from TPR and TNR which are multiplied in order to maximize the resulting likelihood over all (user, item, item)-triplets given a parameter vector. Despite the fact that the authors use kNN or MF as underlying models, they emphasize the change of the optimization criterion. Thus, evaluating the same models with another criterion leads to better results as the new criterion especially targets ranking problems. The authors stress that "prediction quality does not only depend on the model but largely on the optimization criterion." The model was evaluated on the Rossmann (German drug store chain) and Netflix datasets using a leave-one-out scheme.

[Abe16] outlines an experimental comparison of four different optimization techniques used for CF: SGD, bias-SGD (B-SGD), alternating least squares (ALS), and weighted ALS (W-ALS). The author points out CF's major advantage of being domain free while addressing three main challenges of CF that are cold start, sparsity and scalability. The author evaluates CF with respect to memory- and model-based directions. User rating data helps to compute similarities between users and items which is common for kNN to deliver the top-$k$ most similar neighbors, e.g. by using the cosine similarity. However, this approach performs bad on large datasets. Alernatively, MF uncovers latent factors which can be used for prediction. The four techniques are used to learn the parameters for MF experimenting with different datasets. Models are trained using a 70/30 % train/test data split. The author concludes that GD is generally faster than ALS, but ALS performs better on the most sparse dataset (MovieLens). Furthermore, he concludes that ALS is more scalable than SGD.

[LPLH16] extend Bayesian Personalized Ranking (BPR) for multi-channel feedback and present an extended sampling method. Channels describe different feedback types which inherently reflect different levels of meaningfulness or reliability. Prior to using multiple feedback channels one must impose an order within three general feedback level categories: positive feedback, unobserved feedback and negative feedback (see Figure

3.6). Within these categories there may be further different feedback types that need to be ordered. The proposed sampler takes into account these different levels as well as the sample cardinality. Starting with a positive sample, the authors control the ratio of unobserved feedback vs. other subordinary channels. They show that high ratios of unobserved feedback sampling lead to higher accuracy. After sampling, an SGD update adapts the model parameters. The evaluation includes data from three different domains: music recommendation (Kollekt.fm), job recommendation (XING) and movie recommendation (MovieLens). These domains provide explicit or implicit feedback in 3-5 channels. MF-BPR with non-uniform sampling outperforms standard BPR in rather sparse settings, where uniform sampling is competitive in case of the relatively dense MovieLens dataset.

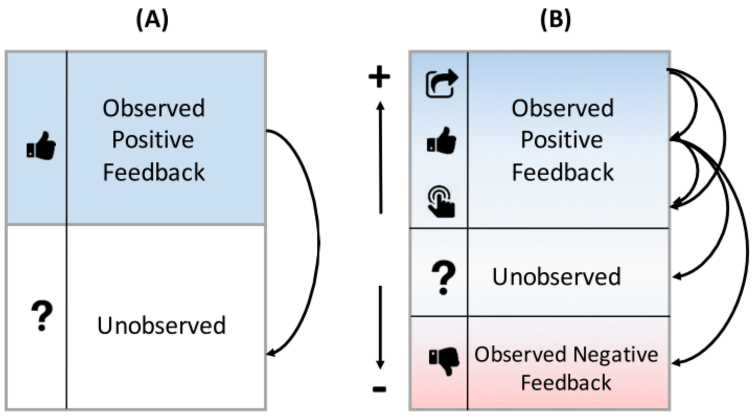
(A)
Observed Positive Feedback
?
Unobserved
(B)
+
-
Observed Positive Feedback
?
Unobserved
Observed Negative Feedback

Figure 3.6.: Feedback Channels and Relations in BPR (A) and MF-BPR (B) [LPLH16]

[TSHL17] present a neural-network-based approach to CF for joint embedding and preference learning. It can be applied to implicit and explicit feedback. The network learns dense representations for users and items from their binary indicator vectors which one-hot-encode the indices they take in their specific sets. Each individual training example consists of a user and two items of which one is strictly preferred to the other given the user's preference relation. Thus, three concatenated sparse input vectors propagate through the network (embedding, flatten, dense layer) with the desired output to be 1 in case of preference and -1 in case of disregard. Embeddings are separately trained per entity before fully connecting them with a dyadic representation of user-item pairs (preferred item and user vs. disregarded item and user). Afterwards, these representations are fully connected to a dense layer each being activated by sigmoid with a weighting of the final sigmoid outputs. Figure 3.7 depicts the network's symmetric architecture. The authors propose to minimize a double ranking loss function to capture the relative ordering error on the binary indicator vectors (target loss) as well as the relative ordering error on the embeddings generated from them (embedding loss). This looks similar to a neural network approach to learn latent vectors as in MF because it only uses interactions, but no content data. The authors propose a minibatch learning technique and sample training data by randomly selecting 10, 20, or 30 items from each user's interactions taking the remaining for testing. They use the normalized discounted cumulative gain @ k and MAP@k to compare their results against those obtained with BoostMF, BPR-MF, and Co-Factor which are different MF methods suited for implicit or explicit feedback. They report significantly better results in most evaluation settings and stress the relevance of proper representation learning by adapting their loss function and respective backpropagation.

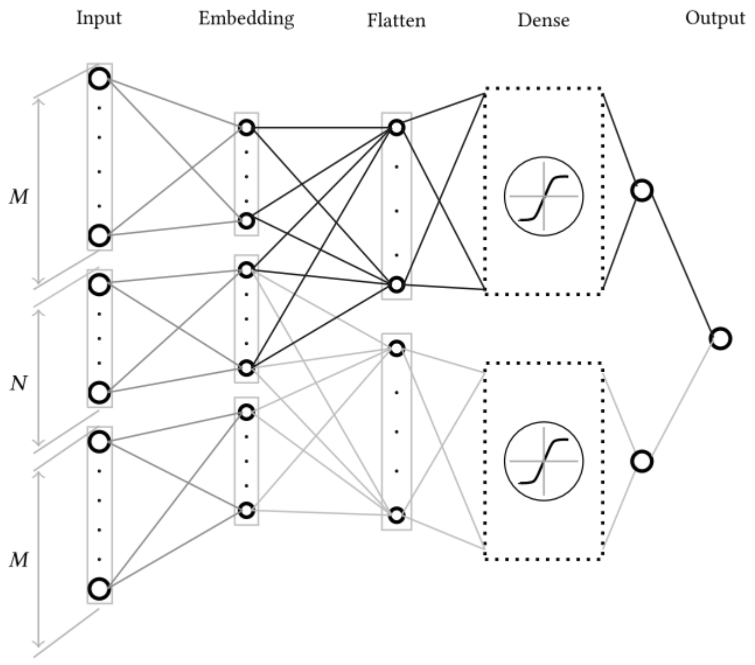

Figure 3.7.: $RecNet_{\alpha,\beta}$ Architecture [TSHL17]

[VKW17] describe an end-to-end graph-based auto-encoder model for the matrix completion task. They build upon the success of DL on graph-structured data and the benefits of embedding a graph-based RS into superordinate graph structures. Thus, they relate matrix completion, i.e. rating prediction, to link prediction. Interactions between users and items resemble an undirected bipartide graph where labeled edges represent observed ratings. The authors focus on explicit rating datasets and interpret unobserved ratings as absent links. Their model generates latent representations for users and items and also allows to incorporate side information like entity features into the resulting dense hidden layer. This representation is consecutively used to reconstruct ratings. Two general components constitute the overall approach as shown in 3.8: a graph encoder and a pairwise decoder model. The first creates respective node embeddings for users and items applying a graph convolutional layer. The latter estimates a probability distribution over discrete rating levels to predict adjacencies for those embeddings and to finally reconstruct links. Final prediction error is summarized by negative log likelihood. To evaluate their approach the authors use the different variants of the MovieLens dataset and show significantly better performance on the rather small and dense MovieLense100k dataset with still competitive results in its larger and more sparse variants. Besides, the authors claim generalization and extensibility as major advantages of their approach. Thus, recent state-of-the-art approaches could be casted into their framework and implicit feedback as well as other side information could easily augment the proposed model.

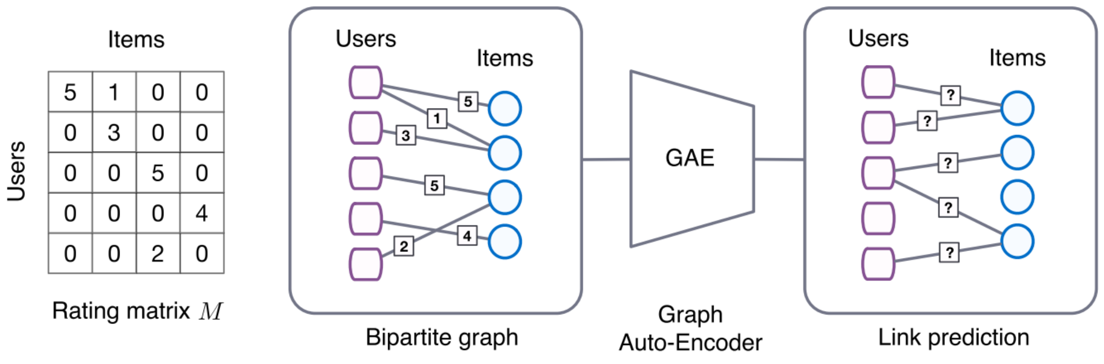

Figure 3.8.: Graph-based Autoencoder Model for Matrix Completion [VKW17]

## 3.3. Surveys and Summaries

[MC07] present a comprehensive framework for analyzing and classifying RSs which they apply to 37 systems. The result is an overview of the current state of art in RSs application in 2007. They refer to RSs as multi-criteria RSs as part of multi-criteria decision making. Initially, they distinguish between different types in term of the base of recommendation (CF, CBF, hybrid), emphasize the difference between predictions and recommendations (prediction problem, Top-N recommendation problem) as well as implicit vs. explicit preferences (feedback). The authors review current classification and analysis approaches, summarize and enrich them with reference to Sirin Roy's general modeling methodology for decision making problems. The final framework is shown in Figure 3.9 and contains 3 main categories on the top level: rationale, approach, and operation.

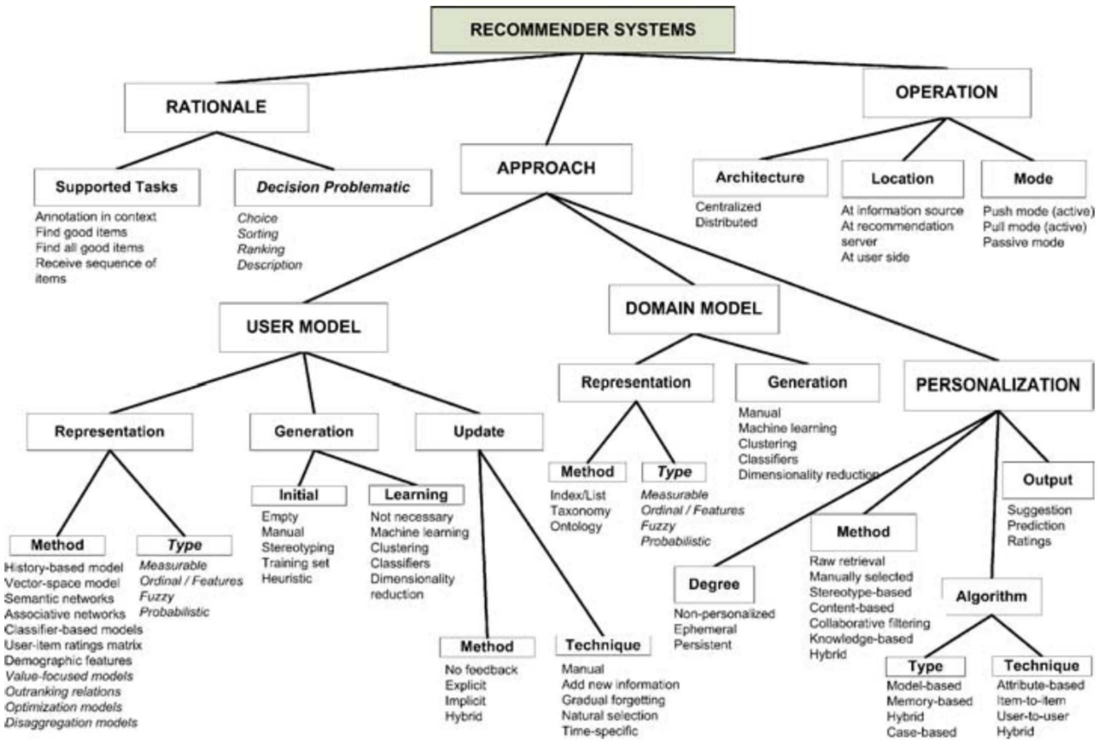

Figure 3.9.: Framework for the Analysis and Classification of Recommender Systems [MC07]

[Zhe16] provide a short overview and critique of different approaches to deep learning on recommender systems (DLRSs). The author subsumes the insufficient exploration of DL on RSs despite the tremendous success of DL in various domains. He eludes RBMs, AEs, CNNs and feed-forward networks in his survey and acknowledges the first promising results. In particular, he points out the capability of automatically learning representations that generalize well and thus help to improve the quality of recommendations. The critique of existing approaches centers around the insufficient use of textual data that is mostly based on the frequency-oriented bag-of-words and thus does not capture significance from word order. Furthermore, the lack of joint modeling of users and items from text needs to be addressed. Thus, combining data from different domains, including meta-data, joint modeling and the full exploitation of textual information are further challenges DL needs to tackle in RSs.

## 3.4. Other Research

[LRJ15] show how to learn individual user representations from heterogeneous sources using neural networks on social network data. Their goal is to provide an "integrated model of homophily in social relations". Hence, integration of different sources and scalability present major challenges. Their approach uses global inference and is scalable by using parallel SGD. Its goal is to infer structured information on user attributes as well as the relationships between their preferences and attributes based on unstructured social media data, e.g. images and text. The authors argue that learned latent features are quite useful as general features for further "downstream tasks".
A neural model is used to learn these latent representations for each user, entity and relation observed from social evidence. It is fed with a user-specific inventory list that contains each user's generated text as a list of words, friend list, relations and attributes. They try to optimize the prediction of all items in user inventory lists.
The authors claim that their inference algorithm is capable of making individual and group prediction crafting peoples' text, social relations and attributes into a single latent vector with the ability to incorporate any kind of information. Since the algorithm captures homophily, latent vector representations could be used for further tasks, e.g. advertising. Besides a detailed description of the model, the authors provide no further information on the type and architecture of neural network they used in their paper.

[HKV08] outline a CF framework developed for a TV recommendation system. First, the authors compare CBF with CF approaches and outline advantages and disadvantages for both. On the one hand, CBF requires no information on past user behavior and does not suffer from the cold-start problem. On the other hand, CF approaches require a history of past user behavior, but are therefore domain free. They distill implicit vs. explicit feedback to abundance vs. quality. Nevertheless, solely using positive feedback in the CF domain misrepresents full user profiles. Furthermore they point out four main characteristics of CF that are non-existing negative feedback, inherent noise, indication of confidence and not preference in implicit feedback as well as appropriate evaluation metrics that exist for this paradigm. Additionally, they elaborate on neighborhood or latent factor models as the two common approaches for learning a CF model.
The paper shows a CF approach using both, kNN and a latent factor model, and uses ALS for training. A shortcoming of latent models is that their results are difficult to explain to the user due to their inherent abstraction. There was no previous work on alleviating this problem, but the authors presented an approach in which the user is shown the respective neighborhood which delivered reasonable results. Another important point the authors stress is the difference between confidence and preference as well as its importance for

the algorithm. It seems important that implicit feedback is transformed into two paired magnitude values.
Their evaluation was done using four weeks as training phase and the consecutive week for testing. The authors describe substantial data preprocessing efforts in order to capture underlying relationships and to distill what preference really means. For their latent features they tried a space ranging from 10 to 200 features and taking all items as neighbors for prediction by cosine similarity.

[ZL16] present a transition from feature to architecture engineering in the field of DL by using RNNs to generate new convolutional and recurrent architectures.
The RNN acts as a controller and samples an architecture $A$ with probability $p$. The resulting child network of architecture $A$ is evaluated and achieves an accuracy $R$. This accuracy is then used to calculate the gradient of $p$ scaled by $R$ to update the controller. As a result, the autoregressive controller gives higher probabilities to architectures with higher accuracies through reinforcement learning by maximizing the expected validation accuracy of the proposed architecture.
Thus, the authors show a new approach to hyperparameter optimization and meta-learning primarily enabling a flexible parameter search space. In the case of convolutional architectures the RNN generates hyperparameters as a sequence of tokens (no. of filters, filter height/width, stride height/width) for one layer and repeats.
The found CNN architecture is evaluated against many human-crafted and performs second-best evaluated on the CIFAR-10 dataset from image recognition. The new recurrent cell performs strictly better than the best human-invented architecture. Nevertheless, the authors applied huge computational power (800 and 400 CPUs respectively) to achieve their results.

[ZDW16] acknowledges that input features in web space are mostly discrete and categorical and criticizes that research centers around continuous features. They propose using DNNs to infer effective patterns from categorical features, e.g. with respect to make CTR predictions. These are superior to shallow and linear models as they do not require manual feature engineering and they embrace better modeling and generalization capacities. They outline city, device type, or ad category as exemplary user features. Each category is associated with a *field* and one-hot encoded. To learn dense representations the authors propose to use embeddings models and two types of DL models: Factorization machine supported neural networks (FNNs) and Sampling-based NNs (SNNs). They pretrain an FNN before supervised fine-tuning adapts weights to minimize a CTR-based loss function. In their evaluation they try denoising AEs and RBMs for the SNN to learn these dense representations. In association with L2-regularization (weight decay) and dropout they achieve competitive results.

