Forget Me Now: Fast and Exact Unlearning in Neighborhood-based Recommendation

Sebastian Schelter
University of Amsterdam
Amsterdam, The Netherlands
s.schelter@uva.nl

Mozhdeh Ariannezhad
AIRLab, University of Amsterdam
Amsterdam, The Netherlands
m.ariannezhad@uva.nl

Maarten de Rijke
University of Amsterdam
Amsterdam, The Netherlands
m.derijke@uva.nl

ABSTRACT

Modern search and recommendation systems are optimized using logged interaction data. There is increasing societal pressure to enable users of such systems to have some of their data deleted from those systems. This paper focuses on "unlearning" such user data from neighborhood-based recommendation models on sparse, high-dimensional datasets. We present CABOOSE, a custom top-k index for such models, which enables fast and exact deletion of user interactions. We experimentally find that CABOOSE provides competitive index building times, makes sub-second unlearning possible (even for a large index built from one million users and 256 million interactions), and, when integrated into three state-of-the-art next-basket recommendation models, allows users to effectively adjust their predictions to remove sensitive items.

CCS CONCEPTS

• **Information systems** → **Data management systems**; **Recommender systems**.

KEYWORDS

Machine unlearning, right-to-be-forgotten, nearest neighbors

ACM Reference Format:
Sebastian Schelter, Mozhdeh Ariannezhad, and Maarten de Rijke. 2023. Forget Me Now: Fast and Exact Unlearning in Neighborhood-based Recommendation. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '23), July 23–27, 2023, Taipei, Taiwan.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3539618.3591989

1 INTRODUCTION

Ranking methods optimize search and recommendation systems so that the resulting rankings perform well for a given metric. Traditionally, most ranking methods applied a supervised learning procedure based on manually-created judgments. As an alternative, ranking methods have been developed that rely on logged user interactions [17].

The "right to be forgotten". Recent law such as the "right to be forgotten" in Europe [General Data Protection Regulation (GDPR), Article 17, 9] requires organizations to delete personal user data, including interaction data, upon request: "The data subject shall have the right to [...] the erasure of personal data [...] where the data subject withdraws consent." Research on "machine unlearning" [4, 11, 16, 30, 35, 37, 38] argues that it is insufficient to delete personal data from primary data stores; machine learning models that have been trained on the stored data also fall under the regulation. Outside Europe, similar regulations are being adopted [1, 34].

The need for timely machine unlearning. GDPR does not specify how soon data must be erased after a deletion request [31], yet it states the "obligation to erase personal data without undue delay" [9] using "appropriate and effective measures" [10]. Currently, data erasure seems to be a rather tedious and lengthy process in practice; e.g., data erasure from active systems in the cloud can take up to two months [31]. Not being able to enforce this right in a timely manner can have dramatic consequences in practice [36].

Therefore, we need to design search and recommendation approaches with timely "unlearning" capabilities, to empower users to quickly delete their interaction data and adjust their predictions on demand. For such unlearning methods to be effective in practice, they have to be (*i*) *fast*, to allow users to interactively delete their data from existing models, and (*ii*) *exact*, (e.g., no approximate updates, no hyperparameters to tune, no restriction on the maximum number of updates), to be easy to integrate into existing models. Unfortunately, machine unlearning is a hard problem, and existing general solutions either require partial iterative retraining of the underlying model [13, 16, 38] (and are therefore not fast), and/or can only conduct approximate updates for a small number of data points [13, 16, 20] (and are therefore not exact). Recent approaches to unlearning in recommendation such as RecEraser [5] inherit these limitations and conduct partial retraining, which can take several hours even for small datasets with 10 million interactions.

Fast and exact unlearning for neighborhood-based recommendation. Fast and exact unlearning is still possible under certain algorithmic and data-specific conditions. One such condition are k-nearest neighbor (kNN) models trained on sparse data. Such models are highly relevant for recommender systems, which work with extremely sparse interaction data. User-centric kNN models have a long tradition in this area, ranging from classical user-based collaborative filtering [28] to recent state-of-the-art algorithms in session-based [23, 24], session-aware [22], next-basket (NBR) [7, 14], and within-basket recommendation [3]. Moreover, compared to neural approaches to recommendation, kNN models are more transparent and explainable [24], cheap to scale to industry workloads [19] and often require an order of magnitude less training time [18].

Our contributions. We formalize the unlearning problem for kNN models in Section 2. Next, we detail CABOOSE, an in-memory index of the top-k most similar users in a sparse interaction dataset, which

SIGIR '23, July 23–27, 2023, Taipei, Taiwan

ACM ISBN 978-1-4503-9408-6/23/07.
https://doi.org/10.1145/3539618.3591989

enables unlearning of interactions in a fast and exact manner (Section 3). We achieve this by exploiting sparsity, choosing efficient data structures and parallelizing updates, while building upon existing work on fast indexing [32]. Our experimental evaluation in Section 4 shows that CABOOSE provides competitive index building times, makes sub-second unlearning possible (even for a large index built from 1 million users and 256 million interactions), and, when integrated into state-of-the-art NBR models, allows users to effectively adjust their predictions to remove sensitive items. Furthermore, we share an implementation of CABOOSE in Rust under an open license at https://github.com/amsterdata/caboose.

