# Item Recommendation from Implicit Feedback

**Steffen Rendle**

## 1 Introduction

Item recommendation, also known as top-n recommendation, is the task to select the best items from a large catalogue of items to a user in a given context. For example, an e-commerce website recommends products to their customers, or a video platform might want to select videos that matches a user's interests. Item recommendation models are usually learned from implicit feedback. Instead of recommending based on explicit preferences such as *a user gave 4 stars to a movie*, the recommendations are based on the past interactions of the user with the system. For example, recommendation can be based on past products purchased by a customer, or videos watched by a user. A good recommender system is often contextual, considering the situation at which a recommendation should be made, e.g., what product is the user currently browsing on, or time information such as weekday vs weekend.

Item recommendation can be formulated as a context-aware ranking problem where the whole set of items should be ordered given a query context. In traditional collaborative filtering, the context would be the user, in more complex cases, the context might be user and time, a user and location, the sequence of previously selected items by a user, etc. To cover these cases, this chapter uses the general concept of a context as a placeholder. A scoring function that expresses the preference between a context and an item is used to order the items. This chapter covers learned scoring functions that are optimized from implicit feedback. The core challenges of learning from implicit feedback are to formulate an optimization objective and to design algorithms that can handle large item catalogues. First,

S. Rendle (✉)
Google Research, Mountain View, CA, USA
e-mail: srendle@google.com

F. Ricci et al. (eds.), *Recommender Systems Handbook*,
https://doi.org/10.1007/978-1-0716-2197-4_4

implicit feedback contains weak signals about a user's preference: the selected items in the past give a weak indicator of what a user likes. Implicit feedback usually does not contain negative interactions, instead weak negatives are derived from all the remaining items, i.e., the items that a user has not interacted with. Considering all remaining items as weak negative signal leads to the second challenge of high training costs. Formulating the item recommendation problem as a standard machine learning task is very costly because every positive signal from the implicit feedback entails negative signal over all items. That makes off-the-shelve training algorithms hard to apply to item recommendation models. Item recommenders are usually trained either (a) using sampling or (b) specialized algorithms taking into account model and loss structure.

This chapter starts with a detailed discussion of the item recommendation problem, covering the scoring function, the retrieval task, common evaluation metrics, and a summary of the core challenges in learning item recommendation from implicit feedback. The main part of this chapter focuses on the training problem. Sampling methods for general scoring functions based on pointwise, pairwise and softmax losses are described. The sampling distribution and weighting function are important for fast convergence and for aligning the algorithm better with the evaluation metrics. Section 5 discusses more efficient training algorithms that can be applied to dot product models and square losses. Finally, at application time, item recommenders need to be able to retrieve the highest scoring items quickly from the whole item catalogue. Section 6 discusses sublinear time approaches for the retrieval task.

## 2 Problem Definition

The goal of item recommendation is to retrieve a subset of interesting items for a given context $c \in C$ from a set of items $I$. This chapter uses the general concept of a context as a placeholder to cover many common item recommendation scenarios. In the simplest case, the context could be the user, in more complex cases, the context might be user and time, a user and location, the sequence of previously selected items by a user, etc. Also the input representation of a context and an item is flexible, it could be a user id or item id, but also more complex such as image pixels of an item, a textual description, etc. Such attributes are especially important in the cold start problem. The discussion in this chapter is independent of these choices and should be able to accommodate most item recommendation settings.

### *2.1 Recommender Modeling*

The preference of a context $c \in C$ to an item $i \in I$ is given by a scoring function

**Table 1** Frequently used notation

| Symbol | Description |
|---|---|
| $I$ | Set of items |
| $C$ | Set of context |
| $S \subseteq C \times I$ | Set of implicit observations |
| $I_c$ | Set of items selected in context $c$, i.e., $I_c := \{i : (c, i) \in S\}$ |
| $C_i$ | Set of context that selected item $i$, i.e., $C_i := \{c : (c, i) \in S\}$ |
| $\hat{y}(i\|c)$ | (Learnable) scoring function over context-item pairs |
| $\boldsymbol{\theta}$ | Model parameters of the scoring function $\hat{y}$ |
| $\boldsymbol{\phi}(c)$ | (Learnable) context representation/ embedding |
| $\boldsymbol{\psi}(i)$ | (Learnable) item representation/ embedding |
| $\langle \cdot, \cdot \rangle$ | Inner/ dot/ scalar product between two vectors or two matrices |
| $\otimes$ | Outer product between two vectors |
| $\delta(b)$ | Indicator function, 1 if $b$ is true, 0 otherwise |

$$\hat{y}(c, i) = \hat{y}(i|c), \quad \hat{y} : C \times I \to \mathbb{R} \tag{1}$$

The choice and design of the scoring function $\hat{y}$ is the problem of recommender system modeling. Most of the discussion here is independent of the particular choice of $\hat{y}$. Chapter "Advances in Collaborative Filtering" focuses on recommender system modeling and contains some examples of possible scoring functions.

The scoring function is parametrized[1] by a set of model parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. Learning the model parameters is a core problem within item recommendation and the main focus of this chapter. The first derivative of $\hat{y}(i|c)$ with respect to a parameter vector $\boldsymbol{\theta}$ is denoted as $\nabla_{\boldsymbol{\theta}} \hat{y}(i|c)$, and the second derivative as $\nabla^2_{\boldsymbol{\theta}} \hat{y}(i|c)$. Table 1 contains an overview of frequently used symbols.

### 2.1.1 Dot Product Models

While most of this chapter does not make any assumption about the scoring function, many recommender models have additional structure which is discussed now. For item recommendation, $\hat{y}$ is often decomposable into a dot product between a context and item representation

$$\hat{y}(i|c) = \langle \boldsymbol{\phi}(c), \boldsymbol{\psi}(i) \rangle \tag{2}$$

where context and items are represented by $d$ dimensional embeddings

$$\boldsymbol{\phi} : C \to \mathbb{R}^d, \quad \boldsymbol{\psi} : I \to \mathbb{R}^d. \tag{3}$$

[1] To be precise, $\hat{y}$ should be $\hat{y}_{\boldsymbol{\theta}}$, but for convenience, the subscript is omitted in this chapter.

Again, $\boldsymbol{\phi}$ and $\boldsymbol{\psi}$ are parametrized by $\boldsymbol{\theta}$. Such models are also known as two tower models, or dual encoders [30].

Unless stated otherwise, the methods described in this chapter will be general and are *not* restricted to dot product models. However, dot product models will be revisited frequently as they have very desirable properties for item recommendation.

### 2.1.2 Examples

Two common examples for scoring functions with a dot product are matrix factorization and two tower deep neural networks. First, the scoring function of matrix factorization is $\hat{y}(i|c) = \langle \mathbf{w}_c, \mathbf{h}_i \rangle$ where each context (or item) is directly represented by a $d$ dimensional embedding vector, $\mathbf{w}_c \in \mathbb{R}^d$ (or $\mathbf{h}_i \in \mathbb{R}^d$). These embedding matrices, $W \in \mathbb{R}^{C \times d}$ and $H \in \mathbb{R}^{I \times d}$ are the model parameters $\boldsymbol{\theta}$. A matrix factorization is a dot product model with $\boldsymbol{\phi}(c) = \mathbf{w}_c$ and $\boldsymbol{\psi}(i) = \mathbf{h}_i$.

Second, a general two tower deep neural network (DNN), $\hat{y}(i|c) = \langle \boldsymbol{\phi}(c), \boldsymbol{\psi}(i) \rangle$, where $\boldsymbol{\phi}(c)$ is a learnable function that extracts a $d$ dimensional representation for each context, and analogously $\boldsymbol{\psi}(i)$ is a learned representation for items. The model parameters of these functions are $\boldsymbol{\theta}$. For example $\boldsymbol{\psi}(i)$ could be a multi layer perceptron that generates an embedding based on the features of an item (e.g., genre, actors, director, release year of a movie)—here the model parameters $\boldsymbol{\theta}$ would be the weight matrices of the hidden layers. Other examples for $\boldsymbol{\psi}(i)$ or $\boldsymbol{\phi}(c)$ would be a convolutional network to extract spatial features from an image and map it to an embedding, or a recurrent model that builds an embedding representation based on sequential data such as previous purchases of a user or textual descriptions of an item.

## *2.2 Applying Item Recommenders*

The scoring function $\hat{y}$ can be used to rank items for a given context. Let $\hat{r}(i|c)$ be the rank/position of item $i$ in the sorted list of items $I$ given context $c$ and scoring function $\hat{y}$, i.e., $\hat{r} : I \rightarrow \{1, \ldots, |I|\}$. Here $r$ is a bijective mapping—in case of ties, there is some resolution, e.g., random. Thus, the inverse function gives the item ranked at a certain position, e.g, $\hat{r}^{-1}(3|c)$ would be the item ranked at the 3rd position for a context $c$.

When a recommendation model $\hat{y}$ is used for the task of item recommendation, it needs to find the $n$ highest scoring items for a given context $c$. That means it has to compute the items $\hat{r}^{-1}(1|c), \hat{r}^{-1}(2|c), \ldots, \hat{r}^{-1}(n|c)$. For example, a recommendation platform might want to show the user the 20 items with the highest score. Section 6 will discuss this problem of computing the top $n$ items in more detail.

For a good user experience there are other considerations besides high scores when selecting items. For example, diversity of results and slate optimization are important factors [11, 28]. A naive implementation just returns the top scoring items. While each item that is shown to a user might be individually a good choice, the combination of items might be suboptimal. For example, showing item $i$ might make item $j$ less attractive if $i$ and $j$ are very close or even interchangeable. Instead, it might be better to choose an item $l$ that complements $i$ better—even if $l$ has a lower score than $j$. Diversification of result sets is an example to avoid some of these effects. Commonly, algorithms for slate optimization and diversification are built on top of an item recommender. The remainder of this chapter will not discuss these important issues further, instead the chapter will focus on the top $n$ item recommendation task.