## 3.5. Conclusion on the State of the Art

As a result of this part, I provided a thorough answer for my first research question from section 1.2:
*What is the current state of art in the application of deep learning for recommender systems?*
We detect a multitude of DL-based approaches to RSs with advantages and disadvantages as well as trends that indicate further progress. Regarding advantageous aspects we note that DL can better address sparsity, learns effective dense representations for users and items, and provides more flexibility to incorporate multimodal features as well as different domains. In addition, it facilitates multi-task learning and captures nonlinear dependencies. We perceive a significant dominance of implicit feedback contexts and sometimes inappropriate open datasets that constitute the need for a new open datasets that goes far beyond MovieLens to better represent real-world proprietary datasets. This would also serve to make results more comparable across different application domains which is often not the case. As a result, we could observe the development of approaches that are more generic with respect to their improvements and not tied to either music or product recommendation. Despite the great contributions some disadvantages remain. Cold-start is not alleviated in a considerable amount of publications and some also explicitly exclude cold-start items or users from their evaluation. Furthermore, efficiency and scalability were reported quite rarely leaving the contributions incomplete with respect to real-world application. Finally, there is a discontent in the RS community that too many publications focus on improving recommendation relevance despite many other important goals like novelty, diversity, or serendipity.[3] DL can help to address these problems and therefore needs further research effort. The current progress is centering around session-aware approaches and makes great use of recurrent networks to exploit the dynamics within single user sessions.

[3]Result of a plenary discussion during the ACM Recommender Systems Conference 2017

# 4. Approach

*"If we have 4.5 million customers, we shouldn't have one store. We should have 4.5 million stores."*

Jeff Bezos

In this chapter I will outline my end-to-end approach to generate recommendations with respect to high relevance (measured by MAP) in a scalable way. Figure 4.1 gives an overview of the process steps that constitute my approach and also specify the structure of this chapter. The first section describes the data as a history of past interactions along with respective vehicle features. In a first step, I will preprocess these data to make them suitable for the subsequent training. Secondly, I will present a classifier model that jointly trains on preference prediction and embedding similarity for user-item combinations. The classifier is a prerequisite for steps three and four. The third step creates a filtered list of item candidates for each user. These user-specific lists are combined with the user representation to rank user-item combinations using the trained classifier from step two. Finally, the candidates with the $k$ highest predictions become recommendations. They are selected and displayed to the user within the last step. The implementation takes place on three hierarchical levels. The top level invokes a script that orchestrates the pipeline, the mid-level comprises distinct elements of the pipeline whereas the bottom level implements transformation or helper methods and functions as well as arbitrary classes.

## 4.1. Data

The target of my approach is to build a DL-based RS (DLRS) that can be viewed as a hybrid RS as it combines CF with CBF. Therefore, it requires data on features for users and items (CBF) as well as a track of interactions among them (CF). I assume that there are no user-specific features which brings up the necessity to infer user features from item features using their interactions. Thus, the required data can be distilled to an item database and an interaction database. As shown in chapter 2.2, a *profile learner* will create the additional user database by connecting information from item and interaction databases. This also introduces a split view on features: item features become deterministic, whereas user features are stochastic. Deterministic item features can change over time which is accounted for by proper versioning. User features are individually aggregated item features and involve uncertainty which justifies the probabilistic view on them. I will cover this in more detail within the next section and refer to it as *user feature ambiguity*. Last

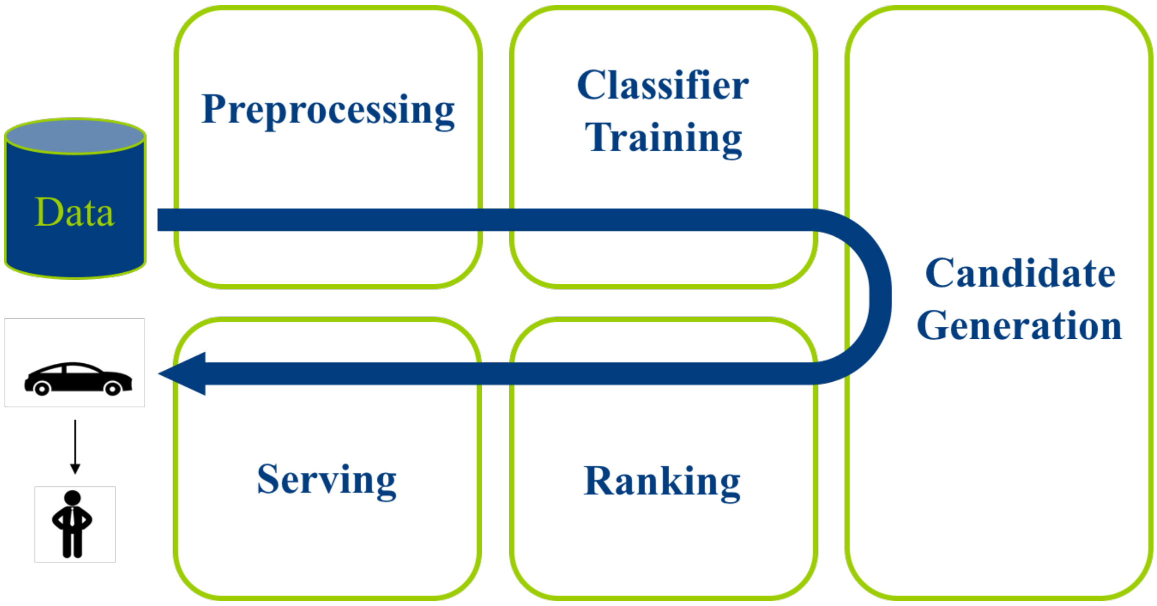

Figure 4.1.: End-to-End Approach for a Deep Learning based Recommender System

but not least, I will distinguish into categorical and continuous features as they require different handling.

Let $U$ be the set of users and $V$ the set of items. Each entity $\mathbf{u}, \mathbf{v}$ is represented by a set of continuous and categorical feature values, $F_{cont}$ and $F_{cat}$. Continuous features take real values: $F_{cont} = \{f_1, \dots, f_n\} \in \mathbb{R}^n$. Categorical features take nominal values, such that $F_{cat}$ is a set of sets $f_{cat}$. Each subset $f_{cat}$ contains $|f_{cat}|$ distinct nominal values the respective categorical feature can take. Latter calculations require numerical comparability. Therefore, I transform nominal into real values using *one-hot* and *one-many-encoding*. As a result, item categorical features become univalent resembling a vector $f_{cat,v} \in \{0,1\}^{|f_{cat}|}$. User categorical features become multivalent resembling a vector $f_{cat,u} \in [0,1]^{|f_{cat}|}$. The multivalent user representation is produced by one-many-encoding which I will cover later. Both encodings requires categorical features to fulfill the following condition:

$$\forall f_{cat} \in F_{cat} : \sum_{l \in |f_{cat}|} f_{cat}^{l} = 1 \tag{4.1}$$

User representations introduce *ambiguity*. This leads to their multivalent categorical representations and also affects the representation of their continuous features. To account for the stochastic nature of continuous user features, we can use different variation and location parameters. The set of those representational parameters is denoted with $M$. Further details on these parameters and ambiguity will be covered later. As a result, we have the following combined representation for both kinds of entities:

$$\mathbf{u} \in \mathbb{R}^{|M| \cdot |F_{cont}| + \sum_{f_{cat} \in F_{cat}} |f_{cat}|} \tag{4.2}$$

$$\mathbf{v} \in \mathbb{R}^{|F_{cont}| + \sum_{f_{cat} \in F_{cat}} |f_{cat}|} \tag{4.3}$$

I combine these representations with unique user and item identification numbers (IDs) $i, j \in \mathbb{N}$ as well as a target variable $r_{i,j} \in \{0, 1\}$. IDs and user/item representations are decoupled since feature values can change over time, but still relate to the the same ID. The classification task is reduced to binary classification for simplicity and interpretability, but also to keep the amount of necessary assumptions low. However, this approach can be adopted to resemble multi-class classification or regression to express finer-grained preferences. For this model, I just distinguish between preference ($r_{i,j} = 1$) and disregard ($r_{i,j} = 0$).

Finally, a single example is a tuple:

$$s = (i, j, \mathbf{u}, \mathbf{v}, r_{i,j}) \tag{4.4}$$

The whole dataset is a combination of positive (1) and negative (0) examples:

$$S = S^{+} \cup S^{-} = \{s \in S | r_{i,j} = 1\} \cup \{s \in S | r_{i,j} = 0\} \tag{4.5}$$

The following part will apply the abstract definitions to the vehicle recommendation task and section 4.2 will then outline the details of preprocessing that lead to the final dataset $S$.

### 4.1.1. Vehicle Recommendations

Data for this research sources from a online market for buying and selling used and new vehicles. The relevant databases contain information on platform interactions of which three are relevant for my research: **item storage**, **user tracking** and **user storage**.

The **item storage** holds an event history of creating, updating and deleting vehicle ads on the platform. When an advertiser creates, updates or deletes his vehicle ad, the version number increments. Therefore, by combining item ID and version number we get a valid unique key (identifier) for the table entries. Each entry holds some geo-temporal information on the event as well as all feature values that are valid for the vehicle ad from the event time up to the next event. Table 4.1 lists $|F_{cont}| = 6$ continuous and $|F_{cat}| = 15$ categorical features that were chosen for this thesis. During a prior project, these features have been identified as the most relevant plus there is already a better understanding of them. I will exclude a deeper justification from this thesis for brevity reasons and to focus on the approach itself. Nevertheless, its inherent flexibility allows adaptions to additional or different features.

Table 4.1.: Features for User and Item Description

| $F_{cont}$ | $F_{cat}$ | |
|---|---|---|
| • consumption | • airbag | • make ID |
| • first registration | • category | • model ID |
| • latitude | • climatisation | • previous owners |
| • longitude | • color | • seats |
| • mileage | • condition | • site ID |
| • price | • country | • subcategory |
| | • doors | • transmission |
| | • fuel | |

The **user tracking** database comprises a history of user interactions that relate to view, save, contact or other events. Each table is a track of interactions between users and items denoted by the type of interaction, e.g. a user that clicks on specific items triggers the creation of view events. Save events refer to users bookmarking items, whereas contact events register when users get in touch with a vehicle advertiser.

These two databases contain the necessary interaction and content information to create a hybrid DL-based RS. As mentioned earlier, we need a *profile learner* to infer user-specific preferences resembled by features for users that are based on their past interactions. It can be viewed as the user counterpart for the item storage. Thus, we regularly update the user preferences in the **user storage** database. It will be described in the following together with an explanation of *user profile ambiguity.*

### 4.1.2. User Preference Learning and Ambiguity

The history of user interactions connects users with items. The item storage contains feature information for items. As we lack dedicated user data, e.g. demographics, only items as the targets of user interaction allow us to infer their preferences, and consequently features. This leads to the fact, that user equal item features despite different means to represent them. Thus, we derive user preferences and define them as sets of user features and values inferred from their interactions within a given timeframe. This inference process is called *profile learning*. The variety of aggregation methods and representation means makes user profiles stochastic which is detailed in the following.

Compared to item features, which provide an exact description of an item valid for a specific timeframe, user profiles are more ambiguous. This has an effect on the representation of features across entities. Whereas for items, each continuous feature resembles a real number and each categorical feature resembles a single nominal value, user features are multivalent. For example, a car has a single color, e.g. black, whereas a user may prefer black, white and grey cars with a preference distribution of $(0.5, 0.4, 0.1)$. This requires a more complicated representation. As a result, each user feature value is represented by a dictionary object. Table 4.2 illustrates this by providing an interaction count for an arbitrary user and for a single week in April. This user interacted with vehicles of different colors. The table counts 48 interactions with items, of which 20 did not provide a color value. Given the remaining 28 interactions and their respective color values, we determine the empirical probability distribution for the user's color preference (as shown in the last column). Besides, there are two meta information that could be used within a learning algorithm. First, the number of interactions that were considered to estimate the user's preference distribution, which is 20 in this case. We can perceive it as confidence in the underlying user preferences which are estimated by the empirical probabilities, i.e. how certain is it that the empirical distribution is consistent with the users preferences. Second, the ratio between the number of known item values and the number of all interactions, in this case $(48 - 20)/48 \approx 58.3$ %. We can compare this individual ratio with the ratio for all items with respect to a specific feature. The comparison of user-specific with general ratio could resolve a confidence in how important this feature may be for the user. This is based on the assumption that users preferably interact with items that provide information on features that matter to them.

Table 4.2.: Exemplary User Profile for Feature *Color*

| Color | abs. | rel. |
|---|---|---|
| None | *20* | *0.416667* |
| BLUE | 8 | 0.166667 |
| RED | 8 | 0.166667 |
| BLACK | 8 | 0.166667 |
| SILVER | 7 | 0.145833 |
| ORANGE | 5 | 0.104167 |
| GREEN | 4 | 0.083333 |
| GREY | 4 | 0.083333 |
| WHITE | 2 | 0.041667 |
| BEIGE | 1 | 0.020833 |
| GOLD | 1 | 0.020833 |
| BROWN | 0 | 0.000000 |
| YELLOW | 0 | 0.000000 |
| PURPLE | 0 | 0.000000 |

n = 48 interactions

The example above illustrates the stochastics in terms of a categorical feature, but continuous feature representations are stochastic as well. For example *consumption* which is also inferred by looking at the consumption values of the ads a user interacted with:

$$6.2,\ 6.0,\ 6.4,\ 8.7,\ 8.9,\ 11.9,\ 8.3,\ 7.7,\ 5.8,\ 7.8,\ 8.0,\ 8.5,\ 7.4,$$
$$10.6,\ 7.4,\ 6.3,\ 5.6,\ 8.3,\ 8.8,\ 9.6,\ 9.7,\ 6.5,\ 9.6,\ 8.6,\ 8.9,\ 7.6$$

There are 26 observed values with a mean consumption of approximately 8.04 and a standard deviation of 1.52. Mean and standard deviation are the most relevant parameters to summarize the multitude of values into a single one. But there may be further like median, other quantiles, minimum, maximum, skewness, kurtosis. The choice of the right parameters to represent user preferences is ambiguous as well. In the following, I refer to the set $M$ containing the different parameters used for the continuous part of a user's profile. I will use $M = \{\mu, \sigma\}$ for my approach to properly represent both views, location and variation, of continuous values. The median is indeed robust to outliers, but due to outlier removal and the Runstats library not supporting the median I retain $\mu$.

These feature inference steps finally yield a **user storage** database providing properly inferred user features. Categorical features are stored as (value, count)-pairs that contain all values observed for a user's categorical feature together with the number of occurrences within the timeframe. For continuous features, single values are already summarized as a Statistics object (see RunStats in chapter 2.3.4). Such an object can return mean, variance, standard deviation, skewness, kurtosis, minimum and maximum as well as the number of values contained in the summary. We can further distinguish user features into the types of interaction they are based on. Thus, each user can have slightly different feature values depending on the items he or she interacted in a specific way with. As these interactions elicit different degrees of preference and combined profiles are better manageable and definite, we need a proper method for profile merging. I propose a Markov-based approach to determine user-specific or general weights for interactions which can be used to create combined profiles as weighted averages of single profiles. The approach is subordinate to the RS and therefore presented in section A.2 of the appendix. We finally have combined user profiles for each selected user for the given timeframe that provide features required by the hybrid CF-CBF and DL-based approach.

## 4.2. Preprocessing

This section describes the preprocessing steps to prepare raw data for later training and prediction. For the evaluation of my approach, I transform data into the proprietary TensorFlow TFRecords format.[1] To compare my approach with other approaches, I also need to provide data suited for CF and CBF experiments using LightFM. Hence, I split this section into a rather technical and a more content-related part. The denomination of these splits is not mutually exclusive, but serves to describe the general notion of processing steps. Target data should distill to the form and sets shown in equations 4.4 and 4.5. Since user and item features are both rather dynamic, time recordings become crucial to form correct samples as result of accurately merged user and item feature values.

[1]https://www.tensorflow.org/api_guides/python/reading_data

### 4.2.1. Technical Preprocessing

This section describes data fetching, proper handling of user profile stochastics, and the creation of examples with time-consistency for user and item feature validity.

To access the databases regarding item storage, user tracking and user profiles, I use Structured Query Language (SQL) and transform the resulting Python lists into dataframes, which is a format used by the Python Pandas package. These are stored in a serialized file format using Pickle, which is part of the Python standard library. Before fetching data on interactions and features, I need to determine a reasonable split into training and evaluation data to clearly separate data fetching. Calendar weeks present a proper time increment as they provide a good balance between too many and too frequent data on the one hand and too much information distortion on the other hand. User preferences can change over time, but are assumed to be nearly consistent within single weeks. Item features change more frequently, e.g. the price generally decreases over time or owners add additional information. Item versioning takes care of this to make changes traceable. Nevertheless, frequent item changes present a severe challenge as they impede an unambiguous relationship between item ID and item representation. Representations change, but still relate to the same ID. Their changes and its proper handling will appear several times throughout my approach and results. Furthermore, I use three different interaction types: view, save, and contact interactions. We adapt queries on user tracking tables to directly join them with rows in the item storage using item ID and version as key. Thus, the fetching process reduces to two directions: interactions joined with item features, and user profiles. Both directions further distinguish between train and test set for which I use a time-oriented 80/20 split. As a result the first four weeks constitute the training and the consecutive week the test set.[2] Figure 4.2 illustrates this time-based split:

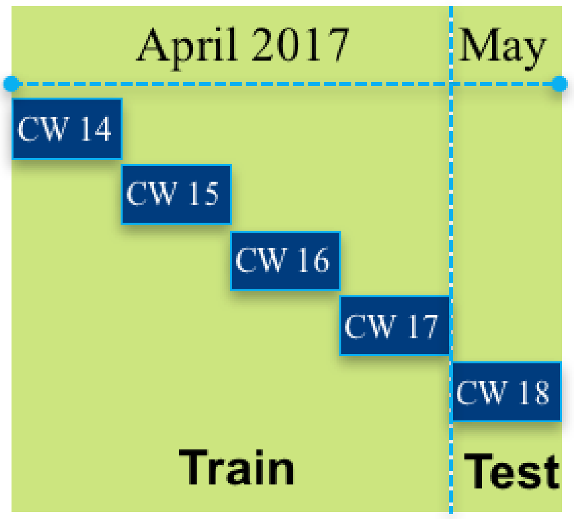

Figure 4.2.: Temporal Train-Test-Split based on Calendar Weeks

Finally, we have the following files as starting point for further processing:

[2]I start with the first full calendar week beginning as of April 3rd 2017.

