Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 265

Systems Group, Department of Computer Science, ETH Zurich

in collaboration with

Digitec Galaxus AG

Improving Session-Based Recommendation Systems with Item Embeddings

by

Mohammed Ajil

Supervised by

Bojan Karlas, Ce Zhang

September 15, 2019

inf | Informatik
Computer Science

Abstract

This work examines how a session-based recommendation system can be combined with pre-trained item embeddings. Session-based recommendation systems are highly relevant in online services such as music or video streaming services and e-commerce, more and more so when the item catalogs grow into the millions. The approach presented in this work is tested in a production setting with live users, using a industry dataset magnitudes larger than the datasets used in works this thesis is based on. We show that session-recommendation performs better than classical recommendation approaches, however the combination of the two approaches was unsuccessful, failing primarily because of the Long Tail phenomenon.

Contents

Chapter 1

Introduction

Given the fact that targeted addressing of customers is becoming increasingly important, recommendation systems gain equally in relevance. Recommendation systems are a field in machine learning that attracts increasing attention. The most general definition of a recommendation system is a system that aids a user in some kind of decision process. Usually a tremendously large number of choices is available in such a decision process, such as which music to listen to from a selection of millions of tracks, or which products to buy from a catalog of millions of products. Depending on the recommendation system, information about the items, about the users, and about the interaction of users with items is taken into account.

The first approaches to recommendation engines mainly used interaction data, since the tools available at that time did not allow for an incorporation of image, text, or side-information. In recent years, deep neural networks (DNNs) enabled the use of more types of datapoints, such as images and text. Recurrent neural networks (RNNs, c.f. 2.2.1) allowed to explicitly model sequence data, which allowed a multitude of new possibilities in this field, since users tend to change their behavior over time. Specifically GRU4Rec (c.f. [6]) gained attention as an RNN based approach to model session data. In this setting, we try to model the sequence of events a user makes when he browses some catalog, instead of just static information about items and users. Another advantage would be that in a session-based approach the identification of the user is not necessary, because identifying users is a difficult task in itself. Since the session serves as a basis for recommendations, the user can be taken into consideration to additionally improve the results, but is not necessary for a recommendation. This is in contrast to other methods such as collaborative filtering where the user must be known. In [11] a model is proposed that does exactly that, it uses GRU4Rec as a basis and extends it with a user-level representation.

As mentioned before, DNNs have allowed the ingestion of other types of

datapoints. This allows the vectorization of structured data, as for example representing items in a euclidian vector space. This type of system provides the advantage of quantifying the abstract concept of similarity. As a consequence, large amounts of items can be better understood and navigated The authors of [15] introduced a model that can performing this function. By injesting session based data about the items as well as categorical side information, the model produces a fixed sized vector in an euclidian space for each item, while similar items will be represented by similar vectors.

The arim of this thesis is to explore the possibility of combining the two concepts in order to enable a session-based recommendation system to make use of the quantifiable similarity between items.

Chapter 2

Background

2.1 Recommendation Systems

2.1.1 Problem Statement

Generally speaking, recommendation systems aim to support users in their decision making, while interacting with a large number of possible choices. They recommend items to users based on their historical preferences. Virtually, any decision-process can be made easier for users by providing recommendations. Recommendation systems typically are used by online services, such as by Spotify for music [2] and YouTube for videos [4]. By using these online services, users generate data that the service can use to improve the recommendations. To illustrate this point, rating videos indicates to the operator of the service how well the video is received by this specific user. However, also retailers are known to use recommendation systems. Based on data generated by customer loyalty programs such as Migros Cumulus [14], retailers customize coupons or other offers to their customers. In principle, it can be stated that data of users interacting with items (viewing, buying, rating etc.) constitute the basis for a recommendation system. Depending on the use-case, this data is utilized to assign scores to items depending on the user. The semantic meaning of a score depends on the objective of a recommendation system: A system which recommends products to be bought might assign a "probability of purchase", whereas a system which recommends videos might predict the probability of a user watching a video to the end.
Bringing this all together we can define a general recommendation system with the following function:

$$s = f(i, u, h_u) \tag{2.1}$$

Where s is the assigned score to item i for the user u and the user's history u_h. The user's history represents all the previous interactions with items.

The aim of a recommendation engine is to determine the function f. In order to assess the quality of the learned function, we need to establish the "true" score for the specific item and user, for which there are different ways. During training, the learned functions are being tested on later interactions with items which are being held back. Given the fact that as soon as the user sees the recommendation, his reality is changed, the performance of a recommendation engine in production also has to be assessed, i.e. the provider cannot be sure if or which recommendation in the user history led the user to the action.

2.1.2 Properties of Recommendation Systems

The following chapter gives a brief overview of different recommendation system properties. Typically, recommendation systems consist of a combination of the following properties:

User-based User-based recommendation systems base their recommendations mainly on the properties of the user, such as gender or age, as well as the user's historical information, i.e. what he has bought or read before. Thus, when generating recommendations for a specific user, the system tries to find similar users and recommends items those users liked.

Item-based Item-based recommendation systems use mostly information about the item to produce recommendations, such as item type (e.g. genre of a movie). However, the basis for a recommendation is always a specific item, for example a product a user is looking at online. Thw item the user is currently interacting with is referred to as the "active item". Usually, item-based recommendation systems then try to find similar items to the currently active item. The method of finding similar items can be arbitrarily complex, which is why identifying similar items is its own field of research. A common method is to combine this approach with a user-based component, where the similar items are sorted or filtered based on the user. An example of this might be the following:

- Steve is a male, looking at black shirts.
- When extracting similar products, we find a range of black shirts, including some women's shirts.
- If we would recommend items by popularity, the women's shirts would appear as the first recommendations, since women tend to buy more shirts.
- Instead of directly displaying the recommendations, we filter out the women's shirts that are part of this selection.

Session-based Session-based recommendation systems are a rather new form of such a system. By definition, session is a sequence of interactions from a user with one or several items. Depending on the use-case and setting, this can be defined differently. Usually, online-services define a session as the sequence of interactions a user produces on the site until closing the browser window. Based on this fact, session differ in length which is why it is difficult to directly feed that information into a recommendation system. However, with the rise of recurrent neural networks (c.f. 2.2.1) a powerful tool for handling variable length sequence data becomes available. Session-based recommendation systems use RNNs to model the sequence data generated by sessions to achieve two things: First, by using RNNs, a fixed dimensional representation of a specific session can be extracted, which allows the comparison of different sessions. Second, by using the fixed representation of a session, the user's intent in a specific session can be identified and based on this, recommendations can be given in order to fulfill the user's intent. An intent can be defined as the goal the user has in a specific session. In the example of an online shop there are a few different, well-known intents identified by analysts such as: browsing for inspiration, searching for a specific product, buying a specific product, researching products. Also these intents exist in different contexts, which in case of an online shop can be different product types (mobile phone, couch, dining table etc.). The above mentioned explanation shows why these systems are more desired by operators of online services, as the recommendations can be targeted much more specifically to the user and his intent, instead of using just general information of the user and the active item.

Collaborative Collaborative recommendation systems mostly use interaction data to generate recommendations. Clicks on products are for example used as a data source in order to predict which products the user will click on next. However, the collaborative aspect comes from the fact that we source other users' interactions as a basis for the recommendations. In principle we interpret different users as versions of possible behavior of a user, the more interactions two users have in common, the more similar they are assumed to be. Therefore, we can extend the behavior (i.e. product views) of a user by looking at what similar users have done on the same active item.

Content-based In contrast to collaborative recommendation systems, content-based systems heavily rely on so called content information. Content information refers to the actual content of different items. The name stems from early recommendation systems which mainly focussed on recommending media. Given the assumtpion that a user likes to see items that are similar to each other (e.g. a user who mostly likes action films), the idea is to actually analyze the content of an item, be it the images in a movie, the

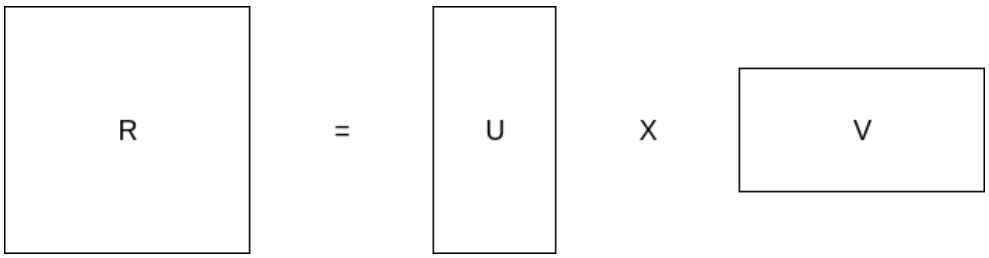

Figure 2.1: Matrix Completion via Matrix Factorization

soundwaves of a song or the text in a book. However, this method can also be used in the context of online shopping, where the "content" of an item might be its textual description or an image of the actual product.

2.1.3 Popular Examples

In the following section, we will look at some popular examples of recommendation engines and their properties.

Collaborative Filtering via Matrix Factorization

As the name suggests, this model is a collaborative approach to a recommendation engine, meaning the model uses primarily interaction data between users and items. As mentioned in [12], the main idea behind collaborative filtering is to recommend items to the user based on what users with similar taste liked in the past. Collaborative filtering refers to a class of models that use similar users' tastes to recommend items. Furthermor this class of models takes the user as a basis for recommendations and not the active item, making it a user-based recommendation system. As seen in [1], there are many ways of implementing a collaborative filtering system. However, the essential part is the problem representation.

In the Model-Based approach, the problem is represented as a matrix completion problem. The matrix R in figure 2.1 represents the interactions of users and items, where r_{ij} represents the interaction of user i with item j. Usually, when dealing with such interaction data, this matrix is very sparse, since in general, a user interacts with a small set out of possibly millions of items. The main idea behind this approach is to fill in the missing entries of the matrix. In order to achieve this, two randomly randomly initialized matrices U and V are defined and when multiplied produce a matrix of the same shape as U. As explained in [1], these two matrices represent low rank representations of users and items respectively. The next step is to use an optimization algorithm to fit U and V such that $r_{ij} \approx UV_{ij}$. The error of the chosen optimization algorithm however, is only applied to entries of R which are known. Therefore, when we find U and V such that the values

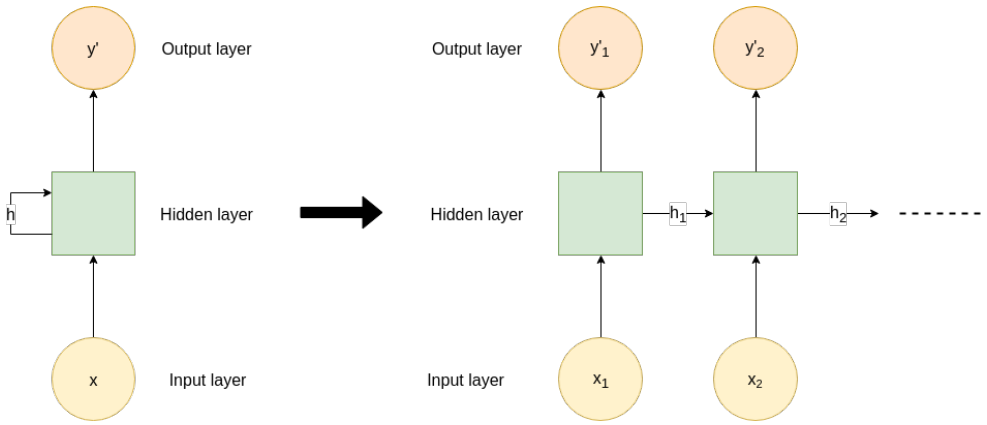

Figure 2.2: Recurrent Neural Network

for the known entries match the values in R, we assume that we also found a good approximation for the values unkown in R and use these to predict the interaction of the respective users and items.