2 PROBLEM STATEMENT

We focus on user-centric kNN models for recommendation [3, 7, 14, 28]. These approaches typically model the interactions of n users with m items via a sparse matrix $\boldsymbol{U} \in \mathbb{R}^{n \times m}$ and compute the top-k similar users per user from this matrix, based on a similarity function between rows, which we denote Φ. At training time, such user-centric kNN models build an index of the top-k similar users per user. We denote this index with $\mathcal{T}_U$ and assume that it is built via a procedure BUILD_INDEX$(\boldsymbol{U}, \Phi, k)$. An entry $\mathcal{T}_U(i) = [(\boldsymbol{u}_{s_1}, \phi_{is_1}), \ldots, (\boldsymbol{u}_{s_k}, \phi_{is_k})]$ of this index for a user $\boldsymbol{u}_i$ contains the k most similar users $\boldsymbol{u}_{s_1}, \ldots, \boldsymbol{u}_{s_k}$ with their corresponding similarities $\phi_{is_1}, \ldots, \phi_{is_k}$. At inference time, such a top-k index provides $O(1)$ access to the neighbors of a particular user.

In order to enable unlearning for user-centric kNN models, we have to answer the following research question: *Given an existing index $\mathcal{T}_U$ and an interaction u_{ig} of a user i with an item g to unlearn, how can we efficiently compute $\mathcal{T}_{(U-u_{ig})}$?*

This deletion affects more than just the entry $\mathcal{T}_U(i)$, as the user $\boldsymbol{u}_i$ could be contained in the top-k entries of many other users! We could obviously simply re-compute the top-k index from scratch as $\mathcal{T}_{(U-u_{ig})} = \text{BUILD\_INDEX}(\boldsymbol{U} - u_{ig}, \Phi, k)$. However, this is expensive, as recomputing the index from scratch can take many hours for large datasets, and wastes a lot of computation, as $\mathcal{T}_U$ and $\mathcal{T}_{(U-u_{ig})}$ are likely to only differ in a relatively small number of entries. Furthermore, this problem is not solved by existing vector indexes such as FAISS [27], Pinecone [26] or Hnswlib [8], which support deletions, but are designed for approximate search on dense vectors with a comparatively small number of dimensions only.

3 PROPOSED APPROACH

To tackle our research question, we design CABOOSE, an in-memory index of the top-k most similar users in a sparse interaction dataset, which enables unlearning interactions in a fast and exact manner.

CABOOSE provides an algorithm FORGET to unlearn an interaction u_{ig} from an existing index $\mathcal{T}_U$, which produces the same result $\mathcal{T}_{(U-u_{ig})}$ as re-computing the index from scratch without the interaction to unlearn:

$\mathcal{T}_{(U-u_{ig})} = \text{FORGET}(\mathcal{T}_U, u_{ig}, \Phi, k) = \text{BUILD\_INDEX}(\boldsymbol{U} - u_{ig}, \Phi, k)$.

At an abstract level, our FORGET algorithm proceeds in four stages to derive the updated index $\mathcal{T}_{(U-u_{ig})}$ from an existing index $\mathcal{T}_{(U)}$:

- *Stage 1*: Update $\boldsymbol{U}$ to $\boldsymbol{U} - u_{ig}$, update $\boldsymbol{U}^\top$ to $(\boldsymbol{U} - u_{ig})^\top$ and the precomputed norm of $\boldsymbol{u}_i$ to reflect the deletion of u_{ig}.
- *Stage 2*: Recompute the dot products $(\boldsymbol{u}_i - u_{ig})(\boldsymbol{U} - u_{ig})^\top$ involving $\boldsymbol{u}_i$ to obtain its updated similarities and identify the set of affected top-k entries R.

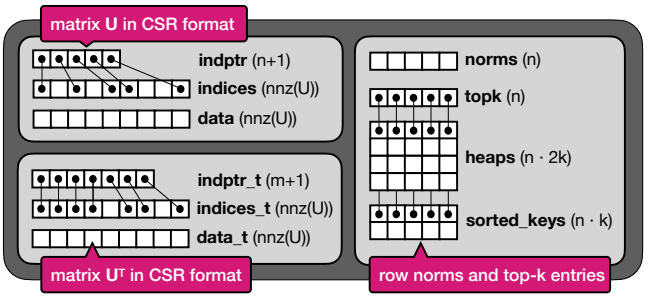

Figure 1: Index layout in memory.

- *Stage 3*: Inspect and (where possible) directly update the top-k entry $\mathcal{T}_U(r)$ for each row $r \in R$ affected by the deletion of u_{ig}.
- *Stage 4*: Recompute the top-k entries from scratch for rows in R, which cannot be directly updated.

For this algorithm to be efficient in practice, we design it with the following characteristics in mind: (i) We *exploit sparsity* wherever possible to only work with the top-k entries directly affected by the unlearning operation. Hence, we restrict our approach to similarity functions which are zero if the