## 2.3 Evaluation

The quality of an item recommendation algorithm is typically measured by a ranking metric over the positions at which it ranks relevant items. For a given context $c$ and a ground truth set of relevant items $\{i_1, i_2, \ldots\}$, the ranks of these relevant items, $R := \{\hat{r}(i_1|c), \hat{r}(i_2|c), \ldots\}$, are computed and then a metric is applied. Several popular choices are discussed next. Chapter "Evaluating Recommender Systems" contains an exhaustive discussion about evaluation and more metrics.

*Precision at position n* measures the fraction of relevant items among the top $n$ predicted items:

$$\text{Prec}(R)_n = \frac{|\{r \in R : r \leq n\}|}{n}. \tag{4}$$

*Recall at position n* measures the fraction of all relevant items that were recovered in the top $n$:

$$\text{Recall}(R)_n = \frac{|\{r \in R : r \leq n\}|}{|R|}. \tag{5}$$

*Average Precision at n* measures the precision at all ranks that hold a relevant item:

$$\text{AP}(R)_n = \frac{1}{\min(|R|, n)} \sum_{i=1}^{n} \delta(i \in R)\text{Prec}(R)_i. \tag{6}$$

*Normalized discounted cumulative gain (NDCG) at n* places an inverse log reward on all positions that hold a relevant item:

$$\text{NDCG}(R)_n = \frac{1}{\sum_{i=1}^{\min(|R|,n)} \frac{1}{\log_2(i+1)}} \sum_{i=1}^{n} \delta(i \in R) \frac{1}{\log_2(i+1)}. \tag{7}$$

*Area under the ROC curve (AUC)* measures the likelihood that a random relevant item is ranked higher than a random irrelevant item.

$$\mathrm{AUC}(R) = \frac{1}{|R|(|I| - |R|)} \sum_{r \in R} \sum_{r' \in (\{1,\ldots,|I|\} \setminus R)} \delta(r < r') \tag{8}$$

These metrics differ in how much emphasize they place on particular ranks. Most metrics focus strongly on the top ranks and give almost no rewards for items that appear late in the result list. AUC is an exception as it has a slow (linear) decay. A more detailed discussion about these metrics can be found in [14].

The choice of the metric depends on the application. For retrieval tasks, head-heavy metrics such as Precision, Recall, AP or NDCG are better choices than AUC because users will likely investigate only the highest ranked items. If the retrieved results are directly shown, NDCG and AP can be better metrics than Precision and Recall because they emphasize the top ranked results stronger, e.g., distinguish between the first and second position. If the retrieval stage is followed by a reranking step, metrics such as Recall with $n$ equal to the number of candidates that will be reranked can be a better choice than metrics that care about the ranks within the top $n$ such as NDCG or AP.

## *2.4 Learning from Implicit Feedback*

This chapter covers implicit feedback between contexts and items. Let $S \subseteq C \times I$ be a set of such observations, where $(c, i) \in S$ is an implicit feedback pair. Examples for $S$ are clicks on links, video watches, purchases of items. Implicit observations provide a (weak) positive signal of a user's preferences. For example, if a user watches a video, it means that this video matches to the user's taste to some degree. Likewise, the items a user has never considered in the past are indicative of their taste as well. These non-selected items contain both the items that a user is not interested in and items that should be recommended to the user in the future. A core problem of learning item recommendation from implicit feedback is to define a training objective for trading off the selected with the non-selected items. Section 3 describes such strategies. Each of these strategies defines the loss over all selected and non-selected items, i.e., not just over $S$ but over the whole set $C \times I$. This leads to a second core challenge of deriving efficient learning algorithms.

This differentiates item recommendation from the learning to rank literature [5] where a smaller set of candidates needs to be reranked. This reranking of a small set of candidates has different problem characteristics than item recommendation because (1) it can be done directly on clickthrough data [12] avoiding the problem to define labels on unobserved data and (2) it is computationally simpler because it does not involve the full catalogue of all items. Such models serve a different purpose than the item recommendation models that are discussed in this chapter, and applying them directly to retrieval over the whole corpus is prone to low retrieval quality due to folding [29]. However, they are very useful as a second stage reranker on top of the retrieval models discussed in this chapter. For example, [6] describes such a two stage architecture where item recommendation is trained from implicit feedback and a second stage reranks the top $n$ retrieved items based on impression data. In this case, the primary goal of item recommendation is to achieve a high recall within the top $n$ results because the final ranking of these candidates is determined by the second stage ranker.

Another class of recommender system problems is based on explicit feedback, most notably *rating prediction*. Here, explicit preferences are provided such as *a user u assigns 5 stars to a movie i, but only 2 stars to movie j*. This is a different problem from item recommendation from implicit feedback. The training objective is simpler because it can be defined directly on the explicit feedback, avoiding the costly optimization over all items. These aspects makes it similar to the learning to rank problem, however it differs from learning to rank in the metrics. A commonality of item recommendation from implicit feedback and recommenders over explicit feedback, is that they share similar design patterns for the scoring function. For example, matrix factorization or item-based collaborative filtering is popular for both problems.

### *2.5 Core Challenges of Item Recommendation*

To summarize, the core challenges of item recommendation are:

1. Formulating a loss function over implicit feedback. This requires to define a strategy to trade off selected with unselected items while considering the ranking metric.
2. Learning algorithms that can handle the large number of unselected items efficiently.

## 3 Learning Objectives

This section describes three popular approaches for casting the item recommendation problem into a supervised machine learning problem.

### *3.1 Pointwise Loss*

The observations, $S$, can be directly casted into a binary classification problem by treating the observed feedback $S$ as positive instances and all non selected items,

$(C \times I) \setminus S$ as weak negatives. Each training instance has a positive or negative label and an example weight. Any positive pair $(c, i) \in S$, is assigned a positive label, e.g., $y = 1$ or, if available, another positive signal such as the strength of the interaction, e.g., number of times selected, or time spent. Additionally all non-observed items $j$ for the context are included with a negative label, e.g., $y = 0$ (for regression) or $y = -1$ (for classification). A weight $\alpha$ can be assigned to each context-item tuple to indicate the confidence about the training case. Usually, the confidence in the non-observed pairs is lower than for observed one. That balances the large amount of $O(|C \times I|)$ negatives with the smaller amount of $|S|$ positives. Other weighting schemes can be used to downweight popular items to promote more tail items in the result, or reduce the weight of users with many positive interactions so that the experience is not dominated by the behavior of heavy users.

From a regression point of view, a good scoring model predicts scores close to $y$. From a probabilistic point of view, the binary event $p(i = \text{true}|c)$ is modeled through $\hat{y}$, e.g., $p(i = \text{true}|c) = \sigma(\hat{y}(i|c))$. These ideas can be formalized as a pointwise loss

$$L(\boldsymbol{\theta}, S) := \sum_{c \in C} \sum_{i \in I} \alpha(c, i)\, l(\hat{y}(i|c), y(c, i)) + \lambda(\boldsymbol{\theta}) \tag{9}$$

where $\alpha : C \times I \to \mathbb{R}^+$ is a weight function, $y$ the label function (e.g., $y(c, i) = \delta((c, i) \in S)$), and $\lambda$ is some regularization function and $l$ is a loss function, e.g.,

$$l^{\text{square}}(\hat{y}, y) = (\hat{y} - y)^2 \tag{10a}$$

$$l^{\text{logistic}}(\hat{y}, y) = \ln(1 + \exp(-y\, \hat{y})) \tag{10b}$$

$$l^{\text{hinge}}(\hat{y}, y) = \max(0, 1 - y\, \hat{y}) \tag{10c}$$

This casts the problem of item recommendations to a standard binary classification or regression problem. What makes this problem special is that the number of training examples is huge $O(|C|\,|I|)$. The number of items can be in the millions for large scale problems which makes application of standard learning methods challenging.

Examples for recommender system algorithms that use a pointwise loss are iALS [10], SLIM [16], or iCD [2].

## 3.2 Pairwise Loss

For retrieval, it is not of primary interest if the score $\hat{y}$ is exactly 1 or 0. Instead the relative ordering of the items in a context is of central importance. Given a context $c$, pairwise methods compare all pairs of items $(i, j)$ where $(c, i) \in S$, $(c, j) \notin S$. A successful scoring function, $\hat{y}$, assigns a higher score to the observed item $i$ than the unobserved one: $\hat{y}(i|c) > \hat{y}(j|c)$. For the final retrieval, the items are simply sorted by the score.

A general pairwise loss function can be defined as:

$$L(\boldsymbol{\theta}, S) := \sum_{(c,i)\in S} \sum_{j\in I} \alpha(c, i, j)\, l(\hat{y}(i|c) - \hat{y}(j|c), 1) + \lambda(\boldsymbol{\theta}) \tag{11}$$

where $\alpha$ is a weight function. See Eq. (10) for some example losses $l$.

The number of pairs in this loss is $O(|S|\,|I|) \supseteq O(|C|\,|I|)$. That means it is at least as expensive as pointwise training, and it faces the same computational challenges.