Often Bought Together

Often bought together is a recommendation system very popular in online stores. The idea behind it is to recommend products that complement the one the user is intending to buy. A classical example for this would be to recommend a protection case when the users adds a smartphone to the basket, therefore it is an item-based approach. The implementation of this recommendation system can be done in a rather simple way, but also be improved a lot by complex systems, which could personalize the recommendations by using the users history, and thereby extending it with a user-based component. However, the basic idea remains to find items that are frequently bought together. The simplest, yet still effective, implementation of this is to count the products appearing in the same order as other products, i.e. for each combination of two products i and j, we will have a count c_{ij} of how many times these products were bought together. When a user adds a product to the basket, the products with the highest count are extracted from a database and recommended to users.

2.2 Concepts

2.2.1 Recurrent Neural Networks

RNNs are a form of artificial neural networks, which allow to explicitly model sequence input data, because they allow the ouput of the network to be fed back in. Furthermore, RNNs carry a so called hidden state, which is

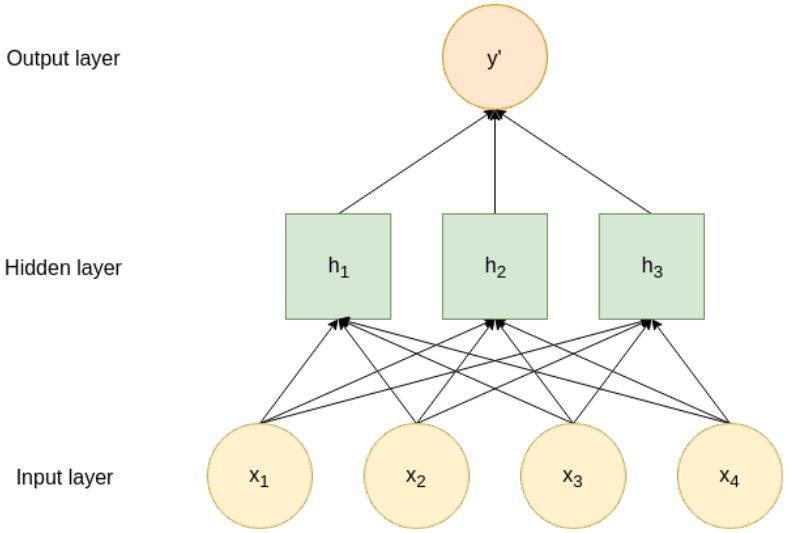

Figure 2.3: Feedforward Neural Network

propagated along the temporal axis. The following two equations from [7] describe the general behavior of a simple RNN unit as illustrated in figure 2.2.

$$h_{i+1} = \sigma(W^{hx}x_i + W^{hh}h_i + b_h) \tag{2.2}$$

$$y'_i = \text{softmax}(W^{yh}h_i + b_y) \tag{2.3}$$

We can see in equation 2.2 how the hidden state is carried over. This equation represents the connections across the temporal axis mentioned above. Figure 2.2 shows how the sequence input denoted by x is unfolded and progressively fed into the network. For each timestep, the corresponding output is computed as well as the hidden state that is carried over to the next timestep. Intuitively, the hidden state can be seen as the variable holding the relevant information from the previous steps to influence the prediction of the next step.

For clarification purposes, it is important to note that the individual inputs x_t represent a datapoint in a sequence denoted by x, and therefore in most cases are represented by a vector. In a feedforward neural network as illustrated in figure 2.3, the inputs x_i represent individual entries in the feature vector of the datapoint x.

Backpropagation

Backpropagation is the technique used to train neural networks. Let us assume we have a simple feedforward neural network as shown in figure 2.3, the behavior of this network is described by the following equations.

$$h_i = \sigma_h(w_{h_i}^T x) \tag{2.4}$$

$$y' = \sigma_y(w_y^T h) \tag{2.5}$$

Each individual cell has a weight vector associated with it, denoted by w_o where o is the respective cell. The arrows in the illustration represent the individual weights of such a weight vector. Each layer has a nonlinear activation function associated with it, denoted by σ_l, l being the respective layer.

In this setting, we usually are in a supervised mode, where for each input x there is a true label y. Using the neural network we estimate y for a given x by applying the equations above to obtain the estimation y'.

The last component needed before training is possible is a loss function $\mathcal{L}(y, y')$, which quantifies the error of the estimation y'.

The basic idea of backpropagation is to attribute parts of the error computed to the different units of the network, and therefore make adjustements to the weight vector of a particular unit. As described in [7], this is done by using the chain rule to compute the derivative of $\mathcal{L}(y, y')$ with respect to each weight vector. The weight vector is then adjusted by gradient descent, i.e. by moving the vector in the direction of the largest decrease in the gradient. Even though there are more parameters involved as for instance the learning rate, they are not essential for understanding the principle.

Backpropagation with RNNs

Also in [7], the authors mention that training RNNs has been known to be especially difficult due to the *vanishing* or *exploding gradients* problem. Given the example of the RNN in figure 2.2, we compute the estimation y'_t, i.e. the output of timestep t. As we can see in the equations 2.2 and 2.3, this output is influenced by both, the current timestep and all the previous timesteps. The problem now arises from the so called recurrent weights, denoted by W^{hh}, connecting the units across the temporal axis. Assuming that the weights are small, i.e. $|w_{ij}| < 1$ for $w_{ij} \in W^{hh}$, the contribution of input x_{t-k} to the output y'_t gets exponentially smaller with increasing k. By computing the gradient with respect to the input, the afromentioned contribution is quantified. Therefore, when k becomes large and the contribution of the input x_{t-k} vanishes, the gradient with respect to the input unit vanishes as well, resulting in the so called vanishing gradient problem. The opposite happens when the recurrent weights are large, i.e. $|w_{ij}| > 1$ for $w_{ij} \in W^{hh}$. In this case, the contribution gets amplified when k becomes large, which would lead to exploding gradients. This represents problem in a setting where we want to learn long-range dependencies, which is often the case in sequence modeling.

Gated Recurrent Unit

The Gated Recurrent Unit (GRU) is a recurrent unit which aims to solve the exploding or vanishing gradients problem mentioned above. This is achieved by modifying the behavior of the hidden units, in between which the recurrent edges of the network are. The authors of [3] introduced this new type of recurrent unit. They retrofitted the recurrent unit with the ability to adaptively remember and forget, by using the *reset* and *update* gate.

The reset gate is responsible for supressing features of the hidden state that were learned to be unimportant for the future. The reset gate is computed using the following formula:

$$r_t = \sigma(W_r x + U_r h_{t-1} + b_r) \tag{2.6}$$

The matrices W_r and U_r as well as the bias b_r are parameters specific to the reset gate and are learned simultaneously with backpropagation.

The update gate controls how much information is propagated from the previous hidden state to the next one, i.e. it learns which features of the previous hidden state will be how important in the future. The update gate is computed as follows:

$$z_t = \sigma(W_z x + U_z h_{t-1} + b_z) \tag{2.7}$$

Again, the weight matrices and bias are specific to the update gate and learned via backpropagation.

To compute the next hidden state, the computed gates are used as follows:

$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tanh(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h) \tag{2.8}$$

Equation 2.8 illustrates how the gates function. When the reset gate is close to 0 in some features, the hidden state is forced to ignore that information before being multiplied with the weight matrix of the recurrent unit.

On the other hand, the update gate controls how much information should be propagated directly from the hidden state, versus the hidden state resulting from the new input. If the reset gate is 1 and the update gate is 0, we get the same formula as in 2.2, effectively returning to the vanilla RNN formulation.

2.2.2 Embeddings of categorical variables

A categorical variable is a variable that can take on one of a limited number of values. Examples are colors, locations or, in the context of e-commerce, product IDs. When using such variables as an input variable for some model,

Color	ID
Red	0
Green	1
Blue	2
Yellow	3

Table 2.1: Integer IDs for Colors

Color	One-Hot Encoding
Red	$[1, 0, 0, 0]$
Green	$[0, 1, 0, 0]$
Blue	$[0, 0, 1, 0]$
Yellow	$[0, 0, 0, 1]$

Table 2.2: One-Hot Encoding for Colors

the variables need to be transformed into a numerical representation, allowing the variable to be used. Assuming that we have a variable which describes the color of a specific product – for reasons of simplicity with only the four colors red, green, blue, and yellow – one possibility would be to just integer IDs to the different colors as shown in table 2.1. This approach entails the problem that integers have a defined order. That means that any model will interpret the color yellow as larger or as a stronger signal, than the color red. This is not desirable, in the case of colors, all of them should have the same magnitude.

A different way of representing categorical variables is the so called one-hot encoding. A one-hot encoding requires a continuous ID space, such as the one in table 2.1. The same colors encoded in a one-hot encoding can be seen in table 2.2. This ensures all the color representations have the same magnitude, therefore the model will get the same signal strength from each of the colors. We basically transform the id encoding to a one-hot encoding by using a zero vector of dimension $\max(\text{ID})$ and set the element at the position $\text{ID}_x - 1$ to 1, for item x. Therefore, the one-hot encoding always has the same dimensionality as the number of categories. There are still two major drawbacks to this approach. As we will see in section 3.4, the number of categories for such a variable can reach into the millions. This means that the dimensionality of the single categorical feature can range into the millions. This is not desirable, since high dimensionality leads to more difficulties in training and more computing resources needed to accomplish the training task. The second drawback is that this representation does not account for the notion of similarity, i.e. red is as similar to blue as it is to

Color	Embedding
Red	$[1, 0, 0]$
Green	$[0, \sqrt{0.5}, \sqrt{0.5}]$
Blue	$[0, 1, 0]$
Yellow	$[0, 0, 1]$

Table 2.3: Possible Embedding for Colors

yellow. However, it is known that green is a composition of blue and yellow, so green should be more similar to blue and yellow than to red.