- Interaction Files
  - Views_Train
  - Saves_Train
  - Contacts_Train
  - Views_Test
  - Saves_Test
  - Contacts_Test
- User Profile Files
  - Profiles_CW14_Train
  - Profiles_CW15_Train
  - Profiles_CW16_Train
  - Profiles_CW17_Train
  - Profiles_CW18_Test

Interaction and user profile files contain the following additional information besides their continuous and categorical feature values (refer to Table 4.1):

- Interaction Files
  - user ID [Integer]
  - item ID [Integer]
  - version [Integer]
  - datetime [Object]
  - date [Object]
- User Profile Files
  - user ID [Integer]
  - createtime [Object]
  - first event [Object]
  - last event [Object]
  - total events [Object]

Each row of a user profile file stands for a single user with a unique ID. Along with the continuous and categorical features there are four other features: *createtime* represents the date and time the profile was resembled; *first event* and *last event* are also datetime objects pointing to the first and last event within the sequence of events used to create the profile; *total events* is a counter holding the number of events that were used for each user; Each profile aggregates information based on a certain period of time. We choose 30 days for this period that directly precede the week the profile is created for. User profiles also change over time, but generating new user profiles after every user interaction presents a huge computational effort with often little effect. Therefore, I chose a refresh interval of 7 days to get in line with the week-based split. This interval appropriately balances computational effort and profile freshness. In addition, only profiles with at least two events within that window are considered. I further constrain my analyses to only those user that are registered and not guests assuming richer data. This reduces the size of data and problem complexity. I assume later extensibility to all users, i.e. also to guest users.

Based on these raw data I apply some data drops and augmentations. First, as I use calendar weeks as time increment, each user profile file gets an additional column *week* that contains an integer value for its corresponding calendar week. The same goes for the interaction files, where I add the same column inferring values from the date of each interaction. After introducing *week* columns, I can drop all other time-related columns across both file types that are: *datetime, date, createtime, first event, and last event.* Thus, I can later merge interactions and items with user profiles using the user ID and week number as unique key. Second, I ensure consistent and equal conditions for my experiments through a further data limitation. I determine the subset of users with a valid profile for each week within the overall training and testing period. From this subset I draw an experimental sample of $n = 100{,}000$ users. Finally, I remove all interactions and profiles that link to users which are not part of the sample.

After these reductions, I close the technical preprocessing with an ID mapping step and introduce consistent and readable column names. Every platform user receives an anonymized ID. I collect IDs for all remaining profiles and items and substitute them with new zero-based IDs from their respective ranges $\{0, 1, \ldots, m-1\}$ and $\{0, 1, \ldots, n-1\}$. This not only increases readability, but also reduces dimensionality. The latter limits computational effort and memory demand for matrix factorization experiments as part of my CF and CBF experiments. To better distinguish between user and item features they receive prefixes **u_** for user and **i_** for item features. In a final step, data types are checked and set to float, integer or object in order to enhance compatibility for later processing.

### 4.2.2. Content-related Preprocessing

This part gives an overview of procedures to prepare the preceding data for training and evaluation. First, I will elaborate on one-hot- and one-many encoding to handle categorical features and ambiguity for items and users. Second, I describe the assumptions regarding outlier removal. Third, I introduce positive labels (1) and present a negative sampling procedure to generate artificial negative feedback (0). Fourth, I will assess and align observed categorical feature values across entities for better consistency. Finally, I will show the split into separate file formats required by different recommendation techniques.

In the first part, I will cover content-related procedures concerning continuous features. The item feature *first registration* is encoded as a concatenation of year (4 digits) and month (2 digits), in which the vehicle's first registration was granted. It has the format 'YYYYMM'. Since this format is nominal and not continuous which is unsuited for further processing, I transform it into seconds since epoch.[3] I also transform the nominal format into its continuous equivalent assuming the start of a respective month as reference date and time for the old representation. This also yields negative values, i.e. when the first registration of a vehicle was earlier than 1970. Afterwards, I replace all continuous user features currently represented as Statistics dictionaries by their means and standard deviations. Consequently, the number of user features grows from $|F_{cat}| = 6$ to $|M| \cdot |F_{cat}| = 12$ for $|M| = 2 = |\{\mu, \sigma\}|$. In order to remove extreme outliers, I use the thresholds described in 4.3 for continuous features which retains over 99% of all items:

After outlier removal, I transform item and user continuous features such that $f_{cont,u} \sim f_{cont,v} \sim \mathcal{N}(0, 1)$. This is called z-normalization and presents a well established method for numerical stability and faster convergence [LBOM12]:

[3]The general reference for dates and times in Python is the so called epoch, which is January 1st 1970.

Table 4.3.: Thresholds for Outlier Removal on Continuous Features

| Feature | unit | min. | max. |
|---|---|---|---|
| consumption | *l/[100 km]* | 0 | 20 |
| first registration | *years* | -20 | 50 |
| latitude | *degree* | 40 | 60 |
| longitude | *degree* | -5 | 25 |
| mileage | *km* | 0 | 1,000,000 |
| price | *€* | 0 | 500,000 |

$$\begin{aligned}\forall f_{cont,u} \in F_{cont,U} : \hat{f}_{cont,u} &:= \frac{f_{cont,u} - \frac{1}{|U|}\sum\limits_{u \in U} f_{cont,u}}{\sqrt{\frac{1}{|U|-1}\sum\limits_{u \in U}(f_{cont,u} - \frac{1}{|U|}\sum\limits_{u \in U} f_{cont,u})^2}} \\ \forall f_{cont,v} \in F_{cont,V} : \hat{f}_{cont,v} &:= \frac{f_{cont,v} - \frac{1}{|V|}\sum\limits_{u \in V} f_{cont,v}}{\sqrt{\frac{1}{|V|-1}\sum\limits_{v \in V}(f_{cont,v} - \frac{1}{|V|}\sum\limits_{v \in V} f_{cont,v})^2}}\end{aligned} \tag{4.6}$$

Thus, I finally get values for features that are in a small range centered around zero which makes the data better suitable for neural networks.

In a second phase, I create comparable and consistent categorical real-valued features from nominal values. Therefore, I introduce *one-hot-encoding* for item and *one-many-encoding* for user features. On the one hand, *one-hot-encoding* transforms nominal to real-valued features by collecting all nominal values that were observed for a specific categorical feature. For each value it creates a new feature, resp. column, representing the combination of categorical features and their corresponding nominal values, and sets its value to 1 if it was observed for that item, or 0 if not. This is done for all items across all categorical features and extends the number of columns from $|F_{cat}|$ to $\sum\limits_{f_{cat} \in F_{cat}} |f_{cat}|$ where $|f_{cat}|$ denotes the cardinality of the set of categorical values observed for that feature. On the other hand, *one-many-encoding* does a similar transformation for categorical user features. The count dictionaries are turned into categorical probability distributions by dividing each count by the number of all observations for the user-feature combination that are not *None*. These combinations of user categorical feature values and their empirical probabilities can now be transformed the same way as for *one-hot-encoding*. The only difference for users is that values can take real numbers in $[0, 1]$. Thus, we do not loose the distributional information and as side-effect introduce feature normalization to $[0, 1]$ which can prevent numeric instabilities during network training. The only information that gets lost is the count of all observations for a specific feature which could be seen as a proxy for confidence in the given probability distribution.

After transforming all categorical features using the corresponding techniques, we have a broad set of values observed for all categorical features which are not necessarily the same for users as for items. To ensure comparability among them, I need to establish the same values and the same order of values for each categorical feature across entities. Therefore, I determine the set of observed categorical feature values for each feature across user and item profiles and determine the union set which is afterwards ordered lexicographically. Finally, I introduce columns to user profiles and item features to account for missing features, to make both entity representations comparable.

Next I introduce binary labels $r_{i,j}$. The dataset $S$ is a union of the subset that contains positively labels examples $S^+$ and the negative subset $S^-$. This is quite easy for $S^+$ as we record implicit, positive feedback signals for which $r_{i,j} = 1$ holds true. I extend the dataset with the respective week $w$ number for each interaction. Furthermore I reduce the set of positive samples to unique combinations by removing all $(u, v, w)$-duplicates which finally leads to:

$$S^+ = \{(i, j, w, \mathbf{u}, \mathbf{v}, r_{i,j}) | r_{i,j} = 1, w \in W = \{14, 15, 16, 17, 18\}\} \tag{4.7}$$

We need to distinguish between positive and negative feedback when using a binary classifier. Unary ratings lack negative feedback. Therefore, I need to sample it using $S^+$. This process is called *negative sampling* for which I propose an algorithm which adapts to the vehicle recommendation domain. $S^+$ includes a rich set of users and items for which all have at least one interaction in the train and test period. This leads to the fact that vehicles without any interaction during that period are not part of $S^+$ which also means that they will not become part of $S^-$. This limitation was made to restrict implementation effort. It is acceptable in the light of the fact that items without interactions within five weeks have minor importance and effect.

I propose to conduct negative sampling as outlined in Algorithm 1 and described as follows: For each week, it separates the positive samples into sets $A$ according to their item values for a specific predetermined categorical feature $F_{cat}$. For each of these sets I extract the contained users $U^+$ and items $V^+$. If there are at least two items and two users, the procedure continues. For all users $u \in U^+$ the algorithm initializes individual negative sampling buckets $S_u^-$ and the set of items $V_u^+$ each user interacted with in the specific week and subcategory. These items are excluded from $V^+$. The remaining user-specific set serves to randomly choose negative samples $v_u^i$ from. The algorithm repeats for all positive user-specific items to sample the same amount as negatives. Each chosen item is associated with user and item ID, week number, user representation as well as a negative label $r_{i,j} = 0$ and becomes an element of the user's negative bucket. After each user, the user-specific negative bucket $S_u^-$ is added to the overall negative samples $S^-$. As a result, each positive sample yields exactly one negative sample which results in a balanced dataset.[4] Only very few user-item-combinations, which do not satisfy the condition of line 7, lack negative samples.

[4]One-to-one positive-negative sampling is also applied in [BBM16].

**Algorithm 1** Category-based Negative Sampling

```
Require: $S^+, W, F_{cat}$
 1: $S^- \leftarrow \emptyset$
 2: for all $w \in W$ do
 3:     for all $f_{cat} \in F_{cat}$ do
 4:         $A \leftarrow \{s \in S^+ \mid w_s = w \wedge v_{f_{cat}} = f_{cat}\}$
 5:         $U^+ \leftarrow \{u \in \{s_u \in A\}\}$
 6:         $V^+ \leftarrow \{v \in \{s_v \in A\}\}$
 7:         if $|U^+| < 2$ OR $|V^+| < 2$ then
 8:             continue
 9:         else
10:             for all $u \in U^+$ do
11:                 $S_u^- \leftarrow \emptyset$
12:                 $V_u^+ \leftarrow \{v \in V^+ \mid (u, v) \in A\}$
13:                 for all $v \in V_u^+$ do
14:                     $v_u^- \leftarrow$ randomly chosen item from $V^+ \setminus V_u^+$
15:                     $S_u^- \leftarrow S_u^- \cup (i_u, j_{v_u^-}, w, u, v_u^-, 0)$
16:                 end for
17:                 $S^- \leftarrow S^- \cup S_u^-$
18:             end for
19:         end if
20:     end for
21: end for
22: Return $S^-$
```

In the specific case of vehicle recommendations, I choose the *subcategory* for $F_{cat}$, which contains about 200 ($= |F_{cat}|$) possible vehicle subcategories. The resulting positive and negative samples now serve as input data for the binary classifier.

Model evaluation requires three partially different datasets for CF, Hybrid-CF-CBF and DL. Furthermore, each dataset needs to be split into a training and test set based on the respective weeks of the training examples. The hybrid models (CF-CBF and DL) use $S$ separated as follows:

$$S^{DL}_{train} = \{s \in S | w_s \in W_{train}\}$$
$$S^{DL}_{test} = \{s \in S | w_s \in W_{test}\}$$
$$W_{train} = \{14, 15, 16, 17\}, W_{test} = \{18\}$$

CF only uses interaction data which is separated as follows:

$$S^{CF}_{train} = \{(i, j)_s \in \{s \in S^+ | w_s \in W_{train}\}\}$$
$$S^{CF}_{test} = \{(i, j)_s \in \{s \in S^+ | w_s \in W_{test}\}\}$$
$$W_{train} = \{14, 15, 16, 17\}, W_{test} = \{18\}$$

Afterwards, $S^{CF}_{train}$ and $S^{CF}_{test}$ are transformed into a sparse matrix format using SciPy, that only stores observed values and returns 0 for all other user-item-combinations. It is important, that the matrix dimensions are equal across train and test set.

## 4.3. Binary Preference Classifier

In the previous sections I have shown the process to obtain a balanced dataset for binary classification. This section now introduces the model for a binary classifier to learn the relationships between user's and item's continuous and categorical features $|F_{cont}|$ and $|F_{cat}|$ as well as their respective labels $p_{u,v} \in \{0, 1\}$. As the model predictions resemble probabilities and not the entries of an interaction matrix, I am going to denote this estimate as $\hat{p}_{u,v}$.

The overall goal is to quantify user preference for items and to use this quantity to impose a user-specific ranking among available items. This ranking then serves as prerequisite for $top - k$ selection. The preference estimate should be a single real number. As a consequence, there are two fundamentally different approaches: regression-based or classification-based. In regression-based scenarios, we need a scoring model to evaluate user-item interactions. Such a model could anticipate different preference intensities, perform time discounting of past events, or grasp potentially higher preference by event repetitions. The resulting multitude of possible score values yields proxies for preference. The RecSys Challenge 2017 [ADEK17] presents a good example for this: it was based on data from the professional network XING and dealt with job recommendations. Therefore, the scoring model valued each user-item pair by adding or subtracting points for different interactions (clicking, bookmarking, replying, recruiter interest, deleting), multiplied with a factor for premium users and added a bonus value in case the job ad was paid. This multitude of potential scores could be used for a supervised regression task. However, classification-based scenarios can also map different preference levels to classes which then could be learned by a binary or multi-class classifier. One might apply a weighting that transforms multi-class activations to a single value for ranking. A rather simple classification approach, but advantageous in terms of interpretability, is binary classification. In this setting the two classes represent disregard and preference. The learning algorithm is trained to discriminate user-item-pairs with respect to these labels. Since $\hat{p}_{u,v}$ takes values in [0, 1], where 1 stands for absolute preference and 0 for total disregard, we can interpret it as the probability of user i to be interested in item j. We can use this likelihood to impose a ranking on items, which is the center of my approach.

The network used for this thesis is inspired by works from [CKHS16; CAS16; ESH15] who used DL architectures for recommendations. The model can be referred to as a **wide and deep model** where learning the deep component is based on user and item embeddings, and learning the wide component is based on their raw categorical feature values. Both components are jointly learned using a cross-entropy loss function. I extend previous works by further augmenting the loss function to also guide training towards embeddings to be geometrically similar for preference, and dissimilar for disregard. Figure 3.5 in chapter 3.1.5 shows the generic network structure as proposed by Google researchers in 2016. I make the following adaptions to this network:

1. Use ELU instead of ReLU as activation function
2. Add a user and item embedding prior to the deep component
3. Introduce joint training for preference and similarity

The first adaption is motivated by [CUH16] which could simultaneously accelerate learning in DNNs and achieve higher classification accuracies compared to different ReLU variants. The second adaption seeks to compress sparse, high-dimensional inputs into dense, low-dimensional embeddings. This reduces memory requirements as after one-hot, resp. one-many encoding, about 97-99% of user and item feature values become zero. Low-dimensional embeddings are required by later candidate generation and ranking steps. Training them in conjunction with preference discrimination yields an integrated learning procedure allowing for better optimization. The last adaption is closely connected to the second and is motivated by the hypothesis that incorporating further objectives relevant for candidate generation and ranking into a common optimization criterion improves final recommendation relevance.

I derive three network variations that differ in terms of their training objectives. The single objective version only learns to minimize classification error, i.e. the right discrimination between favorable and unfavorable user-item combinations. It will be denoted as $DL_{single}$. The multi-objective versions $DL_{multi-cos}$ and $DL_{multi-\overline{cos}}$ jointly train for accuracy and embedding similarity[5]:

$$J_{single}(\theta) = -\frac{1}{|S|} \sum_{s \in S} \left(p_{u,v} \cdot \log(\hat{p}_{u,v}) + (1 - p_{u,v}) \cdot \log(1 - \hat{p}_{u,v})\right), \tag{4.8}$$

[5]For better readability $u_s$ and $v_s$ are shortened to $u$ and $v$ for $s \in S$

$$\begin{aligned} J_{multi-cos}(\theta) = &- \frac{1}{|S|} \sum_{s \in S} \big(p_{u,v} \cdot \log(\hat{p}_{u,v}) + (1 - p_{u,v}) \cdot \log(1 - \hat{p}_{u,v})\big) \\ &+ \lambda \cdot \sqrt{\frac{1}{|S|} \big( \cos(e_u, e_v) - \cos(u, v)\big)^2}, \end{aligned} \tag{4.9}$$

$$\begin{aligned} J_{multi-\overline{cos}}(\theta) = -\frac{1}{|S|} \sum_{s \in S} \Big[ &\big(p_{u,v} \cdot \log(\hat{p}_{u,v}) + (1 - p_{u,v}) \cdot \log(1 - \hat{p}_{u,v})\big) \\ &- \lambda \cdot \big(p_{u,v} \cdot (1 - \overline{\cos}(e_u, e_v)) + (1 - p_{u,v}) \cdot \overline{\cos}(e_u, e_v)\big)\Big], \end{aligned} \tag{4.10}$$

$$\begin{gathered} where \\ \overline{cos}(e_u, e_v) := \frac{cos(e_u, e_v) + 1}{2}. \\ e_u, e_v \in \mathbb{R}^d, u \in U, v \in V : \text{d-dimensional user/item embeddings} \end{gathered} \tag{4.11}$$

$DL_{single}$ minimizes the the mean cross-entropy loss resulting from the comparison of ground truth $p_{u,v}$ and prediction $\hat{p}_{u,v}$. The multi-objective versions work with the well-established cosine similarity as a metric for geometric similarity. $DL_{multi-cos}$ minimizes the sum of mean cross-entropy and a similarity term weighted with $\lambda$. The similarity term is the RMSE between the cosine similarity of both embeddings $e_u, e_v$ and the cosine similarity of user and item raw representations $u, v$. Thus, it penalizes cosine similarity distortions caused by the embedding process and forces the embeddings to maintain the prior geometric similarity. $DL_{multi-\overline{cos}}$ also uses the mean cross-entropy and additionally contains a loss term using an affine transformation $\overline{cos}$ of the cosine similarity likewise to cross-entropy. The target is to maximize similarity between favorable user-item-combinations and minimize similarity for unfavorable combinations. Transforming the cosine similarity pushes its values to the real interval of $[0, 1]$ which is necessary for use in conjunction with $p_{u,v}$. The affine transformation keeps the order, but squeezes the interval, such that afterwards 1, 0.5 and 0 stand for maximum similarity, neutrality and maximum dissimilarity. The best performing classifier will serve for candidate generation and ranking in steps 3 and 4. The role of specially adapted embeddings will also be described there.