## 3.3 *Softmax Loss*

From a probabilistic point of view, pointwise training models the binary event $p(i = \text{true}|c)$ through $\hat{y}$, e.g., $p(i = \text{true}|c) = \sigma(\hat{y}(i|c))$. Pairwise training models the conditional probability of the binary comparison $p(i > j|c)$ through $\hat{y}$, e.g., $p(i > j|c) = p(\hat{y}(i|c) > \hat{y}(j|c)) = \sigma(\hat{y}(i|c) - \hat{y}(j|c))$. A third option is to model the conditional multinomial event $p(i|c)$, i.e., $\sum_{j\in I} p(j|c) = 1$ and $\forall j : 0 \leq p(j|c) \leq 1$. The most common function to translate a real valued score, such as $\hat{y}$, to a multinomial distribution is the softmax:

$$p(i|c) = \frac{\exp(\nu\, \hat{y}(i|c))}{\sum_{j\in I} \exp(\nu\, \hat{y}(j|c))} \propto \exp(\nu\, \hat{y}(i|c)). \tag{12}$$

The numerator of the softmax is often referred to as the *partition function*. $\nu \in \mathbb{R}^+$ of the softmax is an (inverse) temperature parameter. Small choices of $\nu$ (=high temperature) make the distribution more uniform. Larger choices (=low temperature) make it more spiky and concentrated around the large values. For $\nu \to \infty$, softmax approaches the *maximum* operator. And for this limit the probability is concentrated at one index

$$p(i|c) = \delta\left(i = \operatorname*{argmax}_j \hat{y}(j|c)\right) \tag{13}$$

This aligns well with ranking metrics that focus on the top ranked positions. Softmax is widely used in multiclass classification problems, including natural language models where the classes are words, or image classification where the labels are categories.

To keep the notation short, the remainder of this chapter ignores $\nu$ because usually $\hat{y}$ can absorb any scaling effect of $\nu$ by increasing the norm of $\hat{y}$. From the softmax definition of the multinomial probability follows the loss function through the negative log likelihood:

$$\begin{aligned} L(\boldsymbol{\theta}, S) &= -\sum_{(c,i)\in S} \ln p(i|c) + \lambda(\boldsymbol{\theta}) \\ &= -\sum_{(c,i)\in S} \ln \frac{\exp(\hat{y}(i|c))}{\sum_{j\in I}\exp(\hat{y}(j|c))} + \lambda(\boldsymbol{\theta}) \\ &= -\sum_{(c,i)\in S} \left[ \hat{y}(i|c) - \ln \sum_{j\in I} \exp(\hat{y}(j|c)) \right] + \lambda(\boldsymbol{\theta}) \end{aligned} \tag{14}$$

Again, the loss contains a double sum over $S$ and $I$, so the number of elements is $\mathcal{O}(|S|\,|I|)$, or $\mathcal{O}(|C|\,|I|)$ if the partition function is computed once per context.

## 4 Sampling Based Learning Algorithms

Algorithms based on sampling are the most common approaches for learning recommender systems from implicit feedback. Their basic idea is to iterate over the positive examples $(c, i) \in S$, sample negatives $j$ with respect to a sampling distribution $q(j|c)$ and perform a weighted gradient step. This section discusses algorithms for pointwise, pairwise and softmax losses. The choice of $q$ together with a weighting function $\tilde{\alpha}$ are key for designing the learning algorithms. These two choices are related to $\alpha$ in the loss formulation. Many different instances of these algorithms have been proposed by varying $q$ and $\tilde{\alpha}$ and some popular choices are discussed in this section. Most of the results of this section apply to any scoring function, with the exception of Sects. 4.2.4 and 4.3.1 that are restricted to dot-product models.

### *4.1 Algorithms for Pointwise Loss*

The pointwise loss in Eq. (9) is defined over all context-item combinations, $C \times I$. The elements in this set are either a positive observation, if $(c, i) \in S$ or an implicit negative one otherwise. The positive observations are usually informative while the negative provide weaker feedback and are assigned a smaller weight so that they contribute less to the loss. A sampling based learning algorithm iterates over the positive observations and samples $m$ negatives. Algorithm 4.1 sketches this idea.

**Pointwise SGD**

```
1: repeat
2:     sample $(c, i) \in S$                                              ▷ positive item
3:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \left[\tilde{\alpha}(c, i) \nabla_{\boldsymbol{\theta}} l(\hat{y}(c, i), 1) + \nabla_{\boldsymbol{\theta}} \lambda(\boldsymbol{\theta})\right]$
4:     for $l \in \{1, \dots, m\}$ do                                      ▷ $m$ negative items
5:         sample $j$ from $q(j|c)$
6:         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \left[\tilde{\alpha}(c, j) \nabla_{\boldsymbol{\theta}} l(\hat{y}(c, j), 0) + \nabla_{\boldsymbol{\theta}} \lambda(\boldsymbol{\theta})\right]$
7:     end for
8: until converged
```

The main algorithm iterates over positive items as well as samples of negative items and performs an update step on each of them. The proportion of positive and negative items is controlled by $m$. It can help to sample more than one negative item per positive item because there are overall more negative items than positive ones and often each negative item is less informative than a positive one. Each update step moves the model parameters in the opposite direction of the gradient with respect to the loss. This process is known as stochastic gradient descent (SGD). The size of an update step is controlled by the learning rate $\eta \in \mathbb{R}^+$. Larger values of $\eta$ lead to faster progress but if the value is too large, the algorithm diverges. The iterative updates are repeated until some stopping criterion is met, e.g., a fixed number of repetitions, or the learning progress is less than a predefined threshold. The learning progress can be defined as the difference in a metric over two sufficiently spaced evaluations over the training set or a holdout set. It should be noted that due to the stochastic updates, the progress even on the training loss is noisy and not monotonically decreasing.

The main design choices of the algorithm are the sampling distribution $q$ and the weight $\tilde{\alpha}$. Their choice is related to $\alpha$ from the loss. If $q$ and $\tilde{\alpha}$ are independent of the model parameters $\boldsymbol{\theta}$, then the relationship between the three values is

$$\alpha(c, j) = \begin{cases} \tilde{\alpha}(c, j), & (c, j) \in S \\ m|I_c| q(j|c) \tilde{\alpha}(c, j), & (c, j) \notin S \end{cases} \tag{15}$$

where the sampling probability $q(j|c) = 0$ for $(c, j) \in S$. This shows how a weighted loss can be either achieved by changing $q$ or $\tilde{\alpha}$. For example, if the sampling probability $q$ is uniform

$$\alpha(c, j) = \begin{cases} \tilde{\alpha}(c, j), & (c, j) \in S \\ m\tilde{\alpha}(c, j), & (c, j) \notin S \end{cases} \tag{16}$$

A non-uniform sampling might be beneficial because most of the items are trivially negative, e.g., it is unlikely that a customer likes a random product from a large catalogue. Such samples are trivial for the model to predict and presenting an already correctly classified item during training has a (near) zero gradient, so overall convergence will be slow. Sampling items that have a higher chance to be considered by a user are more informative during learning.

Besides uniform sampling, popularity based sampling is another common approach. Here, the items are sampled proportional to their empirical frequency in the dataset $q(j|c) \propto |C_j|^\beta$ with a squashing exponent $\beta$ to desharpen the distribution. Section 4.2 describes some more sophisticated sampling algorithms.

### 4.1.1 Batching

The SGD algorithm presented so far performs one update step per context-item pair. Each step is computationally cheap. Modern hardware can benefit from larger units of computations, so grouping several training examples can result in more efficient utilization of hardware. For the proposed SGD algorithm, several positive and negative items can be sampled and grouped into a batch of training examples, and then their updates are performed in parallel. This scheme is especially useful if the same set of negatives is shared in the batch—a context independent sampler, e.g., frequency based, naturally fits here. Even more, instead of sampling from a global distribution, it is also common to use the positive items in the batch as the negatives. For example, if the batch contains positive observations $(c_1, i_1), (c_2, i_2), \ldots$, then $\{i_1, i_2, \ldots\}$ are treated as the negatives for this batch, e.g., for $c_1$, the negatives would be $(c_1, i_2), (c_1, i_3), \ldots$. In expectation, this corresponds to choosing the empirical frequency sampler for $q$. To summarize, batching does not improve the computational costs from a complexity perspective but it lowers the wall time on modern hardware.

### 4.1.2 Omitted Details

For simplicity, Algorithm 4.1 was sketched with a constant learning rate, $\eta$, but more complex schedules can be applied as well. Also depending on the form of $\hat{y}$, it is common to update only the subset of model parameters that affects $\hat{y}(i|c)$ for the particular choice of $(c, i)$. For example, in an embedding model, usually only a subset of the embeddings are involved in scoring a pair $(c, i)$, and it is common to apply the gradient step only to this subset of embeddings involved. These details are omitted because they are orthogonal to the problem of item recommendation from implicit feedback.

### 4.2 Algorithms for Pairwise Loss

The pairwise loss (Eq. 11) is a double sum over elements $(c, i) \in S$ and items $j \in I$. This scheme can be directly used for sampling and gradient steps can be performed for each item-pair $(i, j)$. Algorithm 4.2 shows a SGD algorithm for optimizing a pairwise loss.

**Pairwise SGD**

```
1: repeat
2:     sample $(c, i) \in S$
3:     sample $j$ from $q(j|c, i)$
4:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \left[\tilde{\alpha}(c, i, j) \nabla_{\boldsymbol{\theta}} l(\hat{y}(c, i) - \hat{y}(c, j), 1) + \nabla_{\boldsymbol{\theta}} \lambda(\boldsymbol{\theta})\right]$
5: until converged
```

The design choices of this algorithm are the sampling distribution of negatives, $q(j|c, i)$, and the weight of the gradient $\tilde{\alpha}(c, i, j)$. These two choices are related to the weight $\alpha(c, i, j)$ in the loss (Eq. (11)). Similar to the pointwise loss, if $q(j|c, i)$ and $\tilde{\alpha}(c, i, j)$ are independent of the model parameters then choosing $q$ and $\tilde{\alpha}$ implies a pairwise loss with $\alpha(c, i, j) = q(c, i, j)\, \tilde{\alpha}(c, i, j)$. Again, fixing two of the quantities will determine the third. For pairwise optimization, several variations with model dependent sampling and weighting have been proposed.

The next subsections discuss several popular variations of the algorithm. The first one samples items uniformly and has no weights. The second scheme adds a weight to consider head-heavy metrics. The third algorithm samples uniformly from items close or above the positive item, the sampling scheme is used to estimate the rank which is used as a weight. Finally, a sampler that oversamples items based on their rank is presented.