These two issues can be solved by using embeddings. Embeddings are a low dimensional real numbered representation of a categorical variable; similar variables will have similar vectors representing them. The difficulty with embeddings is finding a process which produces the embeddings with the desired properties. For a small number of variables it is possible to manually construct them. In table 2.3, we can see a lower dimensional representation of the same colors, however, the distance between green and yellow or blue is smaller than the one between green and red. As all the representations still have the same magnitude none of the colors triggers a stronger signal.

2.3 Previous Work

2.3.1 Meta-Prod2Vec

Meta-Prod2Vec is a method proposed in [15] to compute such embeddings as described in the previous section. The original inspiration for this approach comes from the famous *word2vec* model (c.f. [8]), a method of computing embeddings for words, respecting the similarity between them. Based on word2vec, Yahoo developed a model named *prod2vec* (c.f. [5]), that computes embeddings of products using the receipts their customers are receiving in their inbox. Meta-Prod2Vec is an extension of prod2vec that allows to include meta information about products, such as brands and product type.

The fundamental assumption behind all of the afromentioned models is the famous Distributional Hypothesis [13], which states that words appearing in the same context have similar, if not identical, meaning. For illustration purposes consider the following two example sentences:

- George likes to drink a glass of wine in the evening.
- George likes to drink a glass of whiskey in the evening.

Without knowing the meaning of the words "wine" or "whiskey", one could deduce that they are at least similar concepts. Essentially, the context of a

word limits the number of possible words that can appear in that context. If that context is restrictive enough, there are only few possible words or concepts that can occur in that context. This assumption is translated into the context of e-commerce by interpreting products as words and sessions or purchase sequences as sentences.

Since Meta-Prod2Vec is an extension of prod2vec, it makes sense to first look at the loss function of prod2vec in equation 2.9:

$$L_{J|I} = \sum_i X_i H(p_{\cdot|i}, q_{\cdot|i}) \tag{2.9}$$

Where X_i is the number of occurences of word i and $H(\cdot)$ is the cross entropy loss. J describes the output space, whereas I is the input space, in this context they are the same, specifically all the possible products. The loss function is the weighted cross entropy loss between the empirical probability distribution $p_{\cdot|i}$ and the predicted probability distribution $q_{\cdot|i}$. The weighted cross entropy loss forces the predicted probability for any product to be close to the observed empirical probability. This means that the learned probability function $q_{\cdot|i}$, in the case of observed events, is close to to the empirical probability distribution, while also allowing predictions of yet unseen events.

In the next step, we look at the learned probability function $q_{\cdot|i}$, and how it is related to the embeddings of products. In equation 2.10 we can see how $q_{\cdot|i}$ is defined:

$$q_{j|i} = \frac{\exp(w_i^T w_j)}{\sum_{j \in V_j} \exp(w_i^T w_j)} \tag{2.10}$$

This defines the probability that product j is in the context of product i. In principle this means, that the inner product $w_i^T w_j$ is interpreted as an unnormalized probability, and by applying the softmax function, we obtain a probability distribution over all possible context words V_J.

From looking at the two equations above, it is apparent that computing this loss function is rather expensive, the complexity is $O(n^2)$ for n being the number of products. Therefore, these models are usually trained using negative sampling loss [9].

The loss function of Meta-Prod2Vec, as seen in equation 2.11 is a natural extension of the loss function of prod2vec.

$$L_{MP2V} = L_{J|I} + \lambda \cdot (L_{M|I} + L_{J|M} + L_{M|M} + L_{I|M}) \tag{2.11}$$

Where λ is a regularization parameter controlling how much the side information influences the total loss and M describes the metadata space. In the following, the different components of the loss function are presented:

- $L_{J|I}$: This is the same term as in equation 2.9, it describes the cross entropy between the observed conditional probability and the modeled conditional probability of the output products conditioned on the input products.
- $L_{M|I}$: This is the loss component of the probability distribution of metadata values of surrounding products, given the input product.
- $L_{J|M}$: This component measures the loss of the probability distribution of surrounding products, given the metadata values of the input product.
- $L_{M|M}$: This is the loss incurred from the probability distribution of surrounding products metadata, given the metadata values of the input product.
- $L_{I|M}$: Finally this component measures the loss of the probability distribution of the input products, given their own metadata values.

In principle, the objective function in equation 2.9 is extended to include all possible interaction terms with the metadata. The advantage of this loss function is that it does not change the training algorithm of prod2vec. The only thing changing is the data generation process; additionally to creating the pairs of co-occurring products, also co-occurring metadata and products are generated. Since the individual components of the loss function are computed in exactly the same way, adding these co-occurence terms will automatically compute the total loss. Thus, the metadata values will be embedded in the same output space.

2.3.2 Personalized Session-based Hierarchical Recurrent Neural Network

The work presented in [11] serves as a basis for this thesis. The authors present a model architecture that can model the session behavior of users, as well as learn historical user preferences and apply those to the recommendations. This makes it a user- and session-based recommendation system. The architecture presented is based on a model called *gru4rec* presented in [6]. In figure 2.4, we can see an illustration of the model architecture. Essentially, the difference between gru4rec and hgru4rec is the addition of the the second level GRU, GRU_u, which keeps track of historical user preferences. To formalize the sessions of users, we define the sessions of a specific user u as $\mathcal{S}_u = \{s_u^1, s_u^2, \cdots, s_u^n\}$. A session is defined as $s_u^k = \{e_1^k, e_2^k, \cdots, e_m^k\}$, where an event e_n^k is the product id of the product viewed in session k at position n.

The proposed model uses the session level GRU, GRU_s, to predict the next item that the user would view. The hidden state h_s serves as a fixed size representation of the session. The input to the GRU is defined as follows:

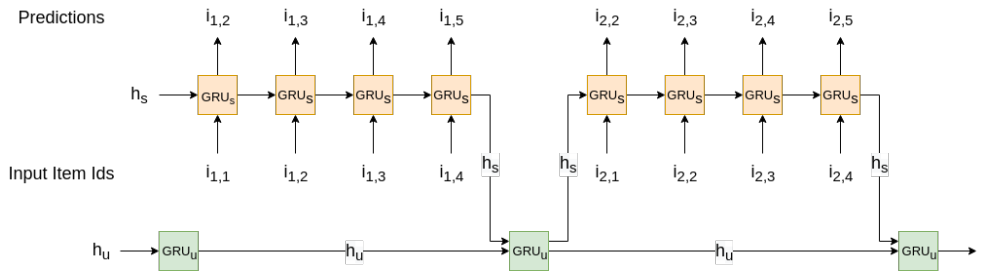

Figure 2.4: hgru4rec model architecture

$i_{m,n} = \text{onehot}(e_n^m)$, where $\text{onehot}(x)$ is the one-hot encoding of the integer x. Now the output and the next hidden state are computed, using the formula introduced in 2.2.1 as follows:

$$r_{m,t}, h_m^t = GRU_s(i_{m,n-1}, h_s^{t-1}) \tag{2.12}$$

Where h_m^t is the session hidden state of session m at time t and $r_{m,t}$ is a score for every item to be recommended in session m at time t. The output of the GRU, $r_{m,t}$, is compared to the one-hot encoding of the next item, i.e. $i_{m,t+1}$ to compute the loss function. The loss function was explicitly designed by the authors for this task. The TOP1 loss is the regularized approximate relative rank of the relevant item. The relative rank of the relevant item in the next step is defined in equation 2.13:

$$\text{relative rank} = \frac{1}{N} \cdot \sum_{j=1}^{N} I\{r_{m,t}^j > r_{m,t}^i\} \tag{2.13}$$

Where N is the number of items, $r_{m,t}^k$ is the score of item k in session m at time t and finally i is the relevant item in the next step. This function computes the number of items which have a higher score than the relevant item. However, this relative rank has two drawbacks: First, the indicator function is difficult to derive and second, the computation of the loss for one item involves the computation of the score for all possible items. Therefore, the relative rank is approximated by the equation 2.14, which further includes a regularization term forcing the irrelevant item scores towards zero.

$$L_{TOP1} = \frac{1}{N_S} \cdot \sum_{j=1}^{N_S} \sigma(r_{m,n}^j - r_{m,n}^i) + \sigma(r_{m,n}^j) \tag{2.14}$$

Where N_S is the number of sampled irrelevant items to avoid the computation of the scores for every item and σ is a sigmoid function. The sampling method will be described in more detail, when presenting at the dataset (c.f. 3). This loss function includes an additional regularization term, which should motivate the scores of the sampled irrelevant items to be close to 0.

After a session is completed, the user-level GRU, GRU_u, comes into play. The task of this GRU is to learn how the session representation of users change over time. This is achieved by using the fixed size session representation as the input to GRU_u. The output of GRU_u then will serve as the new initial session hidden state for the next session.

$$h_u^m, h_m^0 = GRU_u(h_u^{m-1}, h_{m-1}^{n_{m-1}}) \tag{2.15}$$

Where h_u^m is the hidden state of user u at the beginning of session m and n_m is the number of items in session m.

Both GRUs are trained jointly using the same loss function.

2.4 Key Performance Indicators

Key Performance Indicators (KPIs) are metrics used in a business process to measure the performance of said process. In online services there are a few well established metrics; most relevant for this work are the click-through rate and the conversion rate. These metrics are also often used in testing multiple versions of the same element, for example a new design of a button. By measuring these metrics, it can be determined which version most users prefer.

Click-Through Rate The click-through rate (CTR) measures how many users click on a specific element relative to the number of impressions. Impressions are defined as the event when a user has the specific element that is measured in his viewport. Finally, the viewport is the part of the website or web application which is visible to the user in his browser window. In equation 2.16, its formalization is shown.

$$\text{CTR} = \frac{\#\text{clicks}}{\#\text{impressions}} \tag{2.16}$$

Conversion Rate The conversion rate measures how many successful transactions follow the interaction with a specific element. A transaction can be defined in a rather abstract way, such as listening to a song, or watching a video to the end. In the context of e-commerce however, the conversion rate measures how many successful sales are made following an interaction with an element. This is formalized in equation 2.17.

$$\text{Conversion Rate} = \frac{\#\text{sales}}{\#\text{impressions}} \tag{2.17}$$

Since there might be multiple elements for which the conversion rate is measured, there needs to be a strategy of attributing sales to specific elements in

the web application. An exact way of computing the conversion rate is therefore difficult to find, but there are multiple ways to approximate this value. One common way is if the user sees an impression of a specific element and completes a purchase in the same session, then this element receives credit for the sale. This approach entails the problem that the user is not tracked across multiple sessions. The session in which the user finds this item, and the session in which the user actually makes the purchase are often not the same. Also, if the user receives an impression, it is not clear whether the user actually registers the item, or if the user is focused on another part of the application. Finally, this method credits sales to an element even if the purchased product was not even recommended. Thus, we chose another approximation of this metric. If a user purchases an item that was recommended and then clicked by the user, and the click is less than 14 days before the purchase, the element receives credit for the sale. The number 14 days was chosen because more than 99% of product sales, which were clicked by the users after being recommended, fall into this range.

Chapter 3

Dataset

3.1 Data Collection

In this work, we will use the data generated by the tracking systems of Digitec Galaxus AG. The following sequence-diagram provides an overview of the data collection process.