In the following, I will deduce $\hat{p}_{u,v}$ from the model in a top-down manner. I refer to Figure A.2 as illustration of the overall model (for a larger model see section A.3 in the appendix. It can be horizontally split into three parts, the top one which contains the **wide and deep components**, the middle one is the **embedding** part, and the bottom one is the **data preprocessing** part which was already described in section 4.2. The final TFRecords-files contain $u_{cont}, u_{cat}, i_{cont}$, and $i_{cat}$ as inputs for the successive embedding middle layer.

Starting at the top, the overall output is the result of a sigmoid activation function applied to the weighted sum of wide and deep components' outputs plus a bias:

$$\hat{p}_{u,v} := \sigma(w_{wide} \cdot z^{wide} + w_{deep} \cdot z^{(3)} + b^{final}) \tag{4.12}$$

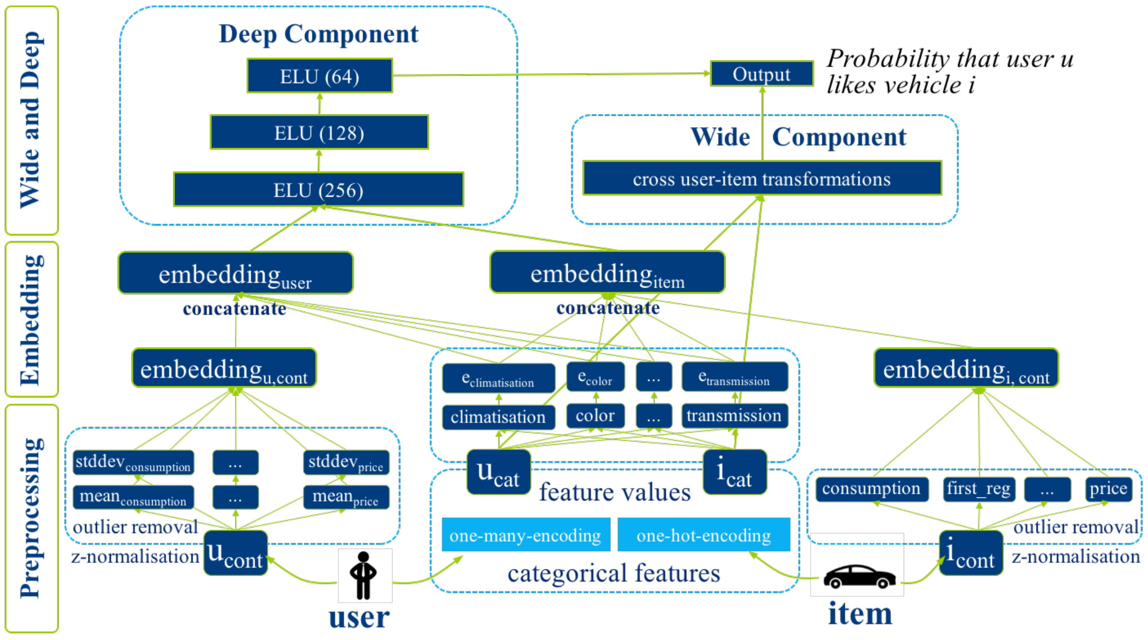

Figure 4.3.: Wide and Deep Learning Classifier with User and Item Embeddings

The **deep component** follows a *tower pattern* [CAS16] where each successive layer contains half the number of units as its previous layer. I followed other architectures by choosing three hidden layers with $L^{(l)}$ denoting the number of units in each layer $l$:

$$L^{(3)} = \frac{L^{(2)}}{2} \qquad L^{(2)} = \frac{L^{(1)}}{2} \qquad L^{(1)} = 2^{\lceil \log_2(|e_u|+|e_v|) \rceil} \tag{4.13}$$

As a result, we need four sets of weights, connecting layer three with the final deep component output, layer two with layer three, layer one with layer two, and linking the embeddings with layer one. I use a fully-connected deep neural network for this purpose. All weight matrices go along with biases. I use ELU to activate the logits of each layer. With $\oplus$ I denote the concatenation of two tensors:

$$\begin{aligned} z^{(3)} &= ELU(W_{L^{(3)} \times L^{(2)}} \times z^{(2)} + b^{(3)}) \\ z^{(2)} &= ELU(W_{L^{(2)} \times L^{(1)}} \times z^{(1)} + b^{(2)}) \\ z^{(1)} &= ELU(W_{L^{(1)} \times (|e_u|+|e_v|)} \times [e_u \oplus e_v] + b^{(1)}) \end{aligned} \tag{4.14}$$

In parallel, the **wide component** performs a linear transformation of categorical feature values for users and items. It is possible to extend this transformation to continuous features after discretizing them through binning. This is not considered in this thesis, but presents a valid opportunity among other adaptions or extensions. The wide component is formalized as follows:

$$z^{wide} = W_{1 \times (|u_{cat}|+|v_{cat}|)} \times [u_{cat} \oplus v_{cat}] \tag{4.15}$$

Initialization of weights and biases needs to break symmetries and keep them at low values to prevent exploding or vanishing gradients. I follow the popular practice of Xavier initialization [GB10] to randomly generate starting values for the weight matrices. Values for biases are drawn from the standard normal distribution.

This formalization covers the top part of the overall model and brings us to the middle part with thorough embeddings for user and item features. Both entity-level embeddings are concatenations of embeddings for continuous and categorical features for each entity. Continuous features are embedded in a $d$-dimensional vector with $d = |F_{cont}|$. This only implies reducing the dimensionality of user continuous features as $|M| > 1$. For categorical features the dimensionality of the embeddings depends on the number of categorical feature values. In particular, I set $d$ as the logarithm to base 2 of the number of unique nominal values the corresponding categorical feature can take. This is just one way of deciding reasonable embedding dimensions sizes. Google also poses as alternative $k \cdot \sqrt[4]{n}$ where n stands for the number of unique nominal values. Both design choices follow [CAS16] and the Google Wide & Deep Learning tutorial [Goo17]:

$$\begin{aligned} \forall f_{cat} \in F_{cat} : d_{f_{cat}} &= \lceil \log_2(|f_{cat}|) \rceil \\ d_{F_{cont}} &= |F_{cont}| \\ \Rightarrow d_{e_u} = d_{e_v} &= d_{F_{cont}} + \sum_{f_{cat} \in F_{cat}} d_{f_{cat}} \end{aligned} \tag{4.16}$$

The experimental data comes with 3271 unique categorical nominal values which are transformed into a concatenated categorical embedding with a final size of 73. In combination with the continuous embedding there are finally 78-dimensional dense embeddings for users and items.

$$\begin{aligned} e_u &= e_{u,cont} \oplus e_{u,cat} \\ e_{u,cat} &= \oplus_{f_{cat} \in F_{cat}} e_{u,f_{cat,u}} \\ e_{u,f_{cat,u}} &= ELU(W_{\lceil \log_2(|f_{cat}|) \rceil \times |f_{cat}|} \times f_{cat,u} + b_{f_{cat,u}}) \end{aligned} \tag{4.17}$$

Categorical and continuous embeddings for users and items are learned using separate weights. This makes the final embeddings independent from each other.

Network training utilizes different GD optimizers. The deep component is trained using an optimizer with an adaptive learning rate and momentum that takes into account the current and previous gradient for a weight update.[6] [KB14] The wide component is trained using an FtrlOptimizer. $DL_{single}$ uses the AdamOptimizer for its deep and embedding component as well as FtrlOptimizer for the wide component. $DL_{multi-cos}$ uses the AdamOptimizer for mean cross-entropy adjusting the weights in the deep and a separate AdamOptimizer for the similarity-based RMSE within its embedding component, and FtrlOptimizer for the wide component based on mean cross-entropy. $DL_{multi-\overline{cos}}$ uses an AdamOptimizer for the deep and embedding component with respect to the total loss and the FtrlOptimizer for the wide components tackling mean cross-entropy. For regularization, I apply dropout with a keep probability of 0.5 to the second layer of the deep

[6]In TensorFlow this type is called AdamOptimizer. The implementation can be found in the TensorFlow API on https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer

component and implemented an early stopping rule.

The training takes place on a GPU server that is equipped with 2 NVIDIA Tesla K80 cards, 4 Intel Xeon CPU (3.50Ghz), 64GB RAM and 850GB local disk. Code is implemented based on TensorFlow 1.2.0. Besides the loss function, I also track classification quality measured in terms of accuracy. The accuracy is inferred by mathematically rounding the cross-entropy part of the loss function. Thus, it represents the percentage of examples for which prediction and label (ground truth) are equal. Since TensorFlow also allows for extensively tracking the development of weights, variables, logits, activations and further information, I use TensorBoard that provides a clean GUI to explore the development for tracing down bugs, but also for visualizations of the learning process and outcome. I will provide an exemplary TensorBoard visualization in chapter 5.

The result is a trained binary classifier, whereas trained is defined by the optimal point at which generalization or test error becomes minimal. The trained classifier then serves as instrument for filtering and ranking which are described in the following.

## 4.4. Candidate Generation

This section describes a fundamental problem in RSs as well as some proposals to solve it as part of my overall approach. The section outlines a proposed concept which will not be covered in the evaluation as their implementation and evaluation would go beyond the scope of this work. I will shortly elaborate on the filtering of appropriate ranking candidates. Afterwards4, I will briefly provide a method to obtain a gold standard that determines the best candidates. This method is less efficient, but serves for the evaluation of results obtained from efficient candidate generation methods. I will finish this section with three possible filtering methods.

Filtering or likewise candidate generation describes the process of reducing the set of all available items to a small subset that is likely to be relevant for a given user. The target is to generate relevant items while keeping computational costs low. For example, a user revisits our website and we use the cookie ID to fetch his user profile from our database. Immediately after identification, we intend to personalize our content display for the user through recommendations. This requires to compare the user profile with all available items which can easily accumulate to millions. After these comparisons, we would finally need to extract the $k$ most promising items for display. In order to provide a fast and smooth browsing experience, we have just 100-200 milliseconds to perform all these steps: identification, profile retrieval, item comparison, ranking, serving. Item comparison is a critical part of this chain as a consequence of generally large item corpora. This impedes a direct comparison with all items in an online context. Therefore, we need to come up with strategies to drastically reduce the required time while still finding the most appropriate items. These items are denoted as *candidates*. Candidate sets $C_u \subset V$ are user-specific and potentially context-specific. A candidate set needs to contain significantly fewer items as the corpus: $k < |C_u| \ll n$ while still being appropriate

$\max \sum_{v \in C_u} \hat{p}(u, v)$. The size of the candidate set is kept constant across users and denoted with $c$.

As the necessity for candidate generation becomes obvious, we need to generate strategies to address it. A naïve way for candidate generation is to compare all user-item combinations. This would lead to a *gold standard* that unfortunately suffers from quadratic computational complexity $O(n^2)$. This is certainly too high to fulfill low serving latencies for recommendations. However, it will serve to evaluate the effectiveness of more efficient strategies:

Dealing with $m$ users and $n$ items there are $m \cdot n$ possible combinations constituting a matrix $\hat{P}_{m \times n}$ that contains the preference estimates for all $|u \in U|$ and $|v \in V|$. For the purpose of calculating $\hat{P}$, I use the trained binary classifier from section 4.3. I collect item IDs $j$ associated with candidates $v$ in $C_u$ that show the highest sum of estimates $\hat{p}_{u,v}$:

$$C_u^{GS} := \{(j, v) | \max_{V' \subset V} \sum_{v \in V'} \hat{p}_{u,v}\} \tag{4.18}$$

This results in two matrices $\hat{P}_{m \times c}$ and $J_{m \times c}$ containing the estimates for each user and *best* item candidates as well as the item IDs of those candidates. Ranking these matrices row-wise by decreasing estimate allows picking the $k$ recommendations for $k \in \{1, ..., c\}$. Thus, it serves as *gold standard* to evaluate three more efficient methods for generating relevant and scalable results in production.

For the generation of appropriate candidates, I will show methods that use *Locally Optimal Product Quantization (LOPQ)* (see 2.3.3). I will outline three filtering strategies for vehicle recommendations. All methods presented below require suitable item representations (embeddings) or additional user representations:

1. $LOPQ_{cluster}$: Item Quantization with two-step Cluster-centroid Filtering
2. $LOPQ_{pseudo}$: Item Quantization with Comparable Pseudo-user Representations
3. $LOPQ_{direct}$: Item and User Quantization based on Comparable Embedding Spaces

**1. Item Quantization with two-step Cluster-centroid Filtering**
This method works with item embeddings $\{e_v\}, v \in V$ as by-product from the middle part of the proposed network. I quantize those embeddings using LOPQ. Quantized embeddings enable fast search within the approximate nearest neighbor space. After successfully constructing the LOPQ space, I separately store all item embeddings $e_v \in V$ that relate to cluster centroid codes $\{C_x\}$. This centroid storage $V_C$ with $|V_C| \ll n$ is not a candidate list, but serves as intermediary in order to fetch candidates. The candidate generation is split into two sub-steps. First, calculating $\hat{p}_{u,v}$ for target users $u$ and $v \in V_C$. Second, selecting clusters with $c'$ highest preference values that exceed a certain minimal preference threshold. From this set I draw in total $c$ candidates in relation to their weighted cluster-specific preference values. This is our candidate list $|C_u|$ that serves as input for ranking. The method is sound, but requires many intermediate parameters and due to this variability poses higher effort in determining the right ones.

**2. Item Quantization with Comparable Pseudo-user Representations**
This strategy does not require the intermediate cluster centroid step. However, it yields another effort. I build pseudo user quantizations that aggregate item quantizations. This approach is similar to the way we build user profiles in domains without user-specific features. As user quantizations are per sé not directly comparable with item quantizations since their quantizations are based on independently learned embeddings, we thus try to create user quantizations that are in fact comparable to item quantizations. Therefore, we need to take into account the item embeddings $e_v$ for all $v \in V_u$ that each user $u \in U$ interacted with in the past. We take the weighted average of these embeddings accounting for the relative importance of the related interaction type as well as the frequency of interactions. The result are adapted user embeddings $\hat{e}_u$ that are now comparable with item embeddings and thus can be quantized in the same way. As a result, we can use a user's pseudo-profile $\hat{e}_u$ to query for the $c$ nearest neighbors using the item quantification space. This method feels more natural and involves slightly less design decision to be made.

**3. Item and User Quantization based on Comparable Embedding Spaces**
The third method assumes that user and item embeddings are directly comparable, e.g. achieved by $DL_{multi-\overline{cos}}$. As the classifier trains for high cosine similarity between user and item embeddings in case of preference, it forces similar concepts to be placed in approximately similar dimensions of the whole embedding space. This placing results in embeddings that are comparable across entities. As a result, we can simultaneously quantize both, item and user embeddings, using LOPQ. In case of a recommendation request, we just calculate the user embedding and use it as search query within the quantization space to return a set of $c$ candidates for which we need to ensure that none of them is actually a quantized user. This is the most intuitive and efficient way of generating candidates, however, it requires specific embeddings for LOPQ.

Table 4.4 qualitatively compares the proposed strategies across evaluation effort for hyperparameter optimization, degrees of freedom regarding parameters and assumptions, expected efficiency for candidate generation, and the severity of embedding requirements. Comparatively low implementation effort, few degrees of freedom, high efficiency and low embedding requirements are rated beneficial (++), with the opposites being unfavorable (--). Implementation effort and freedom degrees are closely related as more assumptions and parameters increase the former. Efficiency goes with this as more steps, like the hierarchy describe in the first strategy, slow down the search for candidates. $LOPQ_{direct}$ besides its low freedom degrees and effort is most efficient as the user quantization is comparable to an item quantization which allows to use the user quantization as query for candidates. Nevertheless, the underlying comparability assumption present a high requirement for cross-entity embeddings.

Table 4.4.: Qualitative Comparison of Candidate Generation Strategies

| Strategy | Evaluation | Degrees of Freedom | Efficiency | Embedding Strength |
|---|---|---|---|---|
| $LOPQ_{cluster}$ | -- | -- | - | ++ |
| $LOPQ_{pseudo}$ | + | + | + | ++ |
| $LOPQ_{direct}$ | ++ | ++ | ++ | -- |

## 4.5. Ranking and Serving

The final ranking and serving steps aim to select the $k$ most relevant items from the candidate list $C_u$ for user-specific recommendation requests and recommend those to the user.

In order to select the $k$ most relevant items, I use the trained classifier from section 4.3. Feeding the classifier with user and item representations for items in the candidate set yields $\{\hat{p}_{u,v}\}_c, v \in C_u$. The resulting scores are sorted in descendant order. Finally, I take the item IDs associated with the $k$ highest scores in $(\hat{p}_{u,v})_c^{sorted}$. This list of items constitutes the user-specific recommendations. The whole process can be executed online or offline based on single users or user batches.

Serving bridges the gap between recommendation engine and user experience. Based on the ordered list of $k$ most relevant items, we can personalize the user experience by adjusting the content display. Recommendation-enriched website content defines $k$ placeholders which are now filled with the ranked vehicles, e.g. by displaying their thumbnails linked to the item-specific websites. This step brings the personalization to the user by linking the insights from his past behavior with the item corpus and providing recommendations according to the algorithmically inferred interests.