#### 4.2.1 Uniform Sampling Without Weight

The most simple instance of this algorithm uses a constant weight $\tilde{\alpha}(c, i, j) = 1$ and a uniform sampling probability $q(j|c) \propto 1$, which optimizes an unweighted loss with $\alpha(c, i, j) = 1$. This loss optimizes the area under the ROC curve, AUC, or equivalently, the likelihood that a random relevant item is ranked correctly above a random irrelevant item. This loss and algorithm is often referred to as *Bayesian Personalized Ranking (BPR)* [20].

There are two major issues with this approach:

1. Uniform sampling will sample irrelevant items that can be easily distinguished by the model and thus the gradient of such pairs becomes 0. The progress of the algorithm will be very slow because most of the pairs are not helpful.

2. Metrics that focus on the top items are often preferred for evaluating item recommendation. As discussed in Sect. 2.3, AUC is not a top-heavy metric. For example, for AUC, the benefit for moving a relevant item from position 1000 to 991 is the same as moving on item from position 10 to 1. For a metric such as NDCG, the second change is much more beneficial and gives a much larger improvement in the metric. In most applications, the second change would be considered more important.

Several improvements are discussed next.

### 4.2.2 Uniform Sampling with Weights

A popular approach in the learning to rank community is LambdaRank [5] which weights the gradient by its influence on a ranking metric. Let $M$ be a metric as defined in Sect. 2.3, then the weight of a pair of items $i, j$ given a context $c$ is $\tilde{\alpha}(c, i, j) = |M(\hat{r}(i|c)) - M(\hat{r}(j|c))|$. That means the weight is directly tied to the metric because it measures the influence of flipping the order of the two items. A practical difficulty of this approach is that the rank of the items $i$ and $j$ needs to be computed. For item recommendation with a large number of items this becomes very costly. This is less of a problem for learning to rank problems where only a small set of items are reranked. An adaption of LambdaRank to item recommendation was proposed in [32].

### 4.2.3 Adaptive Sampling with Weights

WARP (Weighted Approximate-Rank Pairwise) [27] is a pairwise algorithm specifically designed for item recommendation. The motivation of the WARP loss is to directly learn a ranking metric. Each rank is associated with a penalty, $\gamma$, where the better the predicted rank of the relevant item, the lower the penalty. This penalty can be chosen to align with the ranking metric [25]. For example, the penalty can be linearly increasing (as in AUC) or be a step function for a metric such as precision@k, or increase by the reciprocal of the position.

WARP uses an online algorithm for optimizing this loss. First, the sampling distribution is non-uniform. The algorithm tries to sample an irrelevant item $j$ such that it is ranked close to or above the relevant item, i.e., $\hat{y}(j|c) + 1 > \hat{y}(i|c)$. This choice is aligned with the hinge loss (Eq. (10c)), where the gradient is zero if the item is correctly ranked above the margin. The authors propose a rejection sampling algorithm that draws $j$ uniformly, and if it does not meet the condition, it discards $j$ and samples another item. This means the sampling distribution is $q(j|i, c) \propto \delta(\hat{y}(j|c) + 1 > \hat{y}(i|c))$.

Sampling an item $j$ that fulfills the requirement $\hat{y}(j|c) + 1 > \hat{y}(i|c)$ becomes increasingly hard, the better the model is and the more items are in the catalogue. The authors propose to stop the sampling algorithm after $|I|$ repetitions. The number of rounds until the algorithm terminates can be used to estimate the rank of the relevant item: $r(i|c) \approx \lfloor (\#\text{rounds} - 1)/|I| \rfloor$. The penalty, $\gamma$, of the estimated rank is used as the weight in the SGD algorithm:

$$\tilde{\alpha}(c, i, j) = \gamma(\lfloor (\#\text{rounds} - 1)/|I| \rfloor) \tag{17}$$

The definition of $q$ avoids the problem of sampling pairs that have no influence on the gradient step. However, while the algorithm avoids steps on non-influentual items, the cost for finding a relevant item are high because it needs to score all discarded items. Usually, scoring an item and updating an item have similar computational complexity. This gives WARP at most a constant speedup over a trivial method that does not discard items. However, sampling is only one part of the WARP algorithm and the more important part is that WARP uses the estimated rank in the weight for the gradient. This way, WARP optimizes a loss that is more reflective of ranking metrics than uniformly weighted methods such as BPR.

### 4.2.4 Adaptive Sampling Without Weights

As an improvement of the BPR algorithm, [19] propose to sample items based on their rank, here the sampling distribution is $q(j|c) \propto \exp(-r(j|c)/\gamma)$, while keeping the weight constant $\tilde{\alpha} = 1$. While LambdaRank and WARP introduce a weight on the rank by $\tilde{\alpha}$, here the weight is implicitly introduced through $q$ that depends on the rank. Note that this sampling distribution depends on the context and the current choice of model parameters. As the model learns, the sampling distribution adapts.

The novelty of this work is an algorithm to approximately sample from $q(j|c)$ in constant time. Unlike all the methods that have been discussed in this chapter so far and that were applicable to any scoring function, this work assumes a dot product model $\langle \boldsymbol{\phi}(c), \boldsymbol{\psi}(i) \rangle$. Next, the sampling algorithm is described briefly and more background can be found in [19].

For a given context, it first samples an embedding dimension $f^*$ according to

$$q(f|c) = |\phi_{c,f}|\sigma_f, \quad \sigma_f^2 = \text{Var}(\psi_{\cdot,f}) \tag{18}$$

The motivation for this choice is that for a given context embedding $\boldsymbol{\phi}(c)$ different embedding dimensions $f \in \{1, \dots, d\}$ are expected to have different contributions on the score and consequently on the rank. The larger an entry $|\phi_{c,f}|$ of the context embedding, the more influential is this embedding dimension. The sign is ignored at this step because without looking at the item embeddings it is meaningless. Each dimension is normalized by the variance of the item embeddings to ensure that dimensions with larger entries are overweighted in the sampling step.

Then an item is sampled according to the rank induced by $\boldsymbol{\psi}(i)_{\cdot,f^*}$. This is done by first sampling a desired rank $r^*$ from $\exp(-r/\gamma)$, this step is independent of the scoring function. Then the $r^*$-largest item from $\boldsymbol{\psi}(i)_{\cdot,f}$ is returned—or if the sign of $\phi_{c,f^*}$ is negative, the $r^*$-smallest (=most negative). This last step can be achieved by storing a sorted list of items for each embedding dimension. This sorted list is independent of the context and can be refreshed occasionally. In total, this is an amortized constant time algorithm for sampling an item approximately from $q(j|c) \propto \exp(-r(j|c)/\gamma)$.

### *4.3 Algorithms for Sampled Softmax*

A common strategy for training a model over a softmax loss is to sample $m$ negatives $\{j_1, \ldots, j_m\}$ from a distribution $q(j|c)$ and to compute the partition function on this smaller sample. A key difference of sampled softmax to negative sampling for pointwise and pairwise losses is that for softmax the sample is applied inside a logarithm (see Eq. (14)). That makes it difficult to get unbiased estimates. Sampled softmax is commonly used with a correction to the scores and for a sample $\{j_1, \ldots, j_m\}$ of $m$ negative items, and for an observation, $(c, i) \in S$, a gradient step with respect to the following loss is taken:

$$-\hat{y}(i|c) + \ln\left(\exp(\hat{y}(i|c)) + \sum_{l=1}^{m} \frac{\exp(\hat{y}(j_l|c))}{m\, q(j_l|c)}\right) \tag{19}$$

The correction $\frac{1}{m\,q(j|c)}$ ensures that if $m \to \infty$, then sampled softmax is unbiased [3]. However, typically a small (finite) set of negatives is sampled and thus sampled softmax is biased. This means no matter how many training epochs the algorithm is run, as long as $m$ is constant, sampled softmax will converge to a different solution than full softmax. It was shown that the only way to avoid this issue is to use $q(j|c) = p(j|c)$, i.e., to use the softmax distribution itself for sampling [4]. In this case, sampled softmax is unbiased for any sample size $m$. For sure, sampling from $p$ is expensive and not a viable option. Nevertheless, it shows that the bias can be reduced by either increasing $m$ and/or using a sampling distribution closer to $p(j|c)$.

In practice, when optimizing a recommender system with sampled softmax, the sample size $m$ is an important hyperparameter because it directly impacts what loss is optimized. Typically, a large sample size, $m$, is chosen to mitigate the bias. Again, the most common sampling distributions are (squashed) popularity (="unigram") sampling or in-batch sampling (=popularity sampling where $m$ = batchsize). Two more sophisticated sampling approaches that have been specifically designed for softmax are discussed in Sects. 4.3.1 and 4.3.2.

Algorithm 4.3 sketches pseudo code for optimizing a model with SGD for sampled softmax.