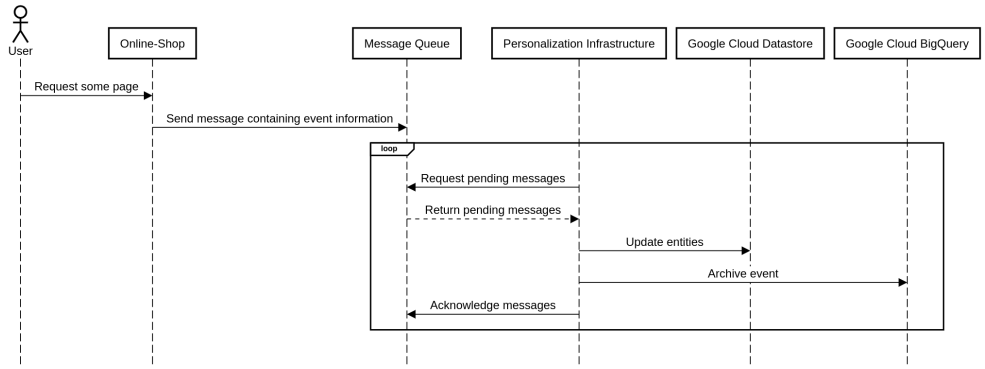

Figure 3.1: Collecting user interaction data on digitec.ch and galaxus.ch

When a user requests a specific page, the online shop Application will collect some information on the user such as UserId, requested page, User Agent. This information is packaged as a message representing this specific event and then sent to a message queue. Subsequently the Personalization Infrastructure will process these events by requesting batches of unprocessed messages. For each event then the Personalization Infrastructure will first update the involved entities and then archive the event. In the former case a managed Key-Value store (Google Cloud Datastore) is used to store entities. Examples of such entities are shopping carts, orders, and last viewed prod-

ucts. These entities basically represent the current state of various entities appearing in the context of the online shop. In the latter case, a managed data warehousing solution (Google BigQuery) is used in order to be able to store large amounts of data in append-only tables. The data stored there is mainly denormalized, in order to enable simple extraction. Each row in these append-only tables contains all information belonging to a single event produced by a user. The data stored in the Key-Value store essentially is the sum of all events stored in the data warehouse.

3.2 Data Preparation

In order to be able to focus on the model implementation when implementing the model, we want to prepare the data for congestion as far as possible. Therefore, the extraction and preparation of the dataset is implemented separately from the model implementation.

Data Extraction In the first step, we extract the raw data from the data warehouse, selecting the following pieces of information: ProductId, LastLoggedInUserId, UserId, SessionId, User Agent, Timestamp. Most of the properties are self-explanatory, except LastLoggedInUserId. This property represents the UserId last seen on a specific device accessing the online shop. When not actively logged into the accout, we would not know which accessed the page. However, using this property, we can complete missing UserIds in the dataset. The data extracted is limited to the events producted by visiting a product detail page as seen in figure 3.2. This filtering is done because this work focuses on recommending products, in another setting, other data might also be relevant. The extracted data is stored in several shards of CSV files, each shard approximately represents the events of one day.

Cleaning Data In the next step, the data is cleaned, which involves mainly two steps: Anytime we encounter an event where the user is unkown, we try to complete the information with the LastLoggedInUserId. If we do not know the LastLoggedInUserId either we discard the event.

There are many well-known bots roaming the internet collecting various types of information on websites. Most famously the Google Bot which crawls the content of any website to determine the quality of the site and to influence the rank of the website in Google searches. There are some open-source lists that collect the User Agents of these bots. The User Agent of a client accessing a website usually describes the type of device used, in the case of ”good” bots, they explicitly tell the server that the ”device” accessing

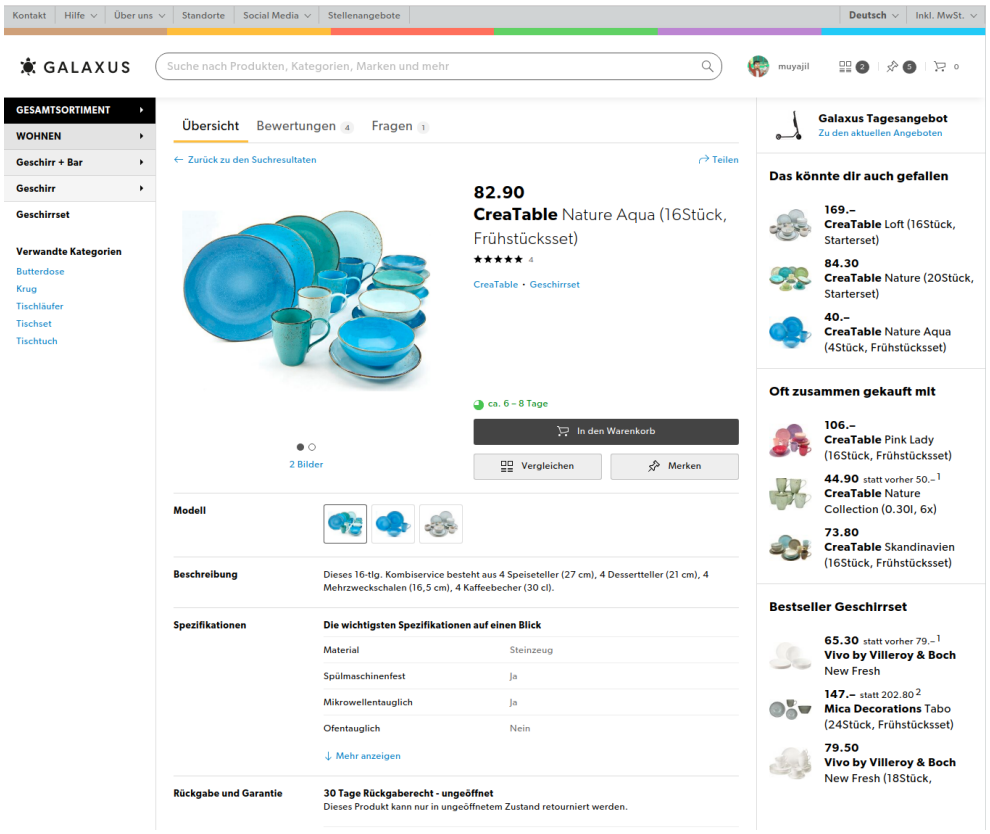

Figure 3.2: Product Detail Page on galaxus.ch

the website is actually a bot. Using one of these lists[1] the events produced by bots are removed from the dataset.

Aggregation of Events The next step is to aggregate the events on two levels: first grouping events that were generated in the same session and second grouping these sessions by users. The resulting data structure looks as fol-

[1] https://raw.githubusercontent.com/monperrus/crawler-user-agents/master/crawler-user-agents.json

```
    "<UserId_1>": {
        "<SessionId_1>": {
            "StartTime": "<Timestamp of first event>",
            "Events": [
                {
                    "ProductId": "<ProductId_1>",
                    "Timestamp": "<Timestamp_1>"
                },
                {
                    "ProductId": "<ProductId_2>",
                    "Timestamp": "<Timestamp_2>"
                }
            ]
        }
    }

```

Listing 3.1: Data Structure for user-events

lows:

This is done by processing one shard after another; when a shard is finished, the JSON representation of this shard is saved in a separate file because the large amount of datapoints cannot be kept in memory altogether.

Merging shards Since the shards only approximate the events of one day, it is not possible to assume that a session is completely represented in one shard, it might be distributed across multiple. Therefore, it is necessary to merge the data in the different shards, but since it is not efficient to keep all the information in one file, a different method of paritioning is required. As we will see later, it is further necessary that all the events produced by a single user are stored in the same file. An easy way of achieving this, is by partitioning by UserId. Each shard is processed as follows:

```
    for shard in shards:
        for i in range(num_target_files):
            relevant_user_ids = list(filter(lambda x: int(x) %
num_target_files == i, shard.keys()))
            output_path = merged_shards_prefix + str(i) + '.json'
            output_file = json.load(output_path)
            for user_id in relevant_user_ids:
                for session_id in shard[user_id]:
                    # Merge events from output_file and shard

```

Listing 3.2: Merging shards

Filtering Having some number of files containing all the information relevant to some users, some datapoints have to be filtered out, which is also mentioned by the authors of [11]. Specifically the following:

- Remove items with low support (min 5 events per product)

- Remove users with few sessions (min 5 sessions per user)
- Remove sessions with few events (min 3 events per session)

Those datapoints are removed since items with few interactions are suboptimal for modeling; too short sessions are not informative, and finally users with few sessions produce few cross-session information.

Subsampling Prototyping models with very large datasets is inefficient. Therefore, several different sizes of the dataset were produced, by defining the approximate number of users and the exact number of products desired in the dataset. This was done by sampling from the set of products with a probability proportional to the number of events on that product. Subsequently, the partitions of the data were processed, iterating through the sessions of the users. For each sessions we remove the products that were not chosen to be kept and keep the session if it is still long enough. Furthermore, a user is kept in the dataset if the number of filtered sessions remains still large enough. This process is repeated for the different partitions until the number of users is larger than the approximate number of desired users.

Embedding Dictionary As we will see in 4.1, a version of the model uses one-hot encodings for representing products. Therefore, the product IDs referenced in the dataset need to be mapped into a continuous ID space, otherwise it is not possible to produce one-hot encodings. The process starts with EmbeddingId 0, then iterates through all the sessions, and each time a product is seen for the first time, the EmbeddingId is assigned to the product increased by 1.

3.3 User Parallel Batches

Since we are dealing with sequence data, it is not trivial to produce batches for the model to ingest. Because the sequences can have different lengths, and the number of sessions varies from user to user. User Parallel Batching is a way of having fixed size batches that can be ingested by the model, while allowing for sequences to have different lengths and users to have a different number of sessions. Usually, in these cases the sequences are padded with some neutral value, however in this case, there is no neutral product, and the introduction of such a neutral product might influence the results. When processing sequence data, it usually means that a datapoint is the input for the next datapoint, which requires the ordered sequence data. To illustrate these batches, we assume that there are 4 users with the sessions as in figure 3.3. A batch of size x will contain an event from x different users. Essentially, the labels are the event that happens after the input event.