# 5. Results and Discussion

*"In God we trust, all others bring data."*

William Edwards Deming

In this chapter, I present the results of my approach. The experiments relate to user-item interactions incuding rich feature information for a period of five weeks. To avoid time-related information leaking the first four weeks serve for training, whereas the fifth week is withhold for testing. I will compare three approaches to recommendation generation: CF, hybrid CF-CBF, and DL. In order to obtain competitive results for CF and hybrid CF-CBF, I will use LightFM. For my own DL-based approach I will use my Python *TensorFlow* implementation for three separate models: $DL_{single}$, $DL_{multi-cos}$, and $DL_{multi-\overline{cos}}$. I report on *MAP@k* for different $k$ regarding the relevance of recommendations. I show that my DL-based approach achieves significantly higher precision compared to conventional techniques.

The structure is as follows: First, I will elaborate on the statistical characteristics of train and test data and justify my choice for $k$. Second, I present some general assumptions regarding the evaluation setting and explicate approach-specific assumptions thereafter. Third, I will report on relevance separated by the superordinate approach categories. Finally, I illustrate the best results across all approaches which is the basis for a conclusion in the following chapter.

## 5.1. Dataset Description and Choice of $k$

Training and testing data, $S_{train}^{CF}$ and $S_{test}^{CF}$, relate to $m = 100{,}000$ users and $n = 1{,}711{,}190$ items. This allows for up to $m \cdot n \approx 171 \cdot 10^9$ possible ratings. From these potential amount, we experience 6,474,853 in the training period, and 1,374,153 in the testing period. Assuming a relatively uniform distribution of observations across weeks, we would expect the density of the test matrix to be a quarter of the train matrix, i.e. 0.000946%. The indeed lower number of ratings is mostly caused by the fact that I exclude known positives from the test set which is motivated in 5.2.
However, train and test datasets are very sparse compared to other datasets, like *NetFlix*, *MovieLens*, or *Last.fm*. For reference Table 5.1 presents an overview of the figures for prominent RS datasets to compare them with the vehicle dataset.

Table 5.1.: Density Comparison between Public and Vehicle Datasets (adapted [Gud16])

| Dataset | Users | Items | Ratings | Density |
|---|---:|---:|---:|---:|
| Movielens 1M | 6,040 | 3,883 | 1,000,209 | 4.2600% |
| Movielens 10M | 69,878 | 10,681 | 10,000,054 | 1.3400% |
| Movielens 20M | 138,493 | 27,278 | 20,000,263 | 0.5300% |
| Jester | 124,113 | 150 | 5,865,235 | 31.5000% |
| Book-Crossing | 92,107 | 271,379 | 1,031,175 | 0.0041% |
| Last.fm | 1,892 | 17,632 | 92,834 | 0.2800% |
| Wikipedia | 5,583,724 | 4,936,761 | 417,996,366 | 0.0015% |
| Vehicles All | 100,000 | 1,711,190 | 7,849,006 | 0.0046% |
| Vehicles Train | 100,000 | 1,711,190 | 6,474,853 | 0.0038% |
| Vehicles Test | 100,000 | 1,711,190 | 1,374,153 | 0.0008% |

The distributions of user and item interactions across both datasets are shown in Table 5.2 and provide an overview of user- or item-specific sparsity. Duplicates and known positives are excluded from these data. For example, duplicate removal reduces the training dataset from 8,692,859 to 6,474,853 unique examples.

Table 5.2.: Interaction Distributions for Users/Items across Train/Test Datasets

| | **Train Set** | | **Test Set** | |
|---|---:|---:|---:|---:|
| | Users | Items | Users | Items |
| **Count** | **99,858** | **1,515,161** | **65,870** | **684,319** |
| Mean | 64.84 | 4.27 | 24.57 | 2.36 |
| Std | 107.28 | 6.36 | 41.51 | 2.73 |
| Min | 1.00 | 1.00 | 1.00 | 1.00 |
| $q_{0.25}$ | 11.00 | 1.00 | 4.00 | 1.00 |
| $q_{0.50}$ | 30.00 | 2.00 | 11.00 | 1.00 |
| $q_{0.75}$ | 75.00 | 7.00 | 28.00 | 3.00 |
| Max | 5,235.00 | 768.00 | 1,035.00 | 246.00 |

User and item counts include all entities which had at least one interaction in the respective period. This accounts for the fact that about one third of all users were inactive and two thirds of items not part of interactions in the last week. Regarding cold-start, we have 142 cold-start-users and 196,029 cold-start-items in the test dataset. The ratings are high positively skewed with the mean being more than twice as big as the median across both datasets and entity types. This distortion follows from few users with extreme activity. These are likely commercial car dealer accounts with many people rather than single users sitting in behind one account. The effect can be observed as well for cars where few very popular vehicles attract disproportionately many clicks. Nevertheless, the median user interacts with 30 different items in the train set and 11 in the test set. Compared to the available corpus of over $1.7 \cdot 10^6$ items these are very few. This also illustrates the very challenging task of recommendation generation as there is little signal on the one hand and much choice on the other hand.

Comparing these insights in terms of item popularity with the MovieLens 1M and Last.fm datasets [BEWL11] leads to Figure 5.1. It displays the heavy skewness of recommendation datasets in general and differences in specific datasets. Skewness can be described qualitatively as few items amalgamating a large share of all unique interactions.[1] For example, 1% of items in the MovieLens 1M and Vehicles All datasets attract about 10% of all interactions, whereas the same portion of items gets approximately three times the user attention in the Last.fm music recommendation case. Last.fm even shows 70% of interactions directed towards the 10% most popular items. 10% of items in the other datasets get nearly 43% of overall attention. This can partially be attributed to the short popularity lifespan of songs compared to movies, or even vehicles. The comparison between Vehicles All and MovieLens 1M also shows that despite very different sparsities skewness can be pretty similar.

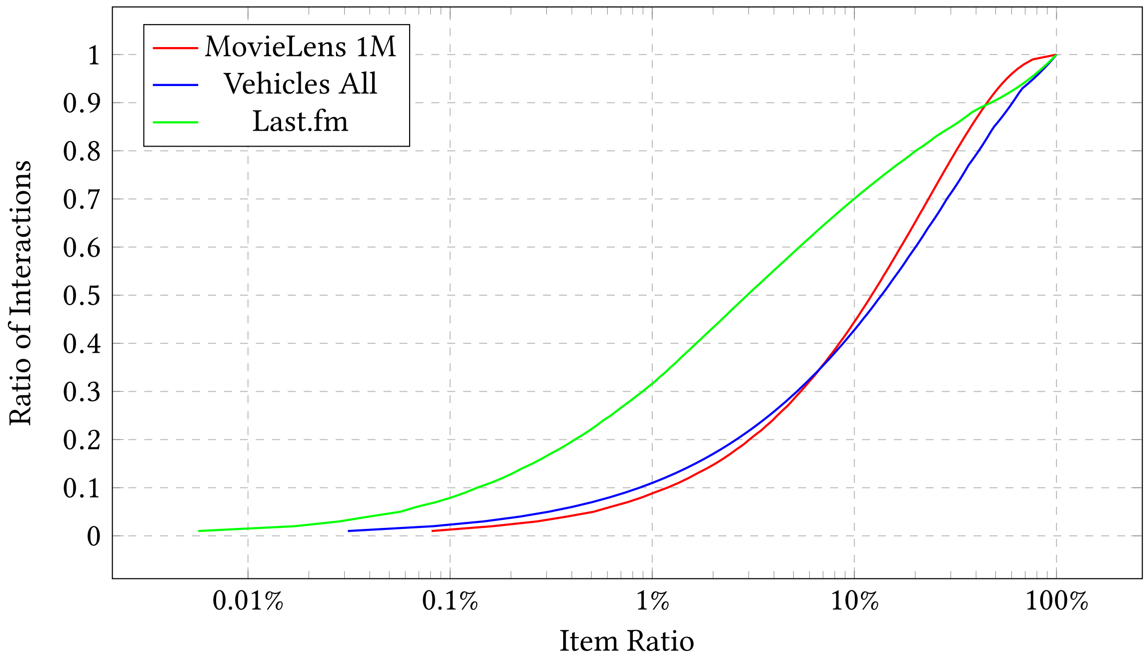

Figure 5.1.: Popularity Skewness for MovieLens 1M, Last.fm and Vehicles All datasets (adapted [LPLH16])

Each user can interact with each item multiple times per week and in multiple fashions. Therefore, the original training set contains 14,133,620 interactions distributed over calendar weeks 14, 15, 16, and 17 as follows: 3,575,017, 3,667,576, 3,715,980, and 3,175,047. The test week counts 3,159,619 interactions. Figure 5.2 depicts the share of different signal types with respect to each week's total interactions:

[1]The number of interactions equals the number of unique users that interacted with the items as redundant user-item combinations were removed.

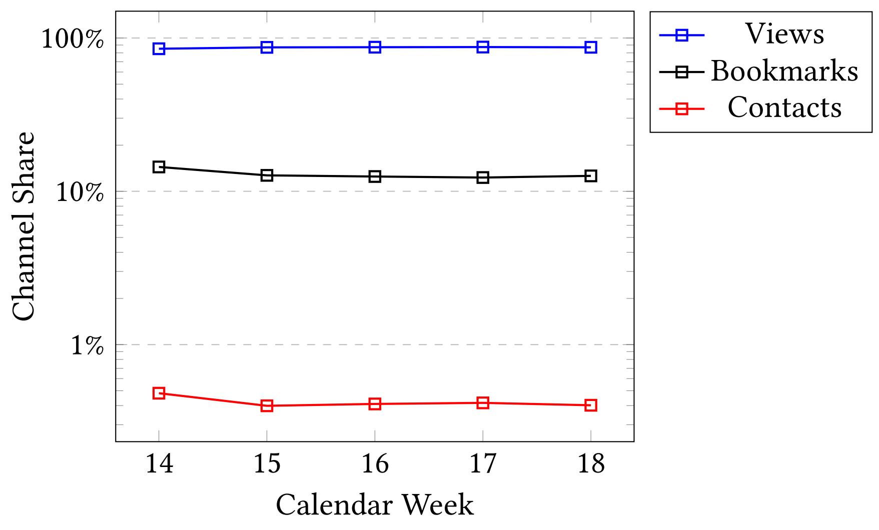

Figure 5.2.: Channel Interaction Share per Week

Weeks 15-18 behave more stable as week 14. The overwhelming majority of interactions ($\sim$ 87%) are views, about 12.6% are bookmarks and only $\sim$ 0.4% are contact interactions. Thus, implicit preference signals with higher underlying preference are more rare and weak signals (clicks) more frequent by over two magnitudes: $\psi(C) > \psi(B) > \psi(V)$ and $h(C) < h(B) < h(V)$.

From these data, I need to make an educated choice for k. There are three drivers for choosing k: application demands, empirical evidence, and comparative standards. The recommendation scenario is application-dependent which would lead to a rather practical choice of k as the number of recommendations requested by our recommendation engine, e.g. on different devices. Since there can be different recommendation situations in a single application this k is not necessarily unique and might lack comparability across different domains. For the vehicle recommendation domain I follow the requirement of $k = 5$. The second driver, empirical evidence, reasons to choose k based on descriptive statistics within the data. From this point of view, I could choose the outlier-insensitive median of test set users as an approximate. This would lead to the choice of $k = 11$ in this case. Finally, I can regard other papers or implementations and guide this choice by different values for $k$ in other domains. In *LightFM*, $k = 10$ is the default value within which is in line with many publications. Finally, I will choose the following set of different k for evaluation: $k \in \{1, 5, 10, 30, 100\}$ with distinguished importance for $k = 5$.

## 5.2. General and Approach-specific Evaluation Assumptions

This part briefly describes the assumptions made for data preparation and the configuration of different baseline models. Its objective is to provide a sound setting for the interpretation of my results. First, I will show some general assumptions that apply to all three model categories. In the second part, I will outline approach-specific assumptions.

### 5.2.1. General Assumptions

1. Interactions that appear in the training data are excluded from the testing data. The removal of these so called *known positives* helps to avoid distortions by already seen items and gives a better projection on the generalizability of each model as only novel predictions are respected. This reduces the original test data by approximately 15.1% from 1,618,191 interactions to the final 1,374,153.

2. *Precision@k* is only based on those users which were active in the test period, i.e. for those 65,870 as mentioned in 5.2. Recommendation generation for inactive users is useless as their *Precision@k* equals zero which would just negatively impair the overall aggregated scores. This procedure is also recommended by [RRS15, pp. 265 sqq.]

3. All models currently work on solely binary information on interactions. Thus, I do currently not incorporate counts for repeated interactions nor do I respect the interaction channel. Although, both signals, frequency and channel, are probably informative about the underlying preference, e.g. higher frequency or more important channel relate to higher preference, they do not fit into a binary approach. To reduce further assumptions and keep models rather simple at first, this is not respected yet, but seen to be significant for a further development of the approach. Nevertheless, frequency and channel, both already influence the used user profile generation (see appendix A.2).

4. As hybrid CF-CBF and DL take item features into account, there is a further assumption to make here. Item features for hybrid CF-CBF represent the last available version of the item experienced in the dataset. It means that there was no full refresh on the item features for the last week. This leads to the following weeks as origin for the most recent item descriptions (18: 684,319, 17: 321,407, 16: 282,210, 15: 214,034, and 14: 209,220. An analysis for a period of two months provides $1.81\frac{days}{change}$ as the average time for which an item is left untouched. Nevertheless, the median number of item versions for this period was 4 such that changes can be assumed to be less frequent and slightly, but not fundamentally, changing the item features.

### 5.2.2. Approach-specific Assumptions

**Collaborative Filtering (CF)**
For this first baseline approach there are few approach-specific assumptions to make. To estimate a good parameter setting and thus a nearly maximum obtainable CF value, I performed grid search to find a good hyperparameter setting which was then used to find the best solutions across different $k$.

**Hybrid Collaborative and Content-based Filtering (Hybrid CF-CBF)**
This model extends CF by adding item and user features. The data is by weeks according to different item and user profile features valid for each week. Thus, one training iteration relates to four sub iterations going through the set of training weeks. Consequentially, user and item features are also separated into different sparse matrices relating to their specific week validity. Profile features were originally fetched on a weekly basis such that there is no time distortion here. For item features, I take their most recent version for each week to represent the specific item. It would be unfeasible to create a relation to the always appropriate version features as this would require to extend item space, which is already large, in a version-specific manner. This would multiply the item space and further reduce density making recommendations as well as resulting evaluation metrics worse. The current week-oriented versioning of interactions and features presents a good trade-off between no and absolutely accurate versioning.

**Deep Learning (DL)**
Regarding the DL-based model, I will use three derivatives of the whole model. The results for each model describe the *gold standard* obtained by predicting $\hat{p}_{u,v}, u \in U, v \in V$ for the test set. I further filter the best $K = 1{,}000$ predictions $\hat{p}_{u,\hat{v}}, \hat{v} \in \hat{V}_u, |\hat{V}_u| = 1{,}000$ together with their associated item IDs $j_{\hat{v}_u}$ for each active user in the test period. These subsets serve to filter the best $k$ items from them. By comparing every user with every item, I can select the best $k$ items to recommend without any uncertainty caused by previous filtering. This *gold standard* is computationally complex and currently takes around 54 hours to run on a NVIDIA Tesla K80 as it propagates $|U| \cdot |V|$ possible pairs through the network. For the current user and item set, this leads to approximately $10^5 \cdot 1.7 \cdot 10^6$ feed-forward passes. As this does not fulfill production requirements, the candidate generation strategies intend to drastically reduce this effort. They exploit user embeddings as efficient search tokens to search for a set of geometrically nearest item embeddings that are then ranked in order to be recommended.

## 5.3. Approach-specific Results

This section illustrates the results separated by approach. Across these distinct results, I will pick the best regarding common $k$ to compare the approaches with each other in the final section of this chapter.

### 5.3.1. Collaborative Filtering

This subsection outlines the CF results for different hyperparameters and $k$ in terms of $MAP@k$. In order to pick a competitive estimate of the maximum CF performance, I perform a grid search with respect to two hyperparameters: the number of embedding dimensions $d$ and the learning rate $\alpha$, where $d \in D = \{50, 100, 200, 300\}$ and $\alpha \in A = \{0.1, 0.03, 0.01\}$. Combined with $k \in \{5, 10, 30\}$ this yields 36 cases to evaluate. Each case runs for 40 epochs to finally evaluate $MAP@k$ in $S^{CF}_{train}$ and $S^{CF}_{test}$. Tables 5.3 and 5.4 show the respective results.
With respect to Table 5.3 we obtain the highest $MAP@k$ consistently across all $k$ for $\alpha = 0.03$ and $d = 100$. Results deteriorate for lower and higher learning rates as well as embedding space dimensions. In terms of $d$, we can take lower $MAP@k$ results for lower $d$ as a sign for underfitting, where $d \geq 200$ creates too complex a model that overfits the training data. This interpretation is confirmed by the results for training data (Table 5.4). We observe a general increase in $MAP@k$ regarding high values for $\alpha$ and $d$. This demonstrates faster convergence using higher learning rates. It also shows that model capacity reflected by $d$ increases and overfits the training data. This leads to poor generalization as we can see by comparing to the test results for the same hyperparameter combinations, e.g. for $k = 10$: $MAP@k^{train}_{d=300,\alpha=0.1} = 0.769451$ vs. $MAP@k^{test}_{d=300,\alpha=0.1} = 0.000136 \ll 0.004017 = MAP@k_{d=100,\alpha=0.03}$.