**Sampled Softmax SGD**

```
1: repeat
2:     sample $(c, i)$ from $S$
3:     sample $\{j_1, \dots, j_m\}$ from $q(j|c)$
4:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \left[ -\hat{y}(i|c) + \ln \left( \exp(\hat{y}(i|c)) + \sum_{l=1}^{m} \frac{\exp(\hat{y}(j_l|c))}{m\, q(j_l|c)} \right) + \lambda(\boldsymbol{\theta}) \right]$
5: until converged
```

### 4.3.1 Kernel Based Sampling

As discussed before, the sampling distribution is very important for sampled softmax optimization because it lowers the number of samples, $m$, required while keeping the bias low. The ideal sampling distribution would be $p(j|c, \boldsymbol{\theta}) \propto \exp(\hat{y}(j|c))$—or for dot product models $p(j|c, \boldsymbol{\theta}) \propto \exp(\langle \boldsymbol{\phi}(c), \boldsymbol{\psi}(j) \rangle)$. This formulation can be seen as a kernel and a mapping function $\pi : \mathbb{R}^d \to \mathbb{R}^D$ from the original embedding space to a larger space can be introduced, such that

$$p(j|c, \boldsymbol{\theta}) \propto \exp(\langle \boldsymbol{\phi}(c), \boldsymbol{\psi}(j) \rangle \approx \langle \pi(\boldsymbol{\phi}(c)), \pi(\boldsymbol{\psi}(j)) \rangle \propto q(j|c, \boldsymbol{\theta}) \tag{20}$$

Such a decomposition allows to compute the (approximated) partition function as:

$$\sum_{j \in I} \exp(\hat{y}(j|c)) \approx \sum_{j \in I} \langle \pi(\boldsymbol{\phi}(c)), \pi(\boldsymbol{\psi}(j)) \rangle = \left\langle \pi(\boldsymbol{\phi}(c)), \underbrace{\sum_{j \in I} \pi(\boldsymbol{\psi}(j))}_{\mathbf{z} \in \mathbb{R}^D} \right\rangle$$
$$= \langle \pi(\boldsymbol{\phi}(c)), \mathbf{z} \rangle$$

The key here is that $\mathbf{z}$ is independent of the context and can be precomputed. Then the sampling probability of an element j, $q(j|c)$, can be computed in $O(D)$ time. Based on this observation, a divide and conquer algorithm can be derived to sample j from $q(j|c, \boldsymbol{\theta})$ in $O(D \log_2 |I|)$ time (see [4] for details). Quadratic expansion [4] and random feature maps [17] have been explored for $\pi$.

### 4.3.2 Two-Pass Sampler

Another sampling strategy is a two stage sampler [1], where first a large set of $M > m$ items is sampled and only the $m$ most promising candidates are accepted. The first set is sampled from the squashed empirical frequency distribution, typically 1–10% of the items $I$ are sampled. Then all items on this large set are scored against a batch of context and a smaller set of the $m < M$ highest scoring items is returned. For efficient computation, it was proposed to distribute the computation of the scores of the $M$ items over several machines, preferably machines with GPUs. Finally, sampled softmax is applied on the smaller set. More details can be found in [1].

#### 4.3.3 Relation of Sampled Softmax to Pairwise Loss

Finally, a relationship between sampled softmax and a pairwise loss is highlighted. When the number of negative samples in sampled softmax is $m = 1$, then

$$
\begin{aligned}
&-\hat{y}(i|c) + \ln\left(\exp(\hat{y}(i|c)) + \frac{\exp(\hat{y}(j|c))}{q(j|c)}\right) = \ln \frac{\exp(\hat{y}(i|c)) + \frac{\exp(\hat{y}(j|c))}{q(j|c)}}{\exp(\hat{y}(i|c))} \\
&= l^{\text{logistic}}\left(\hat{y}(i|c) - \hat{y}(j|c) + \ln q(j|c), 1\right)
\end{aligned}
$$

Which is a pairwise logistic loss where the prediction of the negative item is shifted by the correction term $\ln q(j|c)$.

## 5 Efficient Learning Algorithms for Special Cases

This section investigates efficient learning algorithms for dot product models, i.e., $\hat{y}(i|c) = \langle \boldsymbol{\phi}(c), \boldsymbol{\psi}(i)\rangle$, and square losses. First efficient alternating least squares and gradient descent learning algorithms are presented for pointwise losses. Finally, their application to pairwise square losses is discussed briefly.

### *5.1 Pointwise Square Loss*

The following restrictions are made: (1) the weights and labels for all unobserved tuples $(c, i) \notin S$ is constant: let their weight be $\alpha_0$ and the label 0. It follows

$$
\begin{aligned}
L(\boldsymbol{\theta}, S) &= \sum_{c \in C} \sum_{i \in I} \alpha(c, i)(\hat{y}(i|c) - y(c, i))^2 + \lambda(\boldsymbol{\theta}) \\
&= \sum_{(c,i) \in S} [\alpha(c, i)(\hat{y}(i|c) - y(c, i))^2 - \alpha_0 \hat{y}(i|c)^2] \\
&\quad + \alpha_0 \sum_{c \in C} \sum_{i \in I} \hat{y}(i|c)^2 + \lambda(\boldsymbol{\theta})
\end{aligned}
$$

This can be further simplified to

$$\tilde{L}(\boldsymbol{\theta}, \tilde{S}) = \sum_{(c,i,\alpha,y)\in\tilde{S}} \alpha(\hat{y}(i|c) - y)^2 + \alpha_0 \sum_{c\in C}\sum_{i\in I} \hat{y}(i|c)^2 + \lambda(\boldsymbol{\theta}) \tag{21}$$

With $\tilde{S} = \{(c, i, \alpha(c, i) - \alpha_0, y\alpha(c, i)/(\alpha(c, i) - \alpha_0)) : (c, i) \in S\}$ and both $\tilde{L}$ and L share the same optimum (see [2] for more details). The remainder of this section uses this simplified formulation.

### 5.1.1 Gramian Trick

The first part of the loss $\tilde{L}$ depends only on the small set of observed positive data and is cheap to compute. The second part appears to be expensive with $|C||I|$ terms and a naive computation is prohibitively costly. However, the *Gramian trick* makes the computation of the second part very efficient [2]:

$$\begin{aligned}
\alpha_0 \sum_{c\in C}\sum_{i\in I} \hat{y}(i|c)^2 =& \alpha_0 \sum_{c\in C}\sum_{i\in I} \langle \boldsymbol{\phi}(c), \boldsymbol{\psi}(i)\rangle^2 && (22)\\
=& \alpha_0 \left\langle \sum_{c\in C} \boldsymbol{\phi}(c) \otimes \boldsymbol{\phi}(c), \sum_{i\in I} \boldsymbol{\psi}(i) \otimes \boldsymbol{\psi}(i) \right\rangle && (23)\\
=& \alpha_0 \left\langle G^C, G^I \right\rangle && (24)
\end{aligned}$$

with Gram matrices

$$G^C = \sum_{c\in C} \boldsymbol{\phi}(c) \otimes \boldsymbol{\phi}(c), \quad G^I = \sum_{i\in I} \boldsymbol{\psi}(i) \otimes \boldsymbol{\psi}(i) \tag{25}$$

The advantage of this *Gramian trick* is that the sum over all context-item combination is simplified as a the dot product over two Gram matrices, where each matrix has size $d \times d$. The cost for computing the Gramians[2] is $O(d^2(|I| + |C|))$. The cost for computing the loss, $\tilde{L}$, assuming the Gramians are known is $O(|S|\, d + d^2)$. The overall costs are dominated by the loss over the positive examples, $|S|$, as long as $d \leq |S|/(|C| + |I|)$. This means the loss over *all* examples (including the implicit negative ones) can be computed without paying the computational costs for the negative ones.

The remainder of this section derives efficient solvers for the loss with the Gramian formulation

[2] The analysis here ignores the cost for computing the embeddings $\boldsymbol{\phi}(c)$ and $\boldsymbol{\psi}(i)$. The derived results have a linear complexity in the costs for computing the embeddings.

$$\tilde{L}(\boldsymbol{\theta}, \tilde{S}) = \sum_{(c,i,\alpha,y)\in\tilde{S}} \alpha(\hat{y}(i|c) - y)^2 + \alpha_0 \left\langle G^C, G^I \right\rangle + \lambda(\boldsymbol{\theta}) \tag{26}$$

The Gramian trick is related to the Kernel softmax (see Sect. 4.3.1) where $\exp(\langle \boldsymbol{\phi}(c), \boldsymbol{\psi}(i)\rangle) \approx \langle \pi(\boldsymbol{\phi}(c)), \pi(\boldsymbol{\psi}(i))\rangle$. In particular, using the square function instead of the exp, $\langle \boldsymbol{\phi}(c), \boldsymbol{\psi}(i)\rangle^2 = \langle \pi(\boldsymbol{\phi}(c)), \pi(\boldsymbol{\psi}(i))\rangle$ for $\pi(\mathbf{x}) = \mathbf{x} \otimes \mathbf{x}$.

### 5.1.2 Coordinate Descent/ALS Solver for Multilinear Models

The Gramian trick can be used to derive efficient alternating least squares solvers for the family of multilinear models. This section starts with a recap of multilinear models and then derives a learning algorithm. Besides multilinearity, it is assumed that the regularization $\lambda(\boldsymbol{\theta})$ is a L2 regularization, i.e., $\lambda(\boldsymbol{\theta}) = \lambda||\boldsymbol{\theta}||^2$.

#### Multilinear Model

Following [2], the derivation in this subsection makes the assumption that $\boldsymbol{\phi}(c)$ and $\boldsymbol{\psi}(i)$ do not share model parameters. In particular for every scalar coordinate $\tilde{\theta}$ from the vector $\boldsymbol{\theta}$ of model parameters

$$\forall c \in C : \nabla_{\tilde{\theta}} \boldsymbol{\phi}(c) = 0 \quad \text{or} \quad \forall i \in I : \nabla_{\tilde{\theta}} \boldsymbol{\psi}(i) = 0 \tag{27}$$

To simplify notation, it is further assumed that the model is multilinear in the model parameters. That means for each model parameter $\tilde{\theta}$ the model can be written in a linear form

$$\hat{y}(i|c) = g(i|c) + \tilde{\theta}\, \nabla_{\tilde{\theta}} \hat{y}(i|c), \quad \nabla^2_{\tilde{\theta}} \hat{y}(i|c) = 0 \tag{28}$$

Often, models are not just linear in scalars but linear in subvectors $\tilde{\boldsymbol{\theta}}$ of the model parameters $\boldsymbol{\theta}$

$$\hat{y}(i|c) = g(i|c) + \langle \tilde{\boldsymbol{\theta}}, \nabla_{\tilde{\boldsymbol{\theta}}} \hat{y}(i|c)\rangle, \quad \nabla^2_{\tilde{\boldsymbol{\theta}}} \hat{y}(i|c) = 0 \tag{29}$$

For example, matrix factorization: $\hat{y}(i|c) = \langle \mathbf{w}_u, \mathbf{h}_i\rangle$ is linear in $\mathbf{w}_u$ or $\mathbf{h}_i$. Some other examples for multilinear models are factorization machines [18], or tensor factorization models such as PARAFAC [8] and Tucker Decomposition [24].