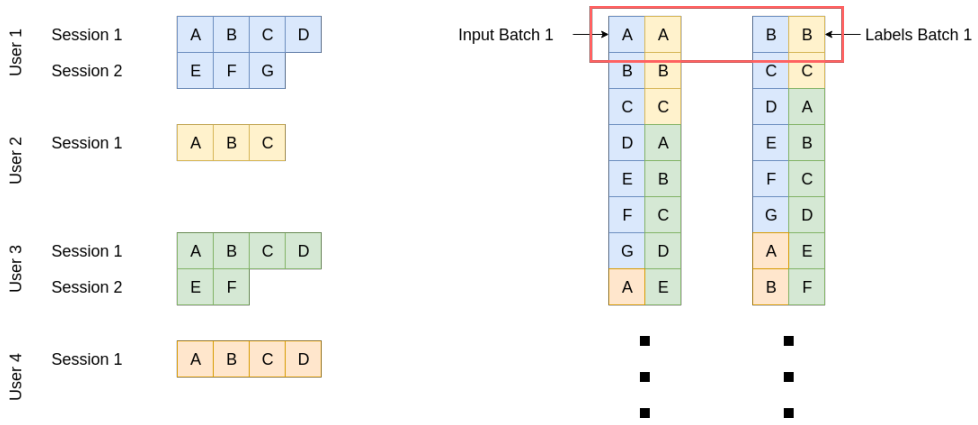

Figure 3.3: User Parallel Batches

Property	MINI Dataset	MIDI Dataset	MAXI Dataset
#Users	23	15'242	242'797
#Products	161	42'103	470'817
#Events	633	1'234'697	28'726'701
#Sessions	183	250'187	4'652'496
#Events per Product	3.93 $\pm$ 5.84	29.33 $\pm$ 69.98	61.27 $\pm$ 263.3
#Events per User	27.52 $\pm$ 19.8	81.01 $\pm$ 135.35	115.05 $\pm$ 215.15
#Events per Session	3.46 $\pm$ 0.82	4.94 $\pm$ 3.07	6.17 $\pm$ 60.22
#Sessions per User	7.96 $\pm$ 5.41	16.41 $\pm$ 22.08	19.16 $\pm$ 28.67

Table 3.1: Dataset Properties

The data representation shown in listing 3.1 is chosen explicitly to enable efficient generation of user parallel batches. To compute the loss function mentioned in 2.3.2, we sample the irrelevant items by using the items from the different sessions in the same batch. This basically enables popularity based sampling, since the samples are directly taken from other sessions.

3.4 Dataset Properties

Using the afromentioned sampling step, we generated different sizes of the dataset to enable model prototyping. In table 3.1, we can see the properties of the different datasets. The MINI dataset is used to ensure that the model works. The goal is to completely overfit to the training data and remember everything, which allows to quickly debug and validate the gradient flow through the model. The MIDI dataset is used for experiments with different model sizes. The purpose of this dataset is to try out different models on a rather realistic dataset, which is of similar size than the ones used in [11],

and to be able to train these models in a reasonable time frame. The MAXI dataset on the other hand, contains all events that were not filtered in the data preparation process. Depending on the model size, training on this dataset can take several days, which is why only the models that performed best on the MIDI dataset were trained on the MAXI dataset. Finally, the models that are deployed in the experiments are all trained on the MAXI dataset.

Chapter 4

System Overview

4.1 Model Architecture

4.1.1 hgru4rec

In section 2.3.2 we introduced the model architecture for hgru4rec. This work proposes and improvement of this model by using the model introduced in 2.3.1. As mentioned before hgru4rec uses a one-hot encoding of items as the input, as seen in equation 2.12. We propose to use pre-computed embeddings resulting from Meta-Prod2Vec as a replacement for the one-hot encodings. Therefore the model architecture is not different from the one seen in figure 2.4. Meta-Prod2Vec is implemented and trained independently of the session-based model. After training Meta-Prod2Vec, the product embeddings are extracted and saved to a key-value store. Afterwards hgru4rec can access this key-value store during training and inference such that the model can consume the product embeddings instead of the one-hot vectors. Further we will test hgru4rec without the user layer, to verify the improvement recorded by the authors of [11] using our dataset. Therefore this work investigates 4 different architectures documented in 4.1.

Model training After testing different optimizers the Adam optimizer performed best, therefore we used this for training the hierarchical RNN. The model was trained until there was no more improvement in the validation metric. The mean reciprocal rank (MRR) at 10 was used as the validation metric. The reciprocal rank is defined in equation 4.1

$$\mathrm{RR} = \begin{cases} \frac{1}{\mathrm{rank(label)}} & \text{if label } \in \text{predictions} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

Where predictions are the Top 10 predictions returned by the model and

Model Name	One-Hot	Embedding	User Layer
OnlySessionOneHot	x	-	-
OnlySessionEmbedding	-	x	-
WithUserOneHot	x	-	x
WithUserEmbedding	-	x	x

Table 4.1: Model Architectures

rank(x) is the position of x in those predictions. The MRR is the mean of the RR over all datapoints.

4.1.2 Meta-Prod2Vec

In section 2.3.1 we introduced the model architecture for Meta-Prod2Vec. The model has a rather simple architecture, consisting of a single embedding layer, which is fitted using the loss function shown in equation 2.11 The following side-information was used:

- Brand
- Product Type
- Price Class

The product type specifies which category a product is in, for example "Mobile Phone" or "Dining Table". The mapping from products to brands and product types is very simple, since these are two properties each product must have. The price class is computed within a product type. The idea is to classify products into a a number of classes, such that the model can learn to fit "premium" products closer together. To achieve that we extract a number of quantiles from all the prices of the products within a certain product type. Then we assign IDs to the different quantiles, and this ID is then attached to the product as an additional side information. Afterwards the product and metadata space are joined, by creating and embedding dictionary that maps all the entities from the product and metadata space to a single contiuous ID space. This is done since the input to the projection layer is a one-hot encoding of the entity, since obviously we are again dealing with categorical data. As we have seen in equation 2.11 the loss function would support a different weighting of the different types of side-information, however we found that this does not help the model learn better embeddings, therefore we use equal weights for all the types of side-information.

Model training The model is simply trained over two epochs. This decision was made because the second pass over the data decreases the loss significantly, whereas the third pass does not.

4.2 API

The models are implemented in Tensorflow[1]. Tensorflow provides a mechanism to export tensorflow models as so called SavedModel[2], which is the way Tensorflow serializes models universally. As is commonly known Tensorflow builds models as a graph, where each operation is a node in the graph and referred to as an ”operation”[3]. To serialize the model the export mechanism needs to know which operations represent the input and the output of the API respectively. This allows the export mechanism to strip away all nodes in the graph that are not used in the path in the graph from the input nodes to the output nodes, which greatly reduces the model size as well as the complexity. A good example of what is stripped away is the optimizer and the loss function nodes. In inference mode these are not useful anymore, therefore it makes sense to remove them from the graph. After the model is serialized it can be used together with a premade Docker image[4] which allows the model to be served as a REST API. The inputs and outputs of the API are described in listing 4.1 and 4.2 respectively.

```
1     {
2         "inputs":
3         {
4             "EmbeddingIds": [<Input Product EmbeddingId>],
5             "SessionEmbeddings": [<Session Embedding>],
6             "UserEmbeddings": [<User Embedding>], # Only present
    if the user layer is used
7             "ProductEmbeddings": [<Product Embedding>], # Only
    present if product embeddings are used
8             "SessionChanged": [<Session Changed>]
9         }
10    }
11
```

Listing 4.1: TF Serve API Input

[1] http://tensorflow.org

[2] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/saved_model/README.md

[3] https://www.tensorflow.org/api_docs/python/tf/Operation

[4] https://hub.docker.com/r/tensorflow/serving

```
{
    "outputs":
    {
        "SessionEmbeddings": [<Updated Session Embedding>],
        "RankedPredictions": [<Ranked EmbeddingIds>]
    }
}
```

Listing 4.2: TF Serve API Output

As can be deduced by looking at these listings the exported model itself does not keep track session embeddings, instead it returns the updated embedding with the predictions. This decision was made because of the production setup. To enable easy horizontal scaling of the recommendation system, the deployed application should be as stateless as possible. Horizontal scaling is important in this case, since at peak times there thousands of users online, serving all those with a single instance of the model is not possible. This can be achieved by storing all the embeddings and embedding IDs in a key-value store which supports concurrent access.

It also makes a lot of sense to abstract the concept of embeddings and embedding IDs from the online shop requesting the recommendations. Therefore we implemented a middleware which takes features known to the online shop and then prepares the features as well as transform the outputs for the model API. The middleware further access the afromentioned key-value store to keep session and user embeddings up to date. The input and output of API exposed by the middleware is described in listings 4.3 and 4.4 respectively.

```
{
    "UserId": <UserId>,
    "ProductId": <ProductId>,
    "SessionId": <SessionId>
}
```

Listing 4.3: Middleware API Input

```
{
    "Predictions": [<Ranked ProductIds>]
}
```

Listing 4.4: Middleware API Output

This allows for a clear separation of domains, i.e. the online shop does not know anything about the internals of the implementation of the model. Further since the key-value store supports concurrent access, both the mid-

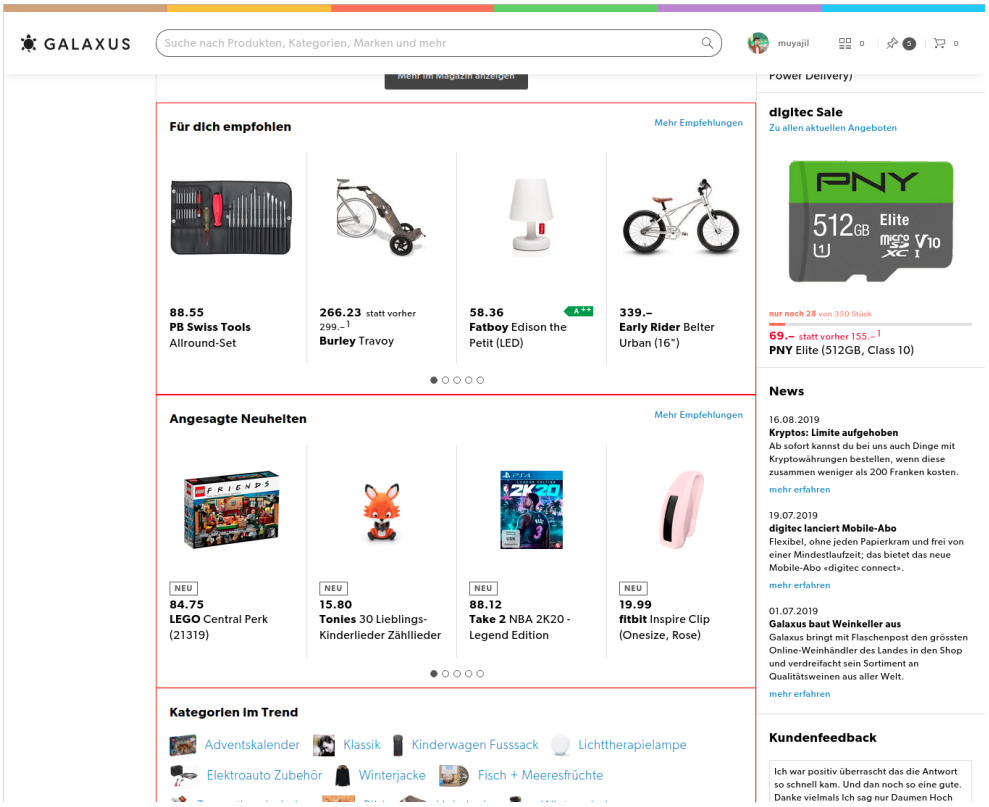

Figure 4.1: Recommendations on the homepage

dleware and the model API can be horizontally scaled without the need for any further control.