Table 5.3.: MAP@k on Test Set for CF Hyperparameter Grid Search

| $k$ | $d$ | $\alpha$ | | |
|---|---|---|---|---|
| | | 0.1 | **0.03** | 0.01 |
| 5 | 50 | 0.001436 | 0.004040 | 0.002840 |
| | **100** | 0.000688 | **0.004365** | 0.003266 |
| | 200 | 0.000177 | 0.004163 | 0.003964 |
| | 300 | 0.000047 | 0.003986 | 0.004194 |
| 10 | 50 | 0.001322 | 0.003642 | 0.002736 |
| | **100** | 0.000813 | **0.004017** | 0.003134 |
| | 200 | 0.000259 | 0.003909 | 0.003626 |
| | 300 | 0.000136 | 0.003658 | 0.003860 |
| 30 | 50 | 0.001280 | 0.003199 | 0.002480 |
| | **100** | 0.000926 | **0.003506** | 0.002664 |
| | 200 | 0.000504 | 0.003369 | 0.003094 |
| | 300 | 0.000358 | 0.003300 | 0.003374 |
| *based on 40 iterations* | | | | |

Table 5.4.: MAP@k on Train Set for CF Hyperparameter Grid Search

| $k$ | $d$ | $\alpha$ | | |
|---|---|---|---|---|
| | | **0.1** | 0.03 | 0.01 |
| 5 | 50 | 0.181009 | 0.139448 | 0.049442 |
| | 100 | 0.595175 | 0.278493 | 0.060608 |
| | 200 | 0.832628 | 0.413790 | 0.079709 |
| | **300** | **0.865673** | 0.484370 | 0.093523 |
| 10 | 50 | 0.154815 | 0.128112 | 0.045642 |
| | 100 | 0.723116 | 0.385712 | 0.073411 |
| | 200 | 0.723116 | 0.385712 | 0.073411 |
| | **300** | **0.769451** | 0.436729 | 0.086425 |
| 30 | 50 | 0.122256 | 0.114017 | 0.038418 |
| | 100 | 0.339135 | 0.221117 | 0.047103 |
| | 200 | 0.513935 | 0.311478 | 0.062715 |
| | **300** | **0.565305** | 0.351917 | 0.077169 |
| *based on 40 iterations* | | | | |

Figure 5.3 depicts a general observation: *MAP@k* decreases with increasing k. Combined with the rating distributions in Table 5.2 this can be attributed to the fact that with an increase in $k$ the share of users having at least $k$ ratings decreases. For example, as the median user rates 11 items in the test set, using $k > 11$ for half of the users automatically deteriorates *MAP@k* as we try to predict more items than the user interacted with. Predictions that exceed the number of requested predictions become useless. Therefore, maximum *MAP@k* decreases with an increase in $k$ falling to almost 19.5% for $k = 200$.

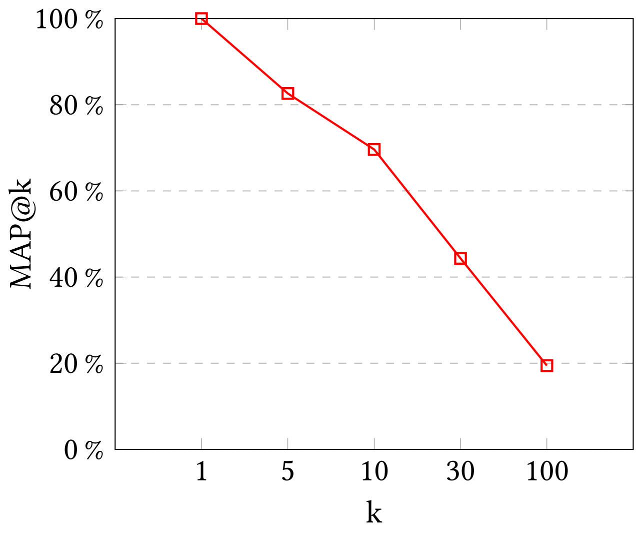

Figure 5.3.: Upper MAP Boundary Across Different $k$

Finally, Figure 5.4 supports the choice for 40 iterations as grid search baseline. It depicts the evaluation of *MAP@k* on $S_{test}^{CF}$ for every 5 iterations of training across different $k$ and the best hyperparameters. We can observe convergence with all curves reaching a plateau after about 40 iterations which is therefore determined to be the grid search base. The convergence results are also in line with the *MAP@k* upper boundary reasoning in the previous section.

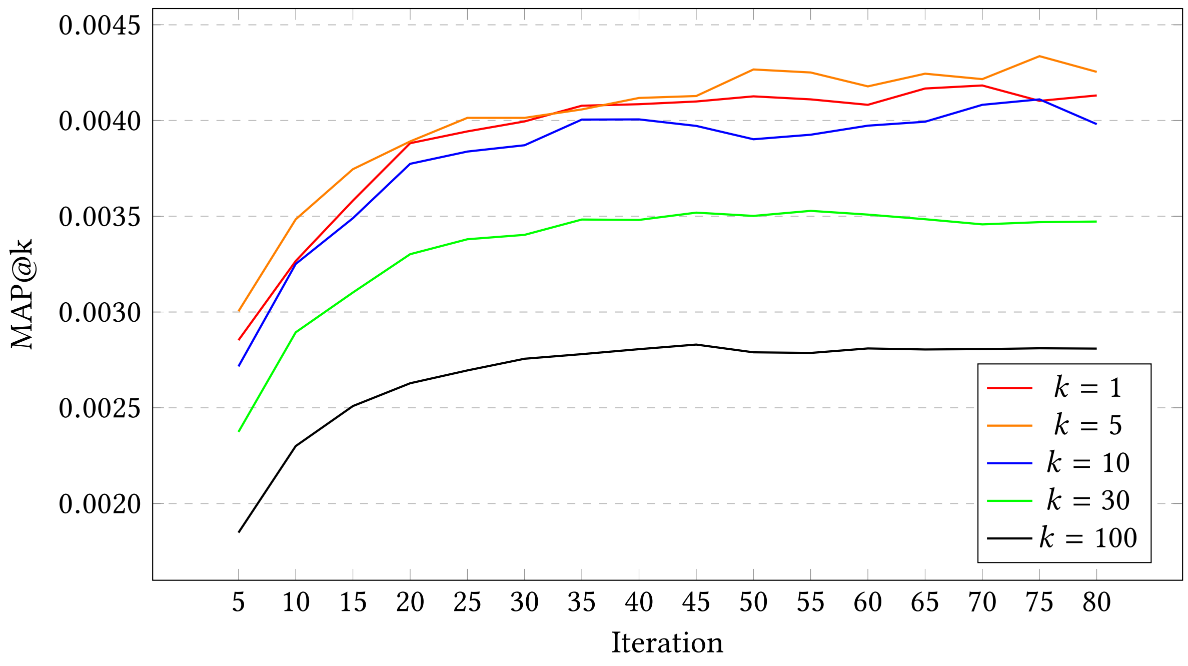

Figure 5.4.: CF Results for $k \in \{1, 5, 10, 30, 100\}$, 80 Iterations, $d = 100$ and $\alpha = 0.03$

### 5.3.2. Hybrid Collaborative-Content-based Filtering

In this section, I present the results for the hybrid CF-CBF approach. Based on grid search results for CF which also uses MF and due to considerably longer training and evaluation durations, I narrow the grid search. Thus, I omit $k \in \{5, 30\}$ and keep the learning rate set of the previous grid. The embedding dimension space set is extended according to results. Furthermore, I limit results on $S_{test}^{DL}$ which is sufficient to conclude on potential overfitting and to constrain intense computation times.[2]

Table 5.5 shows the respective results. We get the best result for $d = 700$ and $\alpha = 0.03$. Lower and higher embedding dimensionalities underfit and overfit the data leading to inferior performance. Results for lower and higher learning rates are consistently worse. Nevertheless, for $\alpha = 0.1$ and increasing $d$ there is no clear trend. As the optimization becomes more difficult by increased complexity from adding user and item features, the learning rate takes eventually too large steps to reach any local optima. This is reflected by the lack of a clear trend and supported by the circumstance that for $\alpha = 0.01$ results are consistently increasing, but slower as for the optimal $\alpha = 0.03$. The higher problem complexity is also reflected by $MAP@k$ for $\alpha = 0.03$ and different $d$. Despite the optimal $d = 100$ for only collaborative signal information (see 5.3.1), we now need a seven times larger dimensionality to additionally embed the content information. This is also reflected by comparing $d = 100$ to $d = 700$ for $\alpha = 0.03$. Allowing higher embedding space, the $MAP@10$ increased by about 26% from 0.005212 to 0.006580.

[2]Durations for $d = 500$ and $\alpha = 0.01$ running parallel on a 12-core machine: training 40 iterations takes 3,817s, a single test set evaluation takes 17,354s.

Table 5.5.: MAP@k on Test Set for Hybrid CF-CBF Hyperparameter Grid Search

| $k$ | $d$ | $\alpha$ | | |
|---|---|---|---|---|
| | | 0.1 | **0.03** | 0.01 |
| 10 | 50 | 0.000507 | 0.004431 | 0.003170 |
| | 100 | 0.000612 | 0.005212 | 0.003713 |
| | 200 | 0.000535 | 0.005897 | 0.004118 |
| | 300 | 0.000596 | 0.005985 | 0.004557 |
| | 400 | 0.000185 | 0.006208 | 0.004702 |
| | 500 | 0.000172 | 0.006398 | 0.004942 |
| | 600 | - | 0.006383 | - |
| | **700** | - | **0.006580** | - |
| | 800 | - | 0.006400 | - |
| | 900 | - | 0.006329 | - |
| | 1000 | - | 0.006197 | - |
| *based on 40 iterations* | | | | |

The training and evaluation time ranges from 7 to 11 hours on a 12 core machine for $d \in \{600, 700, 800, 900, 1000\}$. Combined with the fact that $MAP@10$ was more than one magnitude worse for lower and higher learning rates for $d \in \{50, 100, 200, 300, 400, 500\}$, I refrain from running the omitted cases for obvious reasons.

### 5.3.3. Deep Learning

The following part presents the $MAP@k$ performance for the proposed DL-based models $DL_{single}$, $DL_{multi-cos}$ and $DL_{multi-\overline{cos}}$. All models were trained with an initial learning rate $\alpha = 0.003$ and a similarity parameter $\lambda = 0.5$ (if applicable). I use MBGD with batch size 2048 which exceeds common batch sizes in the range of 16 or 32 up to 256 or 512. Increasing the batch size reduces the variance of the gradient. In order to reduce the fluctuation of the training accuracy curve, I choose this relatively large batch size as conventional batch sizes showed significantly variance. [GBC16, p. 275 sqq.] I trained each model for 5 epochs observing fast convergence to accuracies between 79% and 82%.[3] According to the sampling technique, binary train and test set labels are balanced. I use a test batch size of 10,000 examples and dropout regularization within the second hidden layer of the deep component with a dropout probability being 0.5 for regularization according to [SHKS14]. Figure 5.5 reports on the gold standard obtained for all models. For $k \in \{1, 5, 10, 30, 100\}$ $DL_{multi-cos}$ achieves the best results followed by $DL_{single}$. $DL_{multi-\overline{cos}}$ performs worst. For $k = 5$ we get $MAP@k_{multi-cos} = 0.00715$, $MAP@k_{single} = 0.00660$ and $MAP@k_{multi-\overline{cos}} = 0.00601$.

[3]Based on mathematically rounding $\hat{p}_{u,v}$ to either 0 or 1

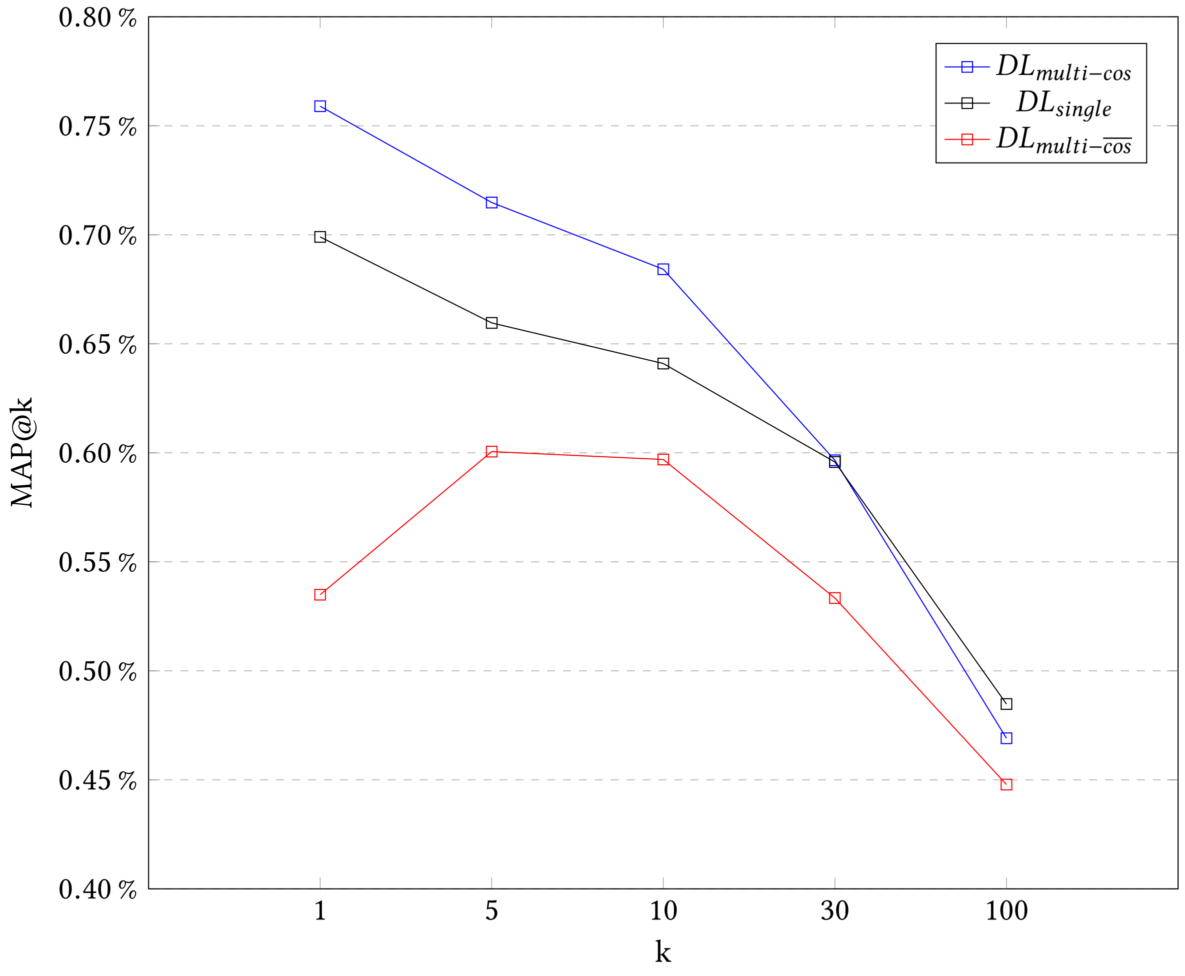

Figure 5.5.: MAP@k with respect to All Active Users in Test Set for $DL_{single}$, $DL_{multi-cos}$, and $DL_{multi-\overline{cos}}$

I verify that $DL_{multi-cos}$ is significantly better compared to $DL_{single}$ for $MAP@k$ with $k \in \{1, 5, 10\}$. This supports the assumption that joint learning of preference and similarity improves recommendation relevance. For the distinguished case of $MAP@5$ the improvement is 8.3%. Generally lower results for $DL_{multi-\overline{cos}}$ indicate that forcing similarities to be either 0 (opposite) or 1 (equal) distorts embeddings. Different vehicles a user interacted with are forced into a very similar embedding as opposed to the user embedding. This may reduce the variance in item embeddings and corresponding content captured by the deep component. This could finally cause the observed performance gap with respect to $DL_{multi-cos}$.

Network training for each DL model took approximately 8 hours plus 54 hours to infer the gold-standard. I expect the second duration to substantially decrease by implementing an efficient candidate generation approach as proposed. However, these vast times obstruct exhaustive grid search within the scope of my thesis. Therefore, I followed default settings and general learning advises for network hyperparameters and parameter initializations. The already outperforming results can therefore be regarded as lower boundaries, which can be leveraged through additional grid search improving *MAP@k* and candidate generation drastically decreasing inference times.

In addition, Figure 5.6 the development of the prediction histogram across iterations. It shows the increasing discrimination, that pushes negative signals (0) apart from positive signals (1) with the statistical masses concentrating around 0 and 1.

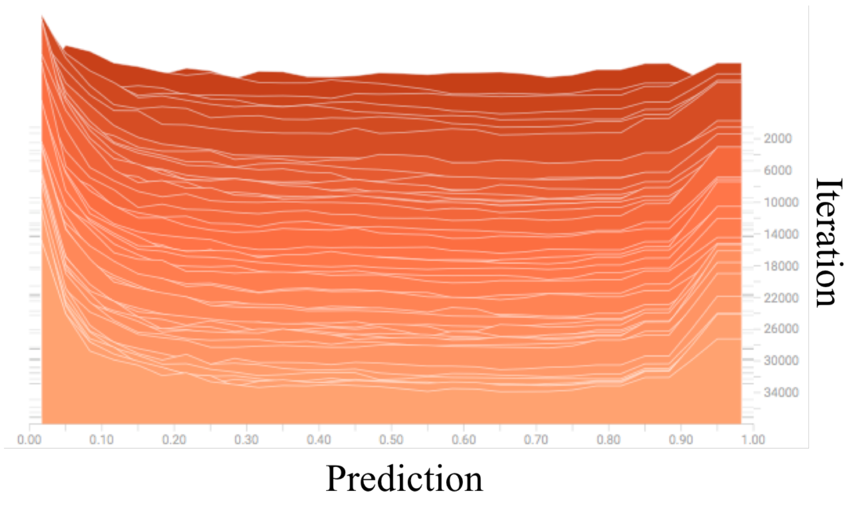

Figure 5.6.: TensorBoard Plot showing Discrimination between Disregard and Preference

## 5.4. Comparative Results across Approaches

In this final section, I compare the results across all approaches. Therefore, I take the best results for each individual approach as representative to compare them with each other. Figure 5.7 depicts those results on the test set for *MAP@k* across different $k$.

The aggregated results allow to conclude with two statements. Hybrid CF-CBF performs significantly better than CF with an increasing difference for growing $k$. It starts with 21% and rises to a 97% improvement. In addition, the best DL-based approach $DL_{multi-cos}$ is significantly better as CF (63-70%).