#### Optimization

With these prerequisites, efficient alternating least squares solvers can be derived. Let $\tilde{\boldsymbol{\theta}}_1, \tilde{\boldsymbol{\theta}}_2, \ldots$ be a partition of the model parameters $\boldsymbol{\theta}$ such that the model is linear in each $\tilde{\boldsymbol{\theta}}$. From this follows that the optimal values for $\tilde{\boldsymbol{\theta}}$ have a closed form solution that can be obtained by a linear regression solver. An efficient computation of the least square solution needs to consider the Gramian trick. This will be discussed next. The solution can be derived by finding the root of $\nabla_{\tilde{\boldsymbol{\theta}}} L(\boldsymbol{\theta}, S)$. A single iteration of Newton's method finds its solution:

$$\tilde{\boldsymbol{\theta}}^* = \tilde{\boldsymbol{\theta}} - (\nabla^2_{\tilde{\boldsymbol{\theta}}} \tilde{L}(\boldsymbol{\theta}, \tilde{S}))^{-1} \nabla_{\tilde{\boldsymbol{\theta}}} \tilde{L}(\boldsymbol{\theta}, \tilde{S})$$

The computation of the sufficient statistics $\nabla^2_{\tilde{\boldsymbol{\theta}}} \tilde{L}(\boldsymbol{\theta}, \tilde{S})$ and $\nabla_{\tilde{\boldsymbol{\theta}}} \tilde{L}(\boldsymbol{\theta}, \tilde{S})$ is now discussed for a vector of model parameters $\tilde{\boldsymbol{\theta}}$ from the context side. The equations for model parameters from the item side are analogously.

$$\nabla_{\tilde{\boldsymbol{\theta}}} \tilde{L}(\boldsymbol{\theta}, \tilde{S}) \overset{(*)}{=} \sum_{(c,i,\alpha,y)\in\tilde{S}} \alpha(\hat{y}(i|c) - y)\nabla_{\tilde{\boldsymbol{\theta}}} \hat{y}(i|c) + \alpha_0 \sum_{c\in C} \boldsymbol{\phi}(c) G^I (\nabla_{\tilde{\boldsymbol{\theta}}} \boldsymbol{\phi}(c))^t + \lambda\tilde{\boldsymbol{\theta}}$$

$$\nabla^2_{\tilde{\boldsymbol{\theta}}} \tilde{L}(\boldsymbol{\theta}, \tilde{S}) \overset{(**)}{=} \sum_{(c,i,\alpha,y)\in\tilde{S}} \alpha \nabla_{\tilde{\boldsymbol{\theta}}} \boldsymbol{\phi}(c) \otimes \nabla_{\tilde{\boldsymbol{\theta}}} \boldsymbol{\phi}(c) + \alpha_0 \sum_{c\in C} (\nabla_{\tilde{\boldsymbol{\theta}}} \boldsymbol{\phi}(c)) G^I (\nabla_{\tilde{\boldsymbol{\theta}}} \boldsymbol{\phi}(c))^t + \lambda I$$

where (*) uses $\nabla_{\tilde{\boldsymbol{\theta}}} \hat{y}(i|c) = \langle \nabla_{\tilde{\boldsymbol{\theta}}} \boldsymbol{\phi}(c), \boldsymbol{\psi}(i)\rangle$ and (**) uses of $\nabla^2_{\tilde{\boldsymbol{\theta}}} \hat{y}(i|c) = 0$.

From these equations follows the generic ALS algorithm from implicit data. The algorithm iterates over parameters from the context side, computes the first and second derivative as outlined above and performs the Newton update step. The same steps are repeated for parameters from the item side.

**iALS-Pointwise for Multilinear Models**

```
1: repeat
2:     $G^I \leftarrow \sum_{i\in I} \boldsymbol{\psi}(i) \otimes \boldsymbol{\psi}(i)$
3:     for $\tilde{\boldsymbol{\theta}} \in \{\tilde{\boldsymbol{\theta}}_1, \tilde{\boldsymbol{\theta}}_2, \ldots\}$ do        ▷ Iterate over parameters of the context side
4:         $\nabla_{\tilde{\boldsymbol{\theta}}} \leftarrow \lambda\tilde{\boldsymbol{\theta}}$
5:         $\nabla^2_{\tilde{\boldsymbol{\theta}}} \leftarrow \lambda I$
6:         for $c \in C$ where $\nabla_{\tilde{\boldsymbol{\theta}}} \boldsymbol{\phi}(c) \neq 0$ do
7:             $\nabla_{\tilde{\boldsymbol{\theta}}} \leftarrow \nabla_{\tilde{\boldsymbol{\theta}}} + \alpha_0 \boldsymbol{\phi}(c) G^I (\nabla_{\tilde{\boldsymbol{\theta}}} \boldsymbol{\phi}(c))^t$
8:             $\nabla^2_{\tilde{\boldsymbol{\theta}}} \leftarrow \nabla^2_{\tilde{\boldsymbol{\theta}}} + \alpha_0 (\nabla_{\tilde{\boldsymbol{\theta}}} \boldsymbol{\phi}(c)) G^I (\nabla_{\tilde{\boldsymbol{\theta}}} \boldsymbol{\phi}(c))^t$
9:         end for
10:        for $(c, i, \alpha, y) \in \tilde{S}$ where $\nabla_{\tilde{\boldsymbol{\theta}}} \boldsymbol{\phi}(c) \neq 0$ do
11:            $\nabla_{\tilde{\boldsymbol{\theta}}} \leftarrow \nabla_{\tilde{\boldsymbol{\theta}}} + \alpha(\hat{y}(i|c) - y)\langle \nabla_{\tilde{\boldsymbol{\theta}}} \boldsymbol{\phi}(c), \boldsymbol{\psi}(i)\rangle$
12:            $\nabla^2_{\tilde{\boldsymbol{\theta}}} \leftarrow \nabla^2_{\tilde{\boldsymbol{\theta}}} + \alpha \nabla_{\tilde{\boldsymbol{\theta}}} \boldsymbol{\phi}(c) \otimes \nabla_{\tilde{\boldsymbol{\theta}}} \boldsymbol{\phi}(c)$
13:        end for
14:        $\tilde{\boldsymbol{\theta}} \leftarrow \tilde{\boldsymbol{\theta}} - (\nabla^2_{\tilde{\boldsymbol{\theta}}})^{-1} \nabla_{\tilde{\boldsymbol{\theta}}}$
15:    end for
16:    Perform a similar pass over parameters from the item side
17: until converged
```

If $\hat{y}$ is a matrix factorization model, and if for $\tilde{\boldsymbol{\theta}}$ a user embedding vector, $\mathbf{w}_u$ or item embedding vector $\mathbf{h}_i$ is chosen, then this is equivalent to the iALS algorithm proposed in [10]. This algorithm has a complexity of $\mathcal{O}(|S|\,d^2 + (|C| + |I|)\,d^3)$ per epoch. This is much more efficient than the complexity of $\mathcal{O}(|C|\,|I|\,d^2 + (|C| + |I|)\,d^3)$ of a naive ALS implementation. The work of [10] was the first to provide this efficient training algorithm for learning matrix factorization from implicit feedback. Later, [9] derived a variation of this algorithm for PARAFAC tensor factorization.

The derivation in this chapter applies to any multilinear model including matrix factorization, PARAFAC, Tucker Decomposition or factorization machines. The coordinate descent version (i.e., choosing a scalar for $\boldsymbol{\theta}$) of this generalized algorithm was proposed in [2]. The alternating least squares version in this chapter is more general because any vector $\boldsymbol{\theta}$ can be chosen, including a vector of size one (i.e., a scalar) which would reduce to a coordinate descent algorithm. Coordinate descent (CD) algorithms have a lower computation complexity than ALS solvers, e.g., $\mathcal{O}(|S|\,d + (|C| + |I|)\,d^2)$ for matrix factorization (see [2] for details). However, on modern hardware, vector operations as used by a ALS can be much more efficient than the scalar operations of CD, and with a careful implementations the higher theoretical complexity is not noticable. Moreover, using vectors instead of scalars reduces synchronization points which is very important in distributed algorithms, so ALS methods can be overall more efficient in wall time than their CD equivalents.

### 5.1.3 SGD Solver for General Models

The previous section discussed multilinear models. Now, following [13], this is generalized to any dot-product model $\hat{y}(i|c) = \langle \boldsymbol{\phi}(c), \boldsymbol{\psi}(i) \rangle$. In particular, $\boldsymbol{\phi}(c)$ and $\boldsymbol{\psi}(i)$ can be any structure, including non-linear DNNs. Such structures are commonly optimized by SGD algorithms that iterate over observed examples $S$.