4.3 Production Setup

Digitec Galaxus AG is the largest online-retailer in Switzerland. They operate galaxus.ch and digitec.ch. The former is a general online shop comparable to Amazon. The latter is specialized in Electronics. Distributed on the different sections of the site there are several recommendation engines populating the content the users see. Examples are the landing page and multiple engines on the product detail page seen in figures 4.1 and 4.2 respectively.

There is a framework that computes probabilities which specific recommendation engine provides the content for a specific location. Further the framework then chooses the content for each location based on the probabilities computed before and some other constraints such as minimum and maximum value. However to test the model implemented in this work this framework is bypassed by a A/B Testing engine, therefore this framework is not part of this work. The specific tests and the test setup is described

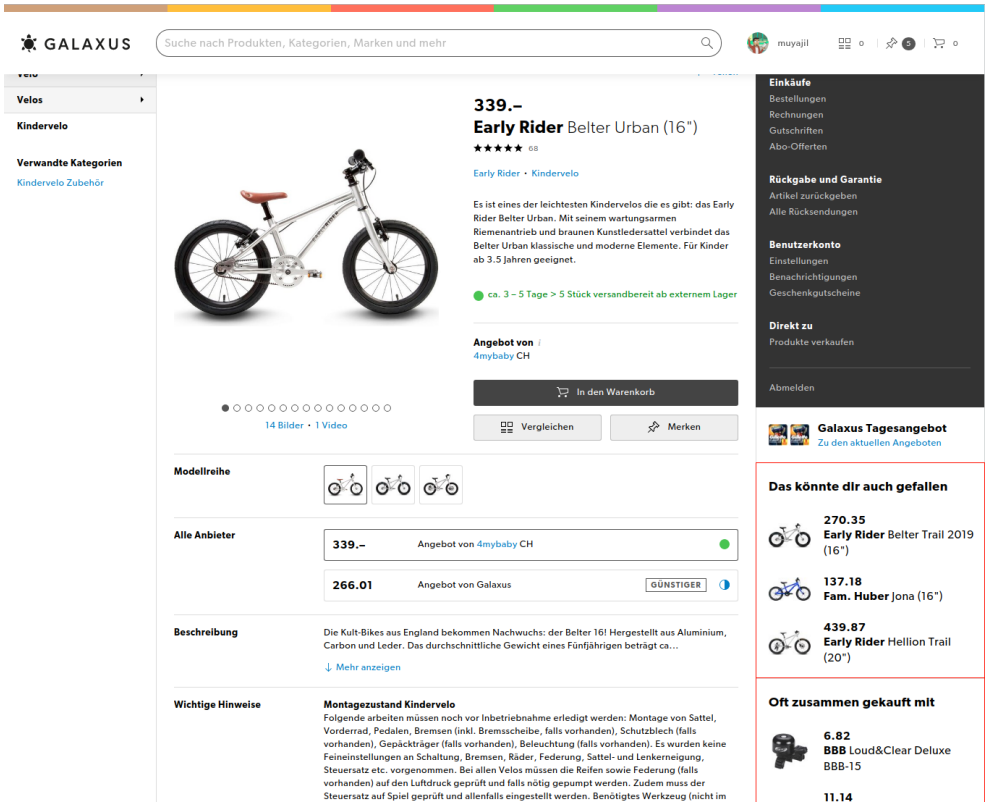

Figure 4.2: Recommendations on the product detail page

in 5.1.1, for the understanding of the following it is enough to assume that some independant system is providing recommendation requests to the recommendation system. Using the API containers described in 4.2 we can serve these requests. The sequence-diagram in figure 4.3 should give an overview on how the system is integrated in the production environment.

The whole process starts when a specific user requests a specific page on either digitec.ch or galaxus.ch. If the requested page has an element where there is an A/B test configured the online shop Application will make a request to the testing engine. The testing engine will return the API location of one of the model versions After that the Online Shop will request the content, in this case, from the Personalization Infrastructure. Note that during this setup each of the four versions of the model is deployed simultaneously, all of them trained on the same dataset. The Personalization Infrastructure then calls the Middleware described above. Before requesting a prediction from tf-serve the Middleware will gather precomputed embeddings for the involved entities. The predictions are returned to the Personalization Infrastructure and from there to the Online Shop. The last step is for the Online Shop to render the content and deliver the page to the user. This process

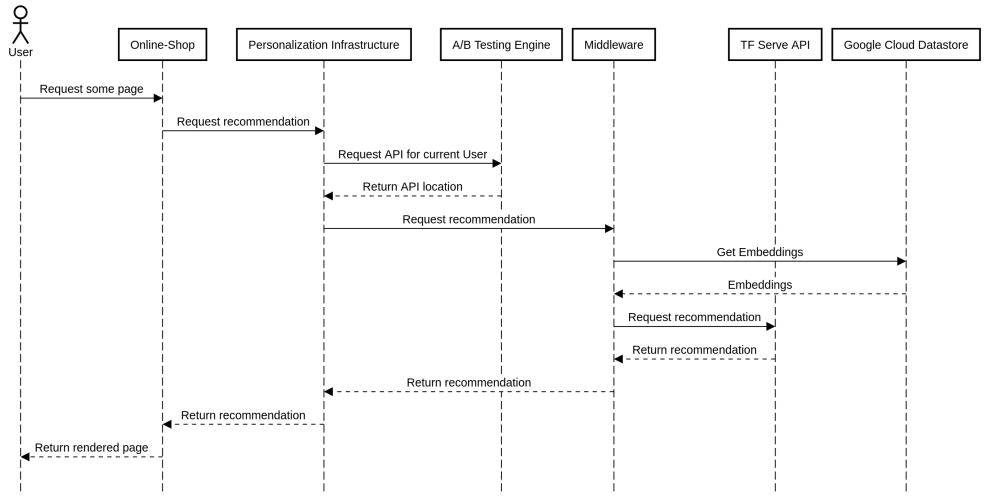

Figure 4.3: Serving Recommendations on digitec.ch and galaxus.ch

takes about 300ms.goal with this

Chapter 5

Experiments

The models described in section 4.1 are all tested in two test scenarios. The offline test scenario measures the performance of the model on historical data, whereas the online scenario measures KPIs in a production setting on new data. The reasoning behind this is that when using recommendation systems and evaluating them on historical data, the results do not necessarily reflect the real life behavior of users. This is because displaying recommendations changes the reality, i.e. what the user sees when using the application is different from what the user saw in the past without or with other recommendation systems.

5.1 Offline

5.1.1 Experiment Setup

In this scenario we do a temporal split and hold off future data for evaluation and train the model on historical data. That means that the most recent sessions of each user make up the test set, while the second most recent sessions of each user make up the validation set. Each model is evaluated on the validation set regurarly during training. The goal of this experiment is to predict the next click of the user, given the currently active product. After completing the training, i.e. after the validation metric does not increase anymore, the model is evaluated on the test set. Before running the model on the MAXI dataset, some parameters were chosen based on experiments on the MIDI dataset, since due to time and resource constraints it was not possible to train all combinations of the models on the MAXI dataset. Specifically the loss function, optimizer, batch size, learning rate, gradient clipping threshold and early stopping criterion were chosen based on the MIDI dataset, since as mentioned before it was not possible to test all these different configurations on the large dataset. Further the one-hot encoding variants used so much time and resources, only the best performing models

User Layer	Loss Function	RNN Units	MRR@10	Recall@10
yes	Cross Entropy	100	0.058	0.167
no	Cross Entropy	100	0.069	0.193
yes	TOP1	25	0.068	0.174
yes	TOP1	50	0.081	0.202
yes	TOP1	100	0.099	0.226
yes	TOP1	250	0.073	0.167
no	TOP1	25	0.073	0.180
no	TOP1	50	0.090	0.215
no	TOP1	100	0.112	0.258
no	TOP1	250	0.121	0.278

Table 5.1: Measurements on MIDI Dataset (One-Hot)

on the MIDI dataset were trained on the MAXI dataset. Finally the models that used the product embedding trained much faster, therefore there were multiple versions tested for the MAXI dataset.

For each model we measured two metrics of the validation set, since each of the metrics measures a different aspect of the performance of the model. Specifically Recall@10 was measured to determine in how many cases the Top 10 recommended products contained the product that was actually clicked by the user. MRR@10 was measured to determine where the product was ranked in the Top 10 predictions. A perfect model would achieve a value of 1 in both metrics, i.e. the next clicked product is always the top prediction. However while MRR@10 is the more informative metrics of the two, giving insight in the ranking of "good" predictions, Recall@10 is the more important one. Achieving a Recall@10 of 1 means the next clicked item is always in the Top 10 predictions, which means the user will see the recommendation, whether it is the top prediction or the fifth, which essentially is the goal of this task.

5.1.2 Measurements

The measurements on the MIDI dataset are summarized in table 5.1. As mentioned above these are all measurements from the models using the one-hot encoding as input. As can be seen the TOP1 loss introduced by the authors of [6] performs better than the cross entropy loss. This is because the cross entropy loss essentially compares the probabilities that are predicted by the model with the "real" probability distribution, in this case with the one-hot encoding of the label. In contrast the TOP1 loss is a strict ranking loss, only concerned with the position of the label in the predictions, which better models the task of recommendation. Further it can be seen that the

User Layer	RNN Units	MRR@10	Recall@10
yes	250	0.055	0.138
no	250	0.061	0.146

Table 5.2: Measurements on MAXI Dataset (One-Hot)

User Layer	RNN Units	MRR@10	Recall@10
no	25	0.012	0.025
no	50	0.011	0.022
no	100	0.012	0.025
no	250	diverged	diverged
yes	25	0.013	0.022
yes	50	0.011	0.023
yes	100	0.011	0.023
yes	250	diverged	diverged

Table 5.3: Measurements on MAXI Dataset (Embedding)

improvement of using 250 units per GRU instead of 100 is marginal, therefore larger models were not tested. Therefore the models using 250 units per GRU were chosen to be trained on the MAXI dataset in addition to the ones using the product embedding. Finally it can be seen that the models using the user level GRU perform approximately the same, sometimes even worse than the models using only the session level GRU, the reason for this will be discussed in the next chapter. The measurements for the models using the one-hot on the MAXI dataset are summarized in table 5.2. There is nothing surprising here, again we can see that the user-level GRU apparently does not help the performance of this task.

In the table 5.3 the measurements for the models using the product embedding on the MAXI dataset are summarized. As can be quickly seen the models do not perform nearly as well as the models using the one-hot encoding. Further the model does not improve when increasing the model size, which indicates that this is due to the product embeddings. Also this will be a topic explored in detail in the next chapter.