The third comparison, DL vs. hybrid CF-CBF, is not consistent across all $k$. Results for the DL-based approach are better for $k \in \{1, 5, 10\}$, whereas DL underperforms for $k \in \{30, 100\}$. Furthermore, hybrid CF-CBF becomes consistently better with increasing k compared to DL. However, there are two points mitigating this result. First, based on application and empirical justification for $k$, values higher than 10 play a subordinate role for the analysis, whereas small $k$ are more relevant for us. Second, application-driven we seek to optimize $k = 5$ and for this case DL clearly outperforms the hybrid CF-CBF approach by 19%.

As a result from these comparisons, DL significantly outperforms hybrid CF-CBF and CF for the relevant small $k$ in terms of relevance measured by *MAP@k*.

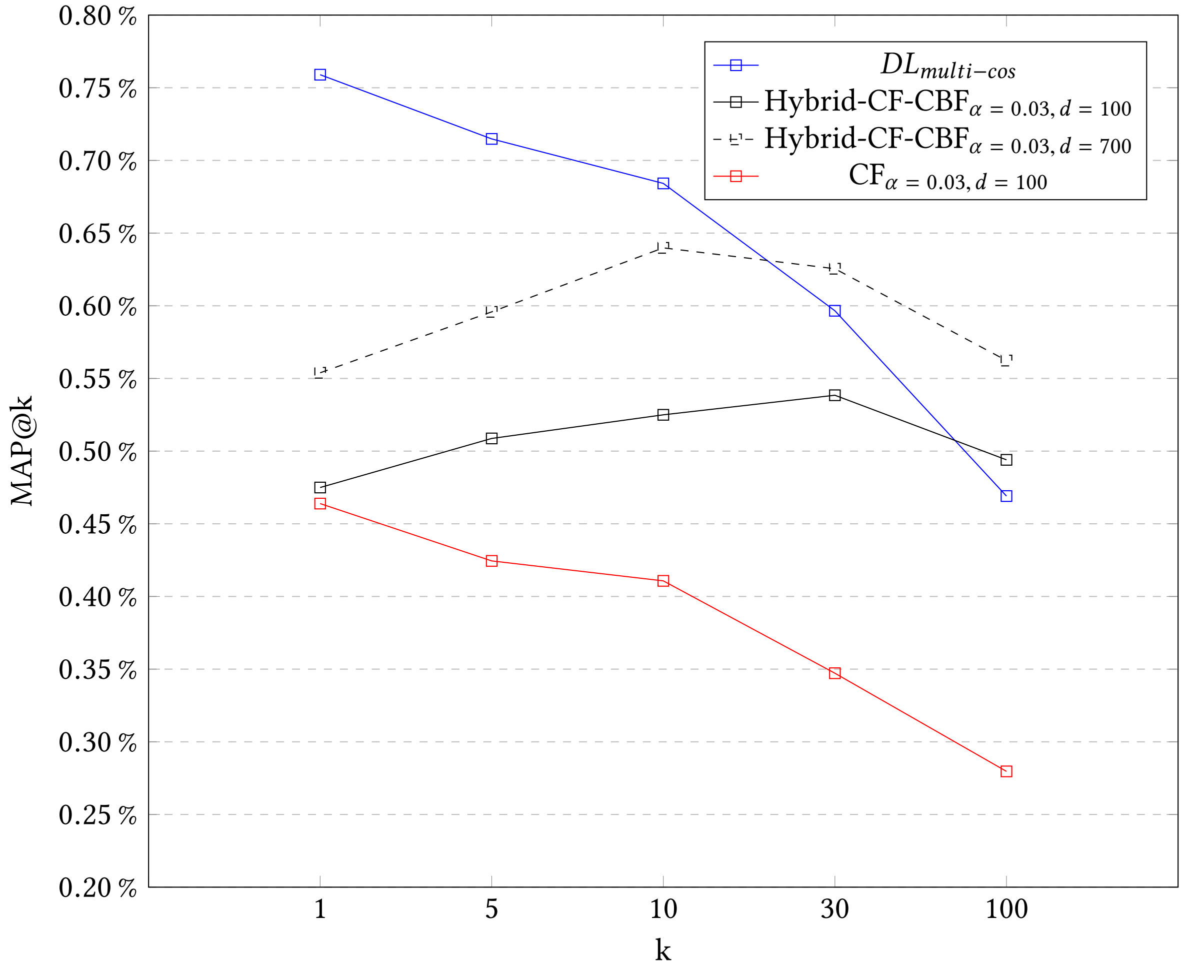

Figure 5.7.: MAP@k on Test Set for CF, Hybrid CF-CBF and $DL_{multi-cos}$

# 6. Conclusion

*"Our intuition about the future is linear. But the reality of information technology is exponential, and that makes a profound difference."*

Raymond Kurzweil,
The Singularity is Near (2005)

In my master thesis, I have presented the current state of art in DL for RSs and presented a wide and deep learning approach with a multi-objective optimization criterion and thorough embeddings for vehicle recommendations.

The current state of art shows diverse DL-based approaches towards RSs. We have seen RNNs, CNNs, AEs, standard DNNs and other types that leverage specific and partially multiple modalities of user/item representation, feedback and domains. DL-based approaches significantly outperform traditional IR-approaches towards RSs. Thus, DL is revolutionizing another field on its current expansion. Nevertheless, some problems still remain insufficiently addressed, like cold-starting users or items, multi-objective optimization criteria, or proper integration of different data modalities.

I addressed the problem of recommendation relevance, candidate generation and data integration by proposing a DL-based approach that extends the wide and deep learning approach by Google. I have shown that my DL-based approach significantly outperforms CF and Hybrid CF-CBF in terms of recommendation relevance measured by $MAP@k$. In order to achieve these results, I conducted three augmentations: changing from ReLU to ELU activation, adding user and item embeddings prior to the deep component, and introducing a joint training function. Changing the unit activation was a mere consequence of recent progress in DL research. The second adaption was motivated by linking candidate generation with ranking through a shared model. Thus, the proposed model not only generates dense low-dimensional embeddings to be used within an approximate nearest neighbor search based on LOPQ. But also as a third adaption, the model jointly trains on embedding similarity and on the distinction between preference and disregard for user/item-combinations. However, approaches towards scalable candidate generation were just proposed for brevity, I validated the superiority of my joint model approach for vehicle recommendations. In particular, the extension of the optimization criterion within $DL_{multi-cos}$ significantly outperformed $DL_{single}$. This additional superiority within the DL-based approach further increases its general supremacy and provides structured, dense representations as a by-product.

Therefore, I recall my second research question from section 1.2:
*How much does a deep learning based recommender outperform classical information retrieval approaches in terms of recommendation relevance?*
I draw the conclusion that my DL-based approach outperforms CF by 62-70% regarding *MAP@k* with respect to $k \in \{1, 5, 10, 30, 100\}$. Furthermore Hybrid CF-CBF was outperformed by 19% for the important case of $k = 5$. As a result, my DLRS significantly exceeds the relevance performance of classical IR approaches for vehicle recommendations.

# 7. Outlook

*"We can only see a short distance ahead, but we can see plenty there that needs to be done."*

Alan Turing, Computing machinery and intelligence (1950)

The additional value of DL for RSs becomes clearly visible throughout my thesis. On the one hand, there is the general development with many application domains in which DL contributes successfully. On the other hand, I can leverage DL to improve recommendations in the vehicle domain. Both settings, general and specific, provide directions for future work.

First, DL-based frameworks need to incorporate features from different modalities of structured and unstructured data from potentially different domains. In addition, there is plenty of research ongoing in the field of session-based RSs that take into account the more dynamic nature of users and the general aspect of time. More contributions in this area could advance existing contributions at this scientific frontline. Furthermore, as DL provides the flexibility, more objectives than relevance should be modeled and trained for in order to provide better recommendations for users. Last but not least, the current division into candidate generation and ranking seen in many approaches seems reasonable, but suboptimal as well. A major contribution would be to integrate these steps in order to globally optimize them instead of locally optimizing each component.

Second, I consciously limited parts of this work. Limitations need to be alleviated for further improvement: binary classification, negative sampling and candidate generation are just a few of them. Future work could take into account interaction frequency and interaction types to create finer-grained feedback signals to achieve a better resolution of preference and disregard. Furthermore, the negative sampling involved could be improved or adapted. E.g. using a regression-based approach would make it obsolete, which could be worth a try. In order to prove the expected scalability of my approach, the most reasonable next step would include to implement and evaluate the proposed candidate generation strategies. Afterwards, A/B-tests could provide information on online improvements as offline and online metrics do not necessarily correlate. Finally, also general projections presented in the beginning apply to this specific domain.

# A. Appendix

## A.1. Example: Matrix Factorization for Collaborative Filtering

This section show a brief example for MF as a model-based approach for CF. We use explicit feedback on a rating scale from 1 to 5. Given user $U = \{1, 2, 3, 4, 5, 6\}$ and items $V = \{A, B, C, D, E, F\}$ we use the following example rating matrix:

$$R = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{pmatrix} & 2 & & 1 & 3 & \\ & & & 4 & & 4 \\ & 3 & & 3 & & \\ & & & 3 & & 3 \\ & & 1 & 2 & & \\ & & & 4 & & 2 \end{pmatrix} \end{array}$$

Choosing $d = 2$ we initialize the elements of $P^{6\times2}$ and $Q^{6\times2}$ from $\mathcal{N}(0, 1)$. For gradient descent we choose a learning rate $\alpha = 0.01$. Furthermore, we only use the unregularized squared error and SGD for the variable update. Training over 100 epochs leads to the following loss decrease:

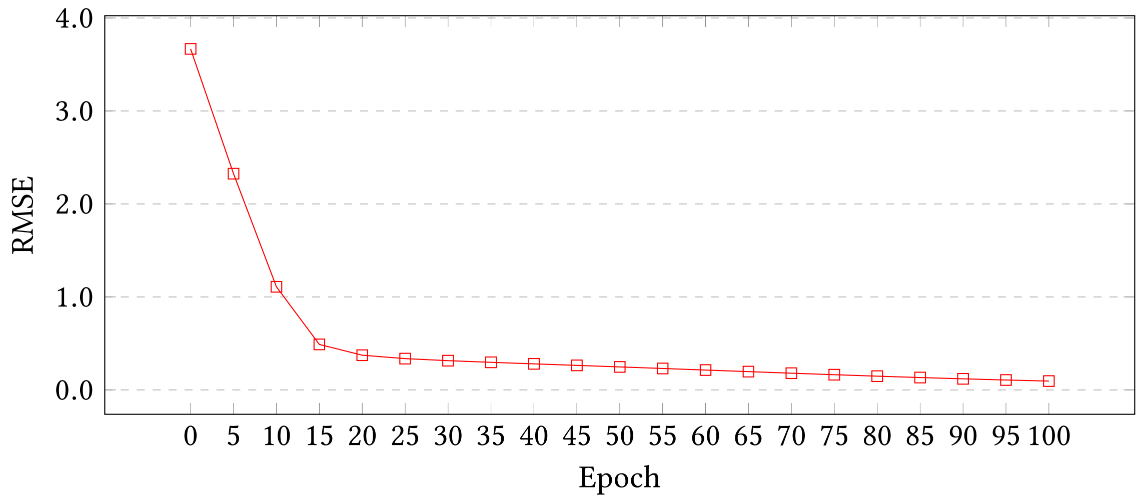

Figure A.1.: RMSE across 100 Epochs for Matrix Factorization Example

We get the following results for user and item embeddings which multiplied yield the reconstructed rating matrix $\hat{R}$ (values rounded to one decimal):

$$\hat{R} = P \times Q^T = \begin{pmatrix} -0.7 & -1.0 \\ 0.9 & -1.5 \\ 0.6 & -1.2 \\ 0.7 & -1.2 \\ -1.4 & -2.1 \\ 1.9 & -0.7 \end{pmatrix} \times \begin{pmatrix} 0.8 & 0.8 \\ 0.3 & -2.3 \\ -0.3 & -0.3 \\ 1.4 & -1.9 \\ -2.1 & -1.4 \\ 0.3 & -2.4 \end{pmatrix}^T \approx \begin{pmatrix} -1.4 & 2.1 & 0.5 & 0.9 & 3.0 & 2.2 \\ -0.5 & 3.8 & 0.2 & 4.1 & 0.3 & 3.9 \\ -0.6 & 3.0 & 0.2 & 3.0 & 0.6 & 3.1 \\ -0.4 & 2.8 & 0.1 & 3.1 & 0.3 & 2.9 \\ -2.9 & 4.4 & 1.0 & 2.0 & 5.9 & 4.6 \\ 0.9 & 2.0 & -0.4 & 3.8 & -2.9 & 2.1 \end{pmatrix}$$

If we exclude known positives, i.e. all items a user already interacted with, we can use the remaining rating predictions to select item recommendations from them. Bold digits in the matrix below indicate the result of this selection with $k = 1$:

$$\begin{array}{c|cccccc} & A & B & C & D & E & F \\ \hline 1 & -1.4 & & 0.5 & & & \mathbf{2.2} \\ 2 & -0.5 & \mathbf{3.8} & 0.2 & & 0.3 & \\ 3 & -0.6 & & 0.2 & & 0.6 & \mathbf{3.1} \\ 4 & -0.4 & \mathbf{2.8} & 0.1 & & 0.3 & \\ 5 & -2.9 & 4.4 & & & \mathbf{5.9} & 4.6 \\ 6 & 0.9 & \mathbf{2.0} & -0.4 & & -2.9 & \end{array}$$

As a result, item $F$ would be recommended to users 1 and 3, item $B$ to users 2, 4 and 6, and item $E$ to user 5. These items are predicted to be the top choices across all unseen, but available items for each user. We can see some patterns in the ratings and reason the recommendations. Users 2, 4, and 6 rated items $D$ and $F$. $D$ was also rated by all other users among which two out of three also rated item $B$. As user 2 rated $D$ with a four, user 3 is more alike in terms of rating $D$ as user 1. This might cause user 2's slight preference towards rating item $B$ with a three instead with a two. These and other underlying relations may be captured by the item and user embeddings and lead to estimates that approximate well the underlying user preferences.

## A.2. Markov Chain based User Interaction Weighting

This section describes an approach to obtain weights for different implicit feedback types. Based on a Markov chain analysis, we can derive weights from a stationary distribution that describes the long-term behavior of a Markov process. We can use these weights to put different importance on item features as part of user interactions for the user profile learner. Weights can also serve to aggregate user profiles that are separated by interaction type into combined profiles.The proposed approach will be applied to the vehicle recommendation domain with three implicit feedback types. Theory on Markov processes used in this section borrows from [WH16, p. 220 sqq.].

First, we make the assumption that the type of an interaction only depends on its previous interaction type. Thus, we fulfill the Markov property requiring that future development of a stochastic process just depends on its most recent state. Thus, it is independent from all other previous states with $I$ denoting the finite state space, $X_t$ being a random variable for taking a state $i \in I$ and $t \in \mathbb{N}_0$ standing for the discrete time step in a sequence of states:

$$P(X_{t+1} = i_{t+1} | X_t = i_t) = P(X_{t+1} = i_{t+1} | X_t = i_t, X_{t-1} = i_{t-1}, ..., X_0 = i_0), i \in I, t \in \mathbb{N}_0$$

The probability for assuming state $i_{t+1}$ based on the previous state $i_t$ is called *transition probability* and will be denoted as $p_{i,j}, i, j \in I$. $P = \{p_{i,j}\}$ is a stochastic matrix that contains all transition probabilities and is called *transition matrix*. The development of a Markov chain over time can be fully described by its transition matrix and initial probabilities $\pi_i(0) := P(X_0) = i, i \in I$. Analyzing the asymptotic behavior of the Markov chain yields the stationary distribution $\pi$ as the long-term probabilities for the random variable taking a state $i \in I$.

We now apply this theory to vehicle recommendations with four interaction types constituting the state space $I = \{V, B, P, C\}$: (V)iew, (B)ookmark, (P)rint and (C)ontact. It goes without saying that the underlying preference denoted as $\psi : I \longrightarrow \mathbb{R}$ shows the following relation: $\psi(V) < \psi(B) < \psi(P) < \psi(C)$. Our task is to estimate $\psi(i), \forall i \in I$ given user interaction histories.