Now, the loss in Eq. (26) is reformulated as a sum over training examples

$$\tilde{L}(\boldsymbol{\theta}, \tilde{S}) = \sum_{(c,i,\alpha,y)\in\tilde{S}} l(c, i, \alpha, y) \tag{30}$$

so that stochastic gradient descent can be applied. To achieve this form, the Gramian term in the loss is rewritten as a sum over training examples:

$$\left\langle G^C, G^I \right\rangle = \frac{1}{2}\left(\sum_{c\in C} \boldsymbol{\phi}(c) G^I \boldsymbol{\phi}(c)^t + \sum_{i\in I} \boldsymbol{\psi}(i) G^C \boldsymbol{\psi}(i)^t\right) \tag{31}$$

$$= \sum_{(c,i,\alpha,y)\in\tilde{S}} \frac{1}{2}\left(\frac{1}{|I_c|}\boldsymbol{\phi}(c) G^I \boldsymbol{\phi}(c)^t + \frac{1}{|C_i|}\boldsymbol{\psi}(i) G^C \boldsymbol{\psi}(i)^t\right) \tag{32}$$

Combining this with the loss on the positive observations, results in the final form of the elementwise loss

$$l(c, i, \alpha, y) = \alpha(\hat{y}(i|c) - y)^2 + \frac{\alpha_0}{2}\left(\frac{1}{|I_c|}\boldsymbol{\phi}(c)G^I\boldsymbol{\phi}(c)^t + \frac{1}{|C_i|}\boldsymbol{\psi}(i)G^C\boldsymbol{\psi}(i)^t\right) \tag{33}$$

Now, gradient descent can be applied to this equation. For the purpose of learning $\boldsymbol{\theta}$, $G^C$ and $G^I$ are treated as constants and replaced by estimates $\hat{G}^C$ and $\hat{G}^I$

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} l(c, i, \alpha, y) &= 2\alpha(\hat{y}(i|c) - y)\nabla_{\boldsymbol{\theta}}\hat{y}(i|c)\\ &+ \alpha_0\left(\frac{1}{|I_c|}\boldsymbol{\phi}(c)\hat{G}^I(\nabla_{\boldsymbol{\theta}}\boldsymbol{\phi}(c))^t + \frac{1}{|C_i|}\boldsymbol{\psi}(i)\hat{G}^C(\nabla_{\boldsymbol{\theta}}\boldsymbol{\psi}(i))^t\right)\end{aligned} \tag{34}$$

The Gramian estimates are updated by gradient descent as well. A good Gramian estimate, $\hat{G}$, is close to the true Gramian, $G$. A reasonable objective to enforce closeness is the Frobenius norm $||\hat{G} - G||_F^2$. To make this objective amendable to SGD, the Gramian loss is reformulated as a sum over positive training examples

$$\underset{\hat{G}^C}{\operatorname{argmin}} \left\|\hat{G}^C - G^C\right\|_F^2 = \underset{\hat{G}^C}{\operatorname{argmin}} \left\|\hat{G}^C - \sum_{c \in C}\boldsymbol{\phi}(c) \otimes \boldsymbol{\phi}(c)\right\|_F^2 \tag{35}$$

$$= \underset{\hat{G}^C}{\operatorname{argmin}} \sum_{c \in C}\left\|\hat{G}^C - |C|\boldsymbol{\phi}(c) \otimes \boldsymbol{\phi}(c)\right\|_F^2 \tag{36}$$

$$= \underset{\hat{G}^C}{\operatorname{argmin}} \sum_{(c,i,\alpha,y)\in\tilde{S}} \frac{1}{|I_c|}\left\|\hat{G}^C - |C|\boldsymbol{\phi}(c) \otimes \boldsymbol{\phi}(c)\right\|_F^2 \tag{37}$$

which results in the gradient of the Gramian estimate:

$$\nabla_{\hat{G}^C}\left[\sum_{(c,i,\alpha,y)\in\tilde{S}} \frac{1}{|I_c|}\left\|\hat{G}^C - |C|\boldsymbol{\phi}(c) \otimes \boldsymbol{\phi}(c)\right\|_F^2\right] \tag{38}$$

$$= \sum_{(c,i,\alpha,y)\in\tilde{S}} \frac{1}{|I_c|}\left(\hat{G}^C - |C|\boldsymbol{\phi}(c) \otimes \boldsymbol{\phi}(c)\right) \tag{39}$$

To summarize, when sampling a training example $(c, i, \alpha, y) \in \tilde{S}$, the update rule for the Gramian estimate is

$$\hat{G}^C \leftarrow \hat{G}^C - \eta\frac{1}{|I_c|}\left(\hat{G}^C - |C|\boldsymbol{\phi}(c) \otimes \boldsymbol{\phi}(c)\right) \tag{40}$$

$$= \hat{G}^C \left(1 - \frac{\eta}{|I_c|}\right) + \frac{\eta}{|I_c|} |C| \boldsymbol{\phi}(c) \otimes \boldsymbol{\phi}(c) \tag{41}$$

See [13] for other estimation algorithms.[3]

Algorithm 5.1.3 shows the final SGD procedure.

**SGD-Pointwise with Gramian Trick**

```
1: repeat
2:     sample $(c, i) \in S$
3:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} l(c, i, \alpha, y)$        ▷ See Eq. (34)
4:     sample $(c, i) \in S$
5:     $\hat{G}^C \leftarrow \hat{G}^C - \eta \frac{1}{|I_c|} \left(\hat{G}^C - |C| \boldsymbol{\phi}(c) \otimes \boldsymbol{\phi}(c)\right)$
6:     $\hat{G}^I \leftarrow \hat{G}^I - \eta \frac{1}{|C_i|} \left(\hat{G}^I - |I| \boldsymbol{\psi}(i) \otimes \boldsymbol{\psi}(i)\right)$
7: until converged
```

Comparing this to the vanilla SGD Algorithm 4.1, it can be seen that there is no negative sampling step necessary when using the Gramian trick. All negatives are already considered in the update step of the positive item through the Gramian (see line 3). In addition, there is the estimation step of the Gramians.

In some sense, negative sampling can be seen as an estimation of Gramians as well: instead of keeping track of an estimate of the Gramian as proposed here, negative sampling rebuilds the Gramian based on only a small sample of $m$ negatives. This is a much less precise estimate of the Gramian than keeping a long-term estimate because the global Gramians are unlikely to change considerably for each step.

## 5.2 *Pairwise Square Loss*

Finally, efficient solvers for pairwise square losses are shortly discussed. For keeping the derivation simple, it is assumed that $\alpha(c, i, j) = 1$; a generalization to $\alpha(c, i, j) = \alpha_{c,i} \alpha_j$ is simple and follows the same pattern. The pairwise square loss is defined as

[3] [13] uses a different weight on each Gramian element which makes the derivation more natural for SGD algorithms. This chapter uses the same Gramian definition as in Eq. (25) to make all results consistent.

$$L(\boldsymbol{\theta}, S) = \sum_{(c,i)\in S} \sum_{j \in I} [\hat{y}(i|c) - \hat{y}(j|c) - 1]^2. \tag{42}$$

This can be reformulated to:

$$L(\boldsymbol{\theta}, S) = \sum_{(c,i)\in S} \sum_{j \in I} [(\hat{y}(i|c) - 1)^2 - 2(\hat{y}(i|c) - 1)\hat{y}(j|c) + \hat{y}(j|c)^2] \tag{43}$$

$$= \sum_{(c,i)\in S} |I|(\hat{y}(i|c) - 1)^2 + \langle G^C, G^I \rangle - 2 \sum_{(c,i)\in S} (\hat{y}(i|c) - 1)\langle \boldsymbol{\phi}(c), \mathbf{z} \rangle \tag{44}$$

with

$$G^C := \sum_{c \in C} |I_c| \boldsymbol{\phi}(c) \otimes \boldsymbol{\phi}(c), \quad G^I := \sum_{i \in I} \boldsymbol{\psi}(i) \otimes \boldsymbol{\psi}(i), \quad \mathbf{z} := \sum_{i \in I} \boldsymbol{\psi}(i) \tag{45}$$

The first two terms in the loss are identical to the pointwise square loss in Eq. (26): a pointwise loss over the positive observations, and the Gramian term. The last term is an additional correction term for pairwise square loss, and its computation is in $\mathcal{O}(|S|\,d)$. The algorithms (both iALS and SGD) introduced for pointwise loss can be extended to take the new term into account. Instead of just storing the Gramian $G^I$, the vector $\mathbf{z}$ has to be stored as well.

Takács and Tikk [23] first introduced an efficient solver for matrix factorization and pairwise square loss. Their derivation is based on a gradient reformulation as previously invented for pointwise square loss by Hu et al. [10]. The derivation in this chapter is more general and makes pairwise square loss applicable to a wider variety of models (multilinear models and general dot-product models) and optimization schemes (ALS, CD, SGD).

## 6 Retrieval with Item Recommenders

The typical application of an item recommender has to return the highest scoring items given a context $c$. This section will first highlight the limitations of a naive brute-force implementation and then discuss efficient algorithms for dot product models.

### *6.1 Limitations of Brute-Force Retrieval*

A naive implementation scores all the items, $I$, with $\hat{y}(i|c)$, sorts the scores and returns the highest ranked ones. This is very costly, with a linear dependency on $|I|$.

This makes this method infeasible for online scoring for large or medium sized item catalogues because applications commonly expect the results in a few milliseconds within a user's request. For moderately sized catalogues, the top items can be precomputed offline and stored for each context that could be queried. However, the size of possible query context can be much larger than the set of training context $|C|$ and evaluating and storing all top lists becomes infeasible. For example, for a sequential recommender, the context might be the $k$ most recent clicks of the user, the potential space of all context is $|I|^k$. It is impossible to precompute the top scoring items for all of these $|I|^k$ context. Offline computation becomes also a problem if the recommender system is trained online, e.g., a user's embedding might be updated online which would require recomputing the top scoring items. To summarize, brute-force algorithms are only applicable if the number of items is small or for a static model if the number of context is small. In the former case, brute-force scoring can be applied online, in the latter, brute-force scoring can be applied offline.

## 6.2 Approximate Nearest Neighbor Search

For dot product models, $\hat{y}(i|c) = \langle \boldsymbol{\phi}(c), \boldsymbol{\psi}(i) \rangle$, the problem of finding the top scoring items is equivalent to the well studied field of *maximum inner product search* or *approximate nearest neighbor search* [26]. The nearest neighbor search problem is to find the closest entries in a database of vectors for a query vector. For item recommendation, the context $\boldsymbol{\phi}(c)$ corresponds to the query and the database of vectors are the items $I$ represented by the embeddings $\boldsymbol{\psi}(i)$. These problems are very well studied with efficient sublinear approximate algorithms that can be applied for the item recommendation problem. Typically, solutions are based on narrowing down the set of possible nearest neighbors that needs to be evaluated, e.g., using partitioning algorithms such as trees [15, 31], and efficient scoring using quantization [26].