5.2 Online Experiment Setup

5.2.1 Experiment Setup

In this scenario we train the models on all the available data and evaluate the performance on live data generated by users that use the web application.

For this we chose the best performing configuration for each variant (c.f. 4.1) from the offline experiments. Using these four models an A/B/C/D/E test was setup. An A/B/n test is often used when testing features for online services. Such a test consists of choosing different versions of the same element or feature and then partition the traffic into groups, where each group is assigned to one version of the element. One of the versions should always serve as a control group, receiving the original version of the element. The user to group assignment is done at random to get evenly distributed groups. This group assigment is also persistent across different sessions. After such a test is setup, the test is run for some time while measuring KPIs per group. Then after ending the test the best performing version can be chosen based on the KPIs measured during the test.

The element that was tested in this work is the first slot for recommendations on the product detail page, the page that receives the most traffic. The slot is illustrated in figure 5.1. As a control version a black-box recommendation system provided by Google called Recommendations AI [1] was chosen. In the normal mode the chosen recommendation slot still has multiple recommendation engines which can be chosen according to a framework, however Recommendations AI is the best performing one. Other than the logic that chooses the products to display nothing was changed, i.e. the title and design remained identical. The test was run for 14 days to reflect one whole business cycle.

Something important to note at this stage is the postprocessing of recommendations coming either from the tested model or Recommendations AI. There are some constraints that are applied on the recommended product, such as filtering products that are inappropriate (e.g. alcoholic beverages, erotic articles etc.) and products that are not available for sale anymore (e.g. discontinued products, not on stock). This post processing is applied on all recommendation engines, therefore still providing a leveled playing field. However this is one of the reasons why production testing makes sense for recommendation systems, since this can never be reproduced in an offline setting, since some of these factors change over time.

5.2.2 Measurements

In table 5.4 we can see the measurements for the different variants of the model in the online setting, as well as the same metric for different models that have been displayed in the same slot as the one we tested in. We focus on the Click-Through rate in this experiment, since as mentioned before the conversion rate is difficult to compute, and due to time constraints we could not wait 14 days after the test was finished to get a good estimation

[1] https://cloud.google.com/recommendations/

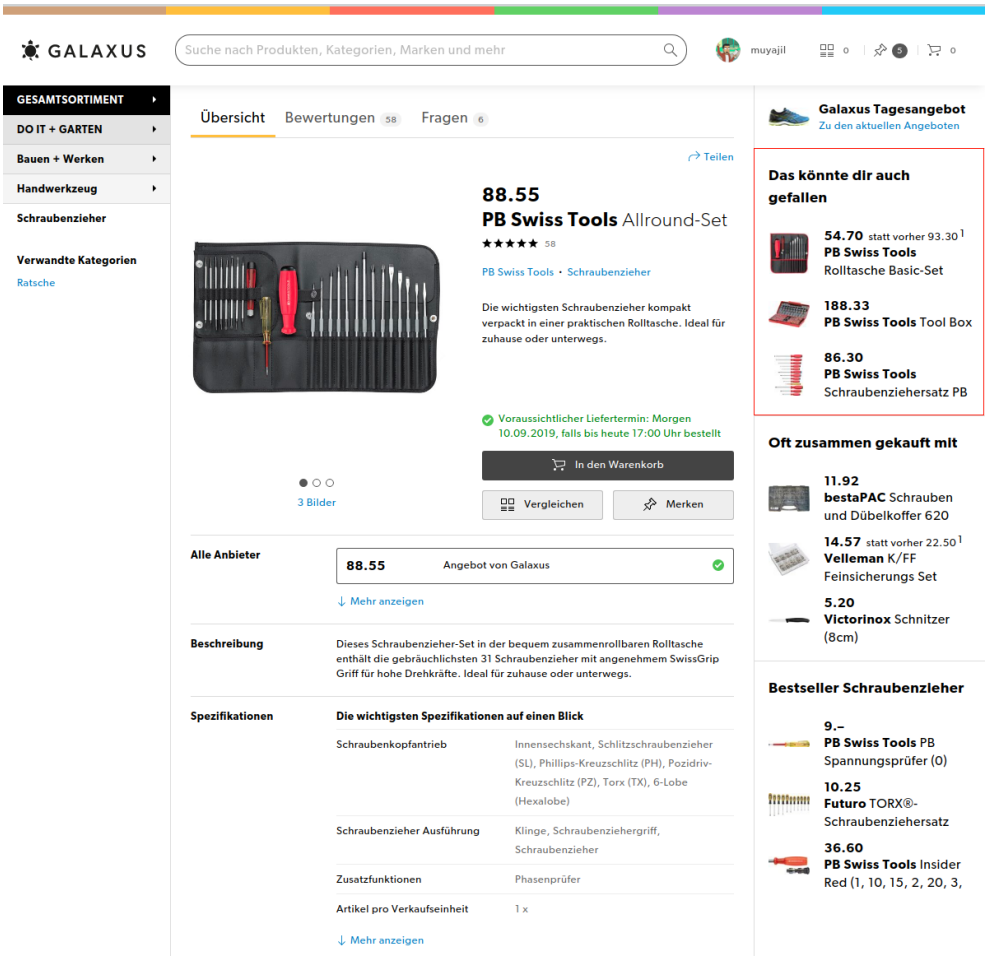

Figure 5.1: Recommendation slot on the product detail page

of this metric. The test was executed on approximately 1.3 Million sessions, each of the variants, as well as Recommendations AI received 20% of the total traffic coming to the product detail page. The metrics for each of the recommendation engines is averaged over all the sessions.

OftenBoughtTogether is a simple heuristic counting co-occurences of products in sales orders. Based on these counts products which are sold most together with the currently active item are recommended. It is very similar to the one described in section 2.1.3. Similar Products is a recommendation engine also using the product embeddings produced by Meta-Prod2Vec. This engine will recommend the items that are closest to the currently active item in the embedding space.

As can be seen in table 5.4 nothing comes close to achieving the same result as Recommendations AI. This is a proprietary system which is acessed di-

Model Name	CTR
Recommendations AI	5.55%
OnlySessionOneHot	2.1%
WithUserOneHot	2%
SimilarProducts	1.19%*
OftenBoughtTogether	0.91%*
WithUserEmbedding	0.9%
OnlySessionEmbedding	0.85%

Table 5.4: Measurements on Live Data (* Values are not based on the same test, but instead on more than one year of data.)

rectly via an API, however the data that is ingested by this system is very diverse. The system does not only consider the currently viewed item and the active user, but also takes into account trends, recent sales, product information and more. However when compared to more classical approaches the session-based approach performs much better, at least when using the one-hot encoding, since also in this setting we can see that the models using the product embedding perform much worse.

Chapter 6

Results

The goal of this thesis was to improve the session-based recommendation system using product embeddings. From the previous chapter it is clear that this approach did not work as intended. There are mainly two interesting things about the measurements in the experiments. First the user-layer does not seem to improve the performance, depending on the configuration even produces worse results than unpersonalized session-based recommendations. The authors of [11] found that the performance of the model depends strongly on the user history length. However in their case the user-level GRU still improves the performance, even for short user histories. The main difference between the datasets used in [11] and the datasets used in this work, is the distribution of the lengths of the sessions. The datasets in [11] in general have less sessions that are longer. If we look at the type of datasets used this makes sense. In [11] the authors used two datasets, one dataset comes from XING, an social network for professionals similar to LinkedIn, the second dataset is from a proprietary video platform similar to YouTube. The fundamental difference between the afromentioned dataset and the one used in this work is the possible entrypoints to the services. Both, XING and the video platform, offer a service which is sort of self-contained. On both services the user is served with everything that is needed in the same platform, searching for jobs, connecting with colleagues etc. can all be done in the same session. For an online-store this is a bit different. Many users use other entrypoints to the online-store, such as Google searches, Ads or price comparison pages. A bounce rate analysis shows, that 46% of users arriving on the product detail page, leave the site without accessing any other pages on the platform, i.e. the users use the online-store for researching product information, but not for browsing. Each of these visits still produced a session, this would explain the shorter sessions in our dataset. Of course it is clear that even when identifying the user, by looking at one product there is not much information to carry over to the next ses-

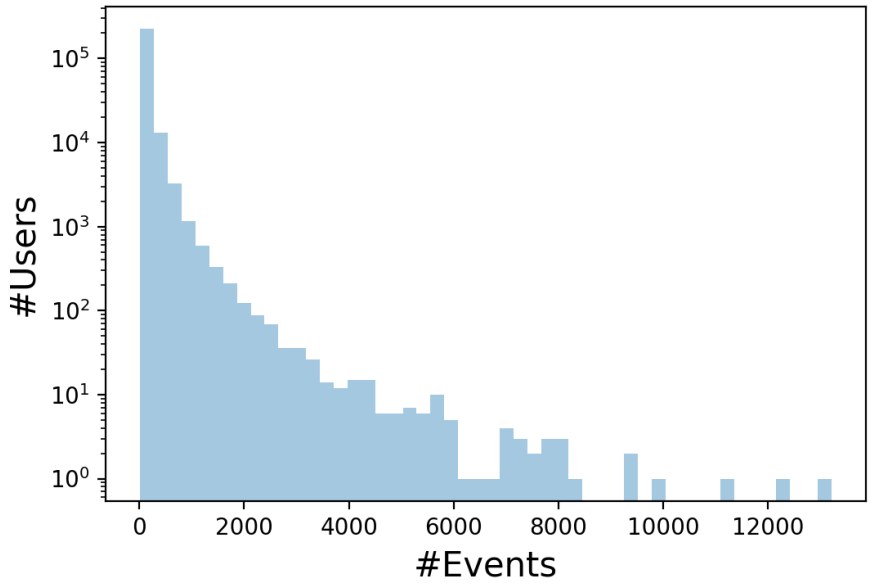

Figure 6.1: Distribution of the number of visits per user (log-scale)

sion. Another issue is that there are very many users which do have very few sessions. If we look at figure 6.1 we can see the distribution of the number of visits per user. What can be seen in the histogram is a phenomenon called *the Long Tail* in the context of web analytics. There is a very high proportion of the users which produce very little datapoints, at the same time there is a very long tail in the distribution, where some users produce extremely many events. Thus the information that can be extracted from the majority of users is rather limited. Of the approximately 250K users there are about 170K users that produced at most 100 events in total across all sessions. This would explain the limited amount of cross session information we can learn when adding the user-level GRU. Because of that the user embedding will not change much from its initial random intitialization, effectively changing nothing about the model, since without the user layer a session would be initialized at random as well.