For a given analysis period of time, we take each user $u \in U$ and the set of items $V_u$ he interacted with. We separate the interactions by items getting $|V_u|$ subsets $S_{u,v}, v \in V_u$. Each subset can be ordered temporally imposing a user-item-specific sequence of interactions. The following example describes such a sequence that allows to determine an empirical transition matrix $t_{u,v}$:

$$S_{u,v}^{ordered} = (V, B, C, V, V, P, V, V, C, V, V)$$

$$t_{u,v} = \begin{array}{c|cccc} & V & B & P & C \\ \hline V & 0.5 & 0.1667 & .01667 & .01667 \\ B & 0 & 0 & 0 & 1.0 \\ P & 1.0 & 0 & 0 & 0 \\ C & 1.0 & 0 & 0 & 0 \end{array}$$

We can determine these matrices for three different levels:

- User-Item transitions
- User transitions
- Domain transitions

User-Item transitions $t_{u,v}$ refer to specific user-item combinations. User transitions $t_u$ take into account interaction sequences with all $v \in V_u$. Finally $T_{U,V}$ does this $(u, v) \in U \times V, |s_{(u,v)}| > 1$. The following matrix shows an example:

$$T_{U,V} = \begin{array}{c|cccc} & V & B & P & C \\ \hline V & 0.60 & 0.20 & 0.15 & 0.05 \\ B & 0.50 & 0.30 & 0.10 & 0.10 \\ P & 0.60 & 0.10 & 0.10 & 0.20 \\ C & 0.75 & 0.05 & 0.05 & 0.15 \end{array}$$

The transition matrices on all three levels can be used to determine a stationary distribution $\pi$. For the matrix above we get $\pi = (0.594, 0.194, 0.125, 0.087)$. Afterwards, we use the inverse of each state probability to calculate the weight of the feedback type:

$$\psi(i) := \frac{1}{\pi(i)}, \quad i \in I \tag{A.1}$$

Finally, we can use these weights for the profile learner to weigh numerical item features of related interactions or to aggregate user profiles for different interaction types into a combines user profile. $F_U$ and $F_V$ denote the user and item feature sets:

$$\begin{aligned} f &\in F \subseteq (F_U \cap F_V), \\ f_u &= \frac{1}{\lambda_u} \sum_{v \in V_u} \sum_{k=1}^{|S_{u,v}|} \psi(S_{u,v}^k) \cdot f_v, \\ \lambda_u &= \sum_{v \in V_u} \sum_{k=1}^{|S_{u,v}|} \psi(S_{u,v}^k). \end{aligned} \tag{A.2}$$

## A.3. Wide and Deep Embedding Binary Classifier Model

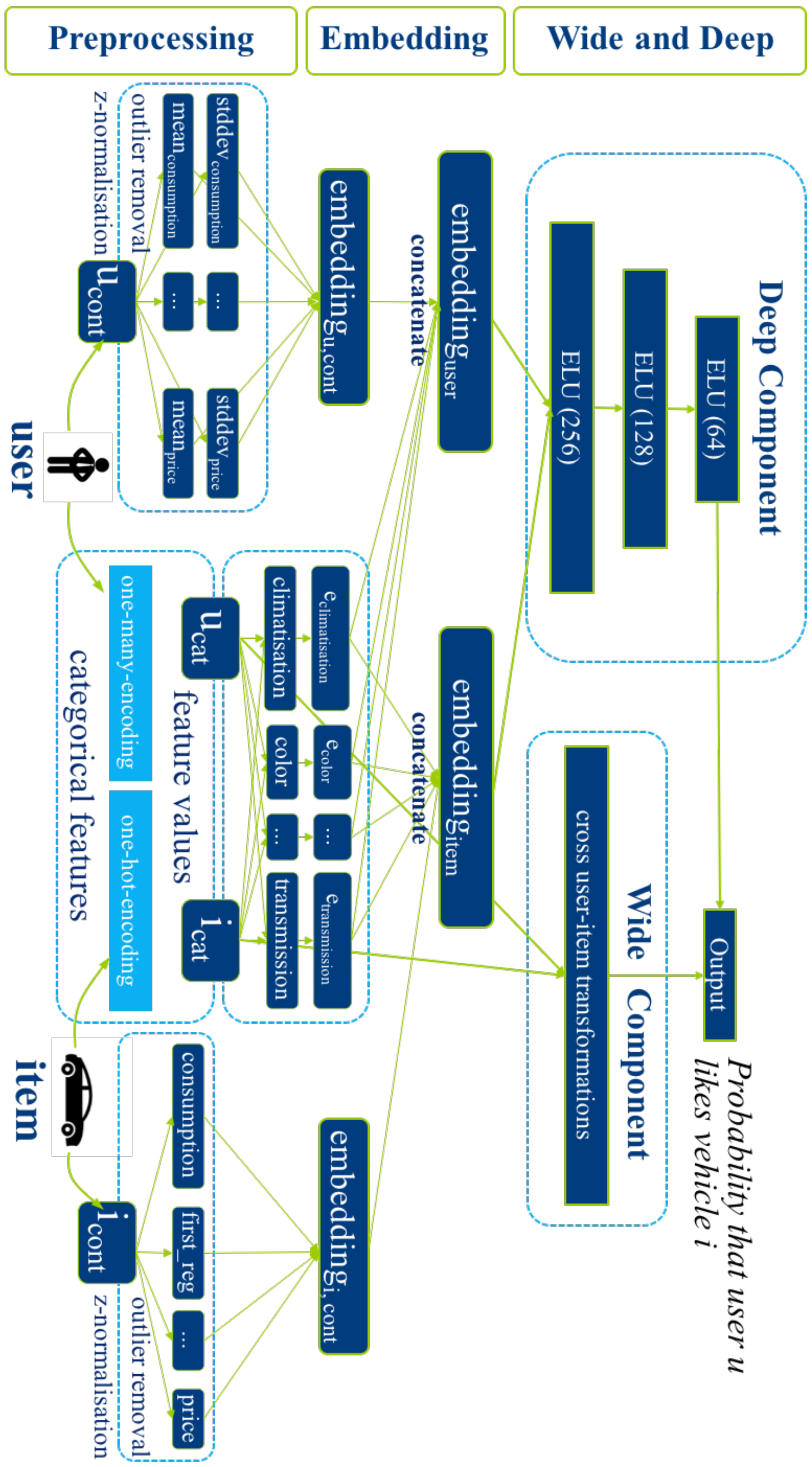

Figure A.2.: Wide and Deep Learning Classifier with User and Item Embeddings

# Bibliography

[AABB16] Martín Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[AB12] Xavier Amatriain and Justin Basilico. *Netflix Recommendations: Beyond the 5 stars (Part 1)*. 2012. URL: `https://medium.com/netflix-techblog/netflix-recommendations-beyond-the-5-stars-part-1-55838468f429` (visited on 09/24/2017).

[Abe16] Christopher R. Aberger. "Recommender: An Analysis of Collaborative Filtering Techniques". 2016.

[ADEK17] Fabian Abel et al. "RecSys Challenge 2017". In: *Proceedings of the 11th ACM Conference on Recommender Systems*. ACM, 2017, pp. 372–373.

[Agg16] Charu C. Aggarwal. *Recommender Systems: The Textbook*. Cham: Springer International Publishing, 2016.

[Asa09] Matt Asay. *Shirky: Problem is filter failure, not info overload*. 2009. URL: `https://www.cnet.com/news/shirky-problem-is-filter-failure-not-info-overload/` (visited on 09/24/2017).

[BBM16] Trapit Bansal, David Belanger, and Andrew McCallum. "Ask the GRU: Multi-task Learning for Deep Text Recommendations". In: *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, 2016, pp. 107–114.

[BEWL11] Thierry Bertin-Mahieux et al. "The Million Song Dataset". In: *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR)* (2011).

[CAS16] Paul Covington, Jay Adams, and Emre Sargin. "Deep Neural Networks for YouTube Recommendations". In: *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, 2016, pp. 191–198.

[Che16] Heng-Tze Cheng. *Wide & Deep Learning: Better Together with TensorFlow*. 2016. URL: `https://research.googleblog.com/2016/06/wide-deep-learning-better-together-with.html` (visited on 09/24/2017).

[CKHS16] Heng-Tze Cheng et al. "Wide & Deep Learning for Recommender Systems". In: *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*. ACM, 2016, pp. 7–10.

[CUH16] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". In: *arXiv preprint arXiv:1511.07289* (2016), pp. 1–14.

[Dha13] Vasant Dhar. “Data Science and Prediction”. In: *Communications of the ACM* 56.12 (2013), pp. 64–73.

[DWTS17] Hanjun Dai et al. “Deep Coevolutionary Network: Embedding User and Item Features for Recommendation”. In: *arXiv preprint arXiv:1609.03675* (2017).

[DYP17] Kaz Dato, Cliff Young, and David Patterson. *An in-depth look at Google’s first Tensor Processing Unit (TPU).* 2017. URL: `https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu` (visited on 09/24/2017).

[EKNK17] Andre Esteva et al. “Dermatologist-level classification of skin cancer with deep neural networks”. In: *Nature* 542.7639 (2017), pp. 115–118.

[ERK10] Michael D. Ekstrand, John T. Riedl, and Joseph A. Konstan. “Collaborative Filtering Recommender Systems”. In: *Foundations and Trends in Human–Computer Interaction* 4.2 (2010), pp. 81–173.

[ESH15] Ali Elkahky, Yang Song, and Xiaodong He. “A Multi-View Deep Learning Approach for Cross Domain User Modeling in Recommendation Systems”. In: *Proceedings of the 24th International Conference on World Wide Web* (2015), pp. 278–288.

[Faw06] Tom Fawcett. “An introduction to ROC analysis”. In: *Pattern Recognition Letters* 27.8 (2006), pp. 861–874.

[FPS96] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. “From Data Mining to Knowledge Discovery in Databases”. In: *AI Magazine* 17.3 (1996), pp. 37–54.

[GB10] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics.* 2010, pp. 249–256.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. 1st ed. MIT Press, 2016.

[Gér17] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn & TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* 1st ed. Sebastopol: O’Reilly Media, Inc., 2017.

[Goo17] Google. *TensorFlow Wide and Deep Learning Tutorial.* 2017. URL: `https://www.tensorflow.org/tutorials/wide%7B%5C_%7Dand%7B%5C_%7Ddeep` (visited on 09/26/2017).

[GRDB15] Mihajlo Grbovic et al. “E-commerce in Your Inbox: Product Recommendations at Scale”. In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 1809–1818.

[Gro17] Roger Grosse. *Introduction to Neural Networks and Machine Learning, Lecture 15: Exploding and Vanishing Gradients.* 2017. URL: `http://www.cs.toronto.edu/%7B~%7Drgrosse/courses/csc321%7B%5C_%7D2017/readings/L15%20Exploding%20and%20Vanishing%20Gradients.pdf` (visited on 09/24/2017).

[GTYL17] Huifeng Guo et al. “DeepFM: A Factorization-Machine based Neural Network for CTR Prediction”. In: *arXiv preprint arXiv:1703.04247* (2017).

[Gud16] Alex Gude. *9 Must-Have Datasets for Investigating Recommender Systems.* 2016. URL: `http://www.kdnuggets.com/2016/02/nine-datasets-investigating-recommender-systems.html` (visited on 09/12/2017).

[Hea15] Jeff Heaton. *Artificial Intelligence for Humans. Volume 3: Deep Learning and Neural Networks.* Vol. 3. 2015.

[HK15] F. Maxwell Harper and Joseph A. Konstan. “The MovieLens Datasets: History and Context”. In: *ACM Transactions on Interactive Intelligent Systems* 5.4 (2015), pp. 1–19.

[HKBR99] Jonathan L. Herlocker et al. “An algorithmic framework for performing collaborative filtering”. In: *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval.* ACM, 1999, pp. 230–237.

[HKBT15] Balázs Hidasi et al. “Session-based Recommendations with Recurrent Neural Networks”. In: *arXiv preprint arXiv:1511.06939* (2015). URL: `http://arxiv.org/abs/1511.06939`.

[HKV08] Yifan Hu, Yehuda Koren, and Chris Volinsky. “Collaborative Filtering for Implicit Feedback”. In: *IEEE International Conference on Data Mining* (2008), pp. 263–272.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome; Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* 2nd ed. Springer Series in Statistics. New York, NY: Springer-Verlag New York, 2009.

[HZRS15] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE International Conference on Computer Vision.* 2015, pp. 1026–1034.

[IS15] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *International Conference on Machine Learning.* 2015, pp. 448–456.

[KA14] Yannis Kalantidis and Yannis Avrithis. “Locally optimized product quantization for approximate nearest neighbor search”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* IEEE, June 2014, pp. 2321–2328.

[KAPM08] Yehuda Koren et al. “Factorization meets the neighborhood: a multifaceted collaborative filtering model”. In: *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 2008, pp. 426–434.

[KB14] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[KE17] Joseph A. Konstan and Michael D. Ekstrand. *Intoduction to Recommender Systems: Non-Personalized and Content Based*. 2017. URL: `https://www.coursera.org/learn/recommender-systems-introduction` (visited on 09/24/2017).

[Kul15] Maciej Kula. "Metadata Embeddings for User and Item Cold-start Recommendations". In: *arXiv preprint arXiv:1507.08439* (2015).

[LBOM12] Yann A. LeCun et al. "Efficient backprop". In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[Lev15] Daniel J Levitin. *Why the modern world is bad for your brain*. 2015. URL: `https://www.theguardian.com/science/2015/jan/18/modern-world-bad-for-brain-daniel-j-levitin-organized-mind-information-overload` (visited on 09/24/2017).

[LPLH16] Babak Loni et al. "Bayesian Personalized Ranking with Multi-Channel User Feedback". In: *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, 2016, pp. 361–364.

[LRJ15] Jiwei Li, Alan Ritter, and Dan Jurafsky. "Learning multi-faceted representations of individuals from heterogeneous evidence using neural networks". In: *arXiv preprint arXiv:1510.05198* (2015).

[LZE15] Dawen Liang, Minshu Zhan, and Daniel P.W. Ellis. "Content-Aware Collaborative Music Recommendation Using Pre-trained Neural Networks". In: *ISMIR 2015: Proceedings of the 16th International Society for Music Information Retrieval Conference*. 2015, pp. 295–301.

[Man12] JP Mangalindan. *Amazon's recommendation secret*. 2012. URL: `http://fortune.com/2012/07/30/amazons-recommendation-secret/` (visited on 09/24/2017).

[MC07] Nikos Manouselis and Constantina Costopoulou. "Analysis and classification of multi-criteria recommender systems". In: *World Wide Web* 10.4 (2007), pp. 415–441.

[McA16] Nathan McAlone. *Why Netflix thinks ist personalized recommendation engine is woth $1 billion per year*. 2016. URL: `http://www.businessinsider.de/netflix-recommendation-engine-worth-1-billion-per-year-2016-6?r=US%7B%5C&%7DIR=T` (visited on 09/24/2017).

[Mit97] Tom M. Mitchell. *Machine Learning*. 1st ed. McGraw-Hill Education, 1997.

[NNFM17] Andrew Ng et al. *Deep Learning Tutorial*. 2017. URL: `http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/` (visited on 09/24/2017).

[ODS13] Aaron van den Oord, Sander Dieleman, and Benjamin Schrauwen. "Deep content-based music recommendation". In: *Advances in neural information processing systems*. 2013, pp. 2643–2651.

[ODZS16] Aaron van den Oord et al. "WaveNet: A Generative Model for Raw Audio". In: *arXiv preprint arXiv:1609.03499* (2016).

[Pet17] Georgios Petropoulos. *Machines that learn to do, and do to learn: What is artificial intelligence?* 2017. URL: `http://bruegel.org/2017/04/machines-that-learn-to-do-and-do-to-learn-what-is-artificial-intelligence/` (visited on 09/24/2017).

[PVGM11] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in {P}ython". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[QKHC17] Massimo Quadrana et al. "Personalizing Session-based Recommendations with Hierarchical Recurrent Neural Networks". In: *Proceedings of the 11th ACM Conference on Recommender Systems.* ACM, 2017.

[RFGS09] Steffen Rendle et al. "BPR: Bayesian Personalized Ranking from Implicit Feedback". In: *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence.* AUAI Press, 2009, pp. 452–461.

[RGR17] David Reinsel, John Gantz, and John Rydning. *Data Age 2025: The Evolution of Data to Life-Critical Don't Focus on Big Data; Focus on the Data That's Big.* Tech. rep. April. Framingham, MA, USA: International Data Corporation (IDC), 2017.

[RN09] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* 3rd. Upper Saddle River: Pearson Education Inc., 2009.

[RRS15] Francesco Ricci, Lior Rokach, and Bracha Shapira. *Recommender Systems Handbook.* 2nd ed. New York: Springer Science+Business Media New York, 2015.

[RV97] Paul Resnick and Hal R. Varian. "Recommender Systems". In: *Communications of the ACM* 40.3 (1997), pp. 56–58.

[SE15] Neil Slater and Mohamed Ezz. *What is the "dying ReLU" problem in neural networks?* 2015. URL: `https://datascience.stackexchange.com/questions/5706/what-is-the-dying-relu-problem-in-neural-networks` (visited on 09/24/2017).

[SHKS14] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[SMG16] Florian Strub, Jérémie Mary, and Romaric Gaudel. "Hybrid Recommender System based on Autoencoders". In: *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems.* ACM, 2016, pp. 11–16.

[SMH07] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. "Restricted Boltzmann machines for collaborative filtering". In: *Proceedings of the 24th international conference on Machine learning.* ACM, 2007, pp. 791–798.

[SVV99] Cheri Speier, Joseph S. Valacich, and Iris Vessey. "The Influence of Task Interruption on Individual Decision Making: An Information Overload Perspective". In: *Decision Sciences* 30.2 (1999), pp. 337–360.

[TSHL17] Mikhail Trofimov et al. "Representation Learning and Pairwise Ranking for Implicit and Explicit Feedback in Recommendation Systems". In: *arXiv preprint arXiv:1705.00105* (2017).

[VKW17] Rianne Van Den Berg, Thomas N. Kipf, and Max Welling. "Graph Convolutional Matrix Completion". In: *arXiv preprint arXiv:1706.02263* (2017).

[WB13] Jonathan S. Ward and Adam Barker. "Undefined By Data: A Survey of Big Data Definitions". In: *arXiv preprint arXiv:1309.5821* (2013).

[WBU11] Jason Weston, Samy Bengio, and Nicolas Usunier. "WSABIE: Scaling up to large vocabulary image annotation". In: *IJCAI International Joint Conference on Artificial Intelligence.* 2011, pp. 2764–2770.

[WH16] Karl-Heinz Waldmann and Werner E. Helm. *Simulation stochastischer Systeme.* 1st ed. Berlin: Springer-Verlag Berlin Heidelberg, 2016.

[WHCZ17] Jian Wei et al. "Collaborative filtering and deep learning based recommendation system for cold start items". In: *Expert Systems with Applications* 69 (2017), pp. 29–39.

[WHLM16] Shuaiqiang Wang et al. "Ranking-Oriented Collaborative Filtering: A Listwise Approach". In: *ACM Transactions on Information Systems* 35.2 (2016), 10:1–10:28.

[WWY15] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. "Collaborative Deep Learning for Recommender Systems". In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM, 2015, pp. 1235–1244.

[ZDW16] Weinan Zhang, Tianming Du, and Jun Wang. "Deep Learning over Multi-field Categorical Data: A Case Study on User Response Prediction". In: *European conference on information retrieval.* Springer, 2016, pp. 45–57.

[Zhe16] Lei Zheng. *A Survey and Critique of Deep Learning on Recommender Systems.* Tech. rep. September. Chicago: Department of Computer Science, University of Illinois at Chicago, 2016.

[ZL16] Barret Zoph and Quoc V. Le. "Neural Architecture Search with Reinforcement Learning". In: *arXiv preprint arXiv:1611.01578* (2016).

[ZNY17] Lei Zheng, Vahid Noroozi, and Philip S. Yu. "Joint Deep Modeling of Users and Items Using Reviews for Recommendation". In: *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining.* ACM, 2017, pp. 425–434.

---

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

**Karlsruhe, September 27th 2017**

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

(Marcel Kurovski)