These efficient sublinear algorithms make dot product models very attractive for large scale item recommendation. If an application requires fast retrieval, dot product models are the best choice.

## 6.3 Dynamic User Model

When a user interacts with a recommender system, the system should be responsive and change the recommendation based on a user's feedback. For example, after watching a video, the system should be able to make better recommendations taking into account this new information. Depending on the type of scoring function, this might require to update the model itself. For example, in a user-item matrix factorization model, the user embedding $\mathbf{w}_u$ would need to be retrained in real time after every interaction—for user-item matrix factorization a projection, Eq. (30) can be applied. Other scoring functions avoid retraining by representing the user as a function of their past behavior. For example, if the user is represented by the average item embedding in their history, the user embedding function $\boldsymbol{\phi}(c)$ is trivial to compute and can be done online at query time. In both cases, the main model can be retrained occasionally offline to propagate all training data through the model.

## 7 Conclusion

This chapter introduced the item recommendation problem and discussed its unique challenges. Several popular approaches to train recommender systems from implicit feedback data were presented. An advantage of sampling based approaches is that they can be applied to most recommender models. That makes them a popular choice for learning item recommenders. Another advantage is that they can be adapted to retrieval metrics by rank based weighting and sampling schemes. However, if the item catalogue is large, sampling based approaches are slow unless they use good samplers which is an open area of research. For special cases the Gramian trick allows to learn item recommenders much more efficently, typically dropping the runtime dependency on the catalogue size $|I|$. However, this trick is only applicable to square losses which can be less effective than a ranking loss such as weighted pairwise or softmax. Dot product models that represent the context and the items by a $d$-dimensional embedding have useful properties throughout learning and retrieval. For learning, their structure can be exploited for faster sampling, or the Gramian trick, and for retrieval it allows real-time retrieval in sublinear time through maximum inner product search algorithms.

As a conclusion of this chapter, a few rules of thumb for applying recommender systems in practice are suggested. Dot product models are a reasonable default choice due to their attractive retrieval properties. The family of dot product models is broad including popular approaches such as matrix factorization but also more complex techniques such as deep neural networks including transformers, recurrent structures, or in general any model that extracts representations which are combined in the final layer with a dot product. For learning, sampling based algorithms such as sampled softmax with a large number of samples $m$ are easy to implement and can result in good ranking quality. If the item catalogue is very large, algorithms based on the Gramian trick can be a better choice, and implementations for models like matrix factorization are simple. In any case, properly tuning and setting up models can be more important than switching to a more complex approach [7, 21, 22].

**Acknowledgments** I would like to thank Nicolas Mayoraz and Li Zhang for helpful comments and suggestions.

## References

1. Y. Bai, S. Goldman, L. Zhang, TAPAS: Two-pass Approximate Adaptive Sampling for Softmax. arXiv: 1707.03073 (2017)
2. I. Bayer, X. He, B. Kanagal, S. Rendle, A generic coordinate descent framework for learning from implicit feedback, in *Proceedings of the 26th International Conference on World Wide Web WWW'17* (2017), pp. 1341–1350
3. Y. Bengio, J.-S. Senecal, Quick training of probabilistic neural nets by importance sampling, in *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics, AISTATS 2003, Key West, January 3–6, 2003* (2003)
4. G. Blanc, S. Rendle, Adaptive sampled softmax with kernel based sampling, in *Proceedings of the 35th International Conference on Machine Learning*, ed. by J. Dy, A. Krause. Proceedings of Machine Learning Research, vol. 80, Stockholmsmässan, Stockholm, 10–15 July 2018 (2018), pp. 590–599
5. C.J.C. Burges, From ranknet to lambdarank to lambdamart: an overview. Technical Report MSR-TR-2010-82 (2010)
6. P. Covington, J. Adams, E. Sargin, Deep neural networks for youtube recommendations, in *Proceedings of the 10th ACM Conference on Recommender Systems, RecSys'16* (Association for Computing Machinery, New York, 2016), pp. 191–198
7. M.F. Dacrema, S. Boglio, P. Cremonesi, D. Jannach, A troubling analysis of reproducibility and progress in recommender systems research. ACM Trans. Inf. Syst. **39**(2), 1–49 (2021). https://doi.org/10.1145/3434185
8. R.A. Harshman, Foundations of the PARAFAC procedure: models and conditions for an "explanatory" multi-modal factor analysis. UCLA Work. Pap. Phonetics **16**(1), 84 (1970)
9. B. Hidasi, D. Tikk, Fast ALS-based tensor factorization for context-aware recommendation from implicit feedback, in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (Springer, Berlin, 2012), pp. 67–82
10. Y. Hu, Y. Koren, C. Volinsky, Collaborative filtering for implicit feedback datasets, in *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining, ICDM'08* (2008), pp. 263–272
11. R. Jiang, S. Gowal, Y. Qian, T. Mann, D.J. Rezende, Beyond greedy ranking: slate optimization via list-CVAE, in *International Conference on Learning Representations* (2019)
12. T. Joachims, Optimizing search engines using clickthrough data, in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'02* (Association for Computing Machinery, New York, 2002), pp. 133–142
13. W. Krichene, N. Mayoraz, S. Rendle, L. Zhang, X. Yi, L. Hong, E. Chi, J. Anderson, Efficient training on very large corpora via gramian estimation, in *7th International Conference on Learning Representations* (2019)
14. W. Krichene, S. Rendle, On sampled metrics for item recommendation, in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD'20* (Association for Computing Machinery, New York, 2020), pp. 1748–1757
15. M. Muja, D.G. Lowe, Scalable nearest neighbor algorithms for high dimensional data. IEEE Trans. Pattern Anal. Mach. Intell. **36**(11), 2227–2240 (2014)
16. X. Ning, G. Karypis, SLIM: sparse linear methods for top-n recommender systems, in *Proceedings of the 2011 IEEE 11th International Conference on Data Mining, ICDM'11* (IEEE Computer Society, Washington, 2011), pp. 497–506
17. A.S. Rawat, J. Chen, F.X.X. Yu, A.T. Suresh, S. Kumar, Sampled softmax with random fourier features, in *Advances in Neural Information Processing Systems*, ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett, vol. 32 (Curran Associates, Red Hook, 2019), pp. 13857–13867
18. S. Rendle, Factorization machines, in *Proceedings of the 2010 IEEE International Conference on Data Mining, ICDM'10* (IEEE Computer Society, Washington, 2010), pp. 995–1000

19. S. Rendle, C. Freudenthaler, Improving pairwise learning for item recommendation from implicit feedback, in *Proceedings of the 7th ACM International Conference on Web Search and Data Mining, WSDM'14* (Association for Computing Machinery, New York, 2014), pp. 273–282
20. S. Rendle, C. Freudenthaler, Z. Gantner, L. Schmidt-Thieme, BPR: Bayesian personalized ranking from implicit feedback, in *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI'09* (AUAI Press, Arlington, 2009), pp. 452–461
21. S. Rendle, L. Zhang, Y. Koren, On the difficulty of evaluating baselines: a study on recommender systems. CoRR, abs/1905.01395 (2019)
22. S. Rendle, W. Krichene, L. Zhang, J. Anderson, Neural collaborative filtering vs. matrix factorization revisited, in *Proceedings of the 14th ACM Conference on Recommender Systems, RecSys'20* (2020)
23. G. Takács, D. Tikk, Alternating least squares for personalized ranking, in *Proceedings of the Sixth ACM Conference on Recommender Systems, RecSys'12* (Association for Computing Machinery, New York, 2012), pp. 83–90
24. L.R. Tucker, Some mathematical notes on three-mode factor analysis. Psychometrika **31**, 279–311 (1966)
25. N. Usunier, D. Buffoni, P. Gallinari, Ranking with ordered weighted pairwise classification, in *Proceedings of the 26th Annual International Conference on Machine Learning, ICML'09* (Association for Computing Machinery, New York, 2009), pp. 1057–1064
26. J. Wang, W. Liu, S. Kumar, S.-F. Chang, Learning to hash for indexing big data - a survey. CoRR, abs/1509.05472 (2015)
27. J. Weston, S. Bengio, N. Usunier, Wsabie: scaling up to large vocabulary image annotation, in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Three, IJCAI'11* (AAAI Press, Palo Alto, 2011), pp. 2764–2770
28. M. Wilhelm, A. Ramanathan, A. Bonomo, S. Jain, E.H. Chi, J. Gillenwater, Practical diversified recommendations on youtube with determinantal point processes, in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM'18* (Association for Computing Machinery, New York, 2018), pp. 2165–2173
29. D. Xin, N. Mayoraz, H. Pham, K. Lakshmanan, J.R. Anderson, Folding: why good models sometimes make spurious recommendations, in *Proceedings of the Eleventh ACM Conference on Recommender Systems, RecSys'17* (Association for Computing Machinery, New York, 2017), pp. 201–209
30. J. Yang, X. Yi, D. Zhiyuan Cheng, L. Hong, Y. Li, S. Xiaoming Wang, T. Xu, E.H. Chi, Mixed negative sampling for learning two-tower neural networks in recommendations, in *Companion Proceedings of the Web Conference 2020, WWW'20* (Association for Computing Machinery, New York, 2020), pp. 441–447
31. P.N. Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, in *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'93* (Society for Industrial and Applied Mathematics, Philadelphia, 1993), pp. 311–321
32. F. Yuan, G. Guo, J.M. Jose, L. Chen, H. Yu, W. Zhang, Lambdafm: Learning optimal ranking with factorization machines using lambda surrogates, in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, CIKM'16* (Association for Computing Machinery, New York, 2016), pp. 227–236