The second thing is that the product embedding greatly worsens the results in the offline as well as the online setting. The reason for this is similar as explained above. In figure 6.2 we can see the same long tail effect as with the users. Of the approximately 470K products about 320K have at most 100 events recorded. In figure 6.3 we can see a PCA analysis of the product embeddings. As can be seen in the PCA there is a separation of datapoints in the first principal component. In figure 6.4 we can see the percentage of dat-

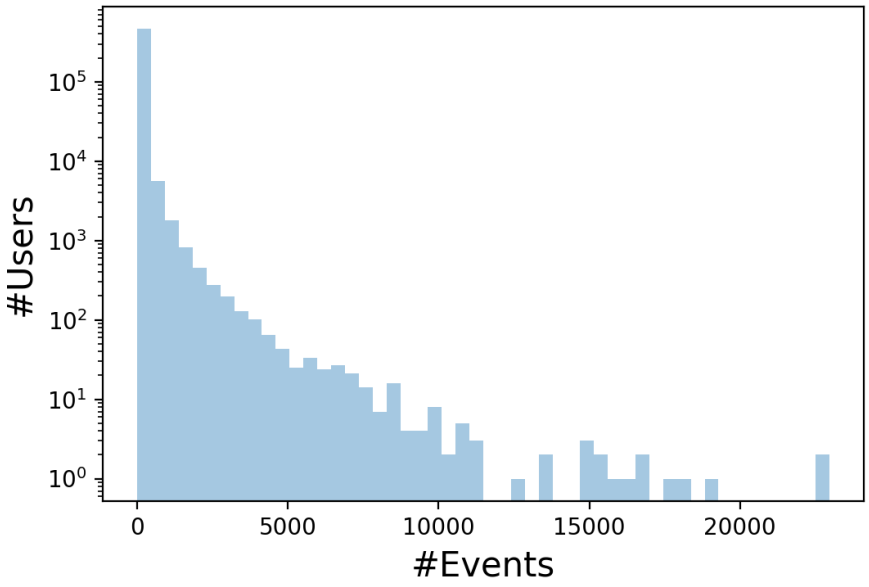

Figure 6.2: Distribution of the number of visits per product (log-scale)

apoints that have the first principal component smaller than 0, grouped by the number of events per product. It is clear from this that the products that generate a lot of events can clearly be separated from the ones producing few events. That means again that the majority of the product embeddings does not move much from the initial random initialization, since the majority of the products have less than 100 events. Therefore it can happen that for example a dining table with very few visits is very similar to a shoe with few visits, since both vectors are initialized uniformly at random drawing values from the same distribution. The problem with this is the signal that is propagated to the GRU. When using the one-hot encoding there is always a clear signal which product was the one that was viewed, with the product embedding this is not the case. Therefore the network in many cases, basically receives random noise as an input, not being able to produce something else in the output layer. This has the effect that the predictions in the models using the product embeddings are almost always the same, recommending items that are viewed many times, but do not have much relation to the currently active item.

In conclusion it can be said that the sesssion-based approach clearly works, even if it does not match the performance of a commercial product, it performs better than more classical approaches to recommendation. However personalizing these recommendations turns out to be more difficult than

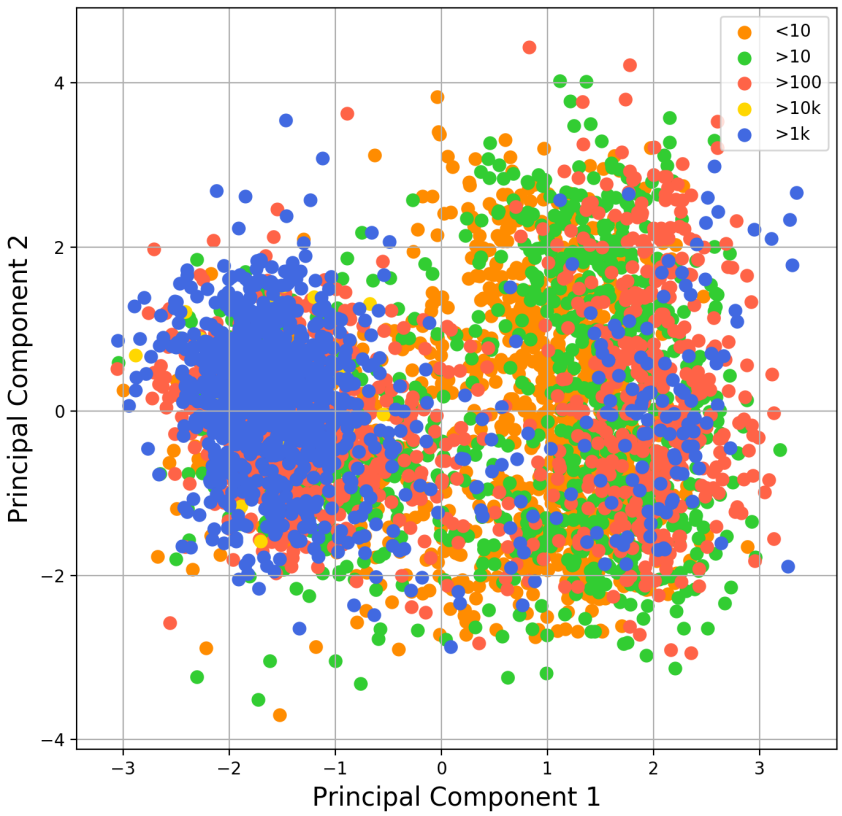

Figure 6.3: PCA of product embeddings

initially thought. It greatly depends on the number of events produced by each user. Therefore it might make sense to focus on personalizing the experience for the subset of users that actually produce enough events. The same can be said for the introduction of product embeddings, the product embeddings have to be very good such that the network can interpret the signal correctly. At least the properties of the dataset have a very large influence on the success of both the personalization of the recommendations as well as the usage of product embeddings.

A future improvement of this work would be the use of transfer learning. If the currently active item is an item with very few events, a similar item with more events can be used as a replacement as a basis for the recommendation. However this would require a secondary similarity measure. It would also be useful to improve the product embedding, in [10] a model is introduced

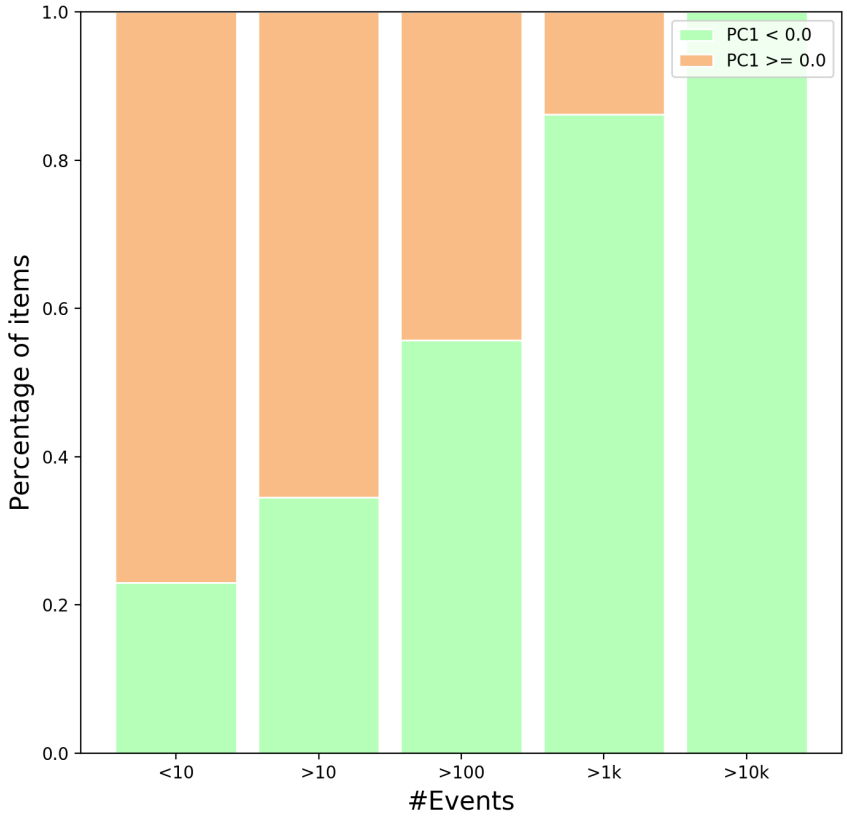

Figure 6.4: Value of PC1

which, in addition to meta information also takes into account images of a product and its textual description. Another approach to improving the product embedding could be to parametrize the distribution for the initialization of the vectors to some side-information. This would at least remove the issue that very dissimilar products have close embeddings. Finally another improvement that could be explored is to not only model the sequence of product detail pages, but all the different types of pages in such an online-store. This would give the system the ability to recommend review articles or categories for the user to explore.

Bibliography

[1] Christopher R. Aberger. Recommender : An analysis of collaborative filtering techniques. http://cs229.stanford.edu/proj2014/Christopher%20Aberger,%20Recommender.pdf, 2014.

[2] Erik Bernhardsson. Music discovery at Spotify. https://www.slideshare.net/erikbern/music-recommendations-mlconf-2014, 2014.

[3] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. https://arxiv.org/abs/1406.1078, 2014.

[4] Paul Covington, Jay Adams, and Emre Sargin. Deep Neural Networks for YouTube Recommendations. https://ai.google/research/pubs/pub45530, 2016.

[5] Mihajlo Grbovic, Vladan Radosavljevic, Nemanja Djuric, Narayan Bhamidipati, Jaikit Savla, Varun Bhagwan, and Doug Sharp. E-commerce in your Inbox: Product Recommendations at Scale. https://arxiv.org/abs/1606.07154, 2016.

[6] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. Session-based Recommendations with Recurrent Neural Networks. https://arxiv.org/abs/1511.06939, 2015.

[7] Zachary C. Lipton, John Berkowitz, and Charles Elkan. A Critical Review of Recurrent Neural Networks for Sequence Learning. https://arxiv.org/abs/1506.00019, 2015.

[8] Tomas Mikolov, Kai Chen, Greg Corrade, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. https://arxiv.org/abs/1301.3781, 2013.

[9] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. https://arxiv.org/abs/1310.4546, 2013.

[10] Thomas Nedelec, Elena Smirnova, and Flavian Vasile. Specializing Joint Representations for the task of Product Recommendation. https://arxiv.org/abs/1706.07625, 2017.

[11] Massimo Quadrana, Alexandros Karatzoglou, Balázs Hidasi, and Paolo Cremonesi. Personalizing Session-based Recommendations with Hierarchical Recurrent Neural Networks. http://arxiv.org/abs/1706.04148, 2017.

[12] Francesco Ricci, Lior Rokach, and Bracha Shapira. Introduction to Recommender Systems Handbook. http://www.inf.unibz.it/~ricci/papers/intro-rec-sys-handbook.pdf, 2011.

[13] Magnus Sahlgren. The distributional hypothesis. *Italian Journal of Disability Studies*, 20:33–53, 2008.

[14] Marcel Urech. Migros CIO Interview. https://www.netzwoche.ch/news/2017-06-29/martin-haas-ueber-cumulus-twint-und-die-migros-it, 2017.

[15] Flavian Vasile, Elena Smirnova, and Alexis Conneau. Meta-Prod2Vec - Product Embeddings Using Side-Information for Recommendation. http://arxiv.org/abs/1607.07326, 2016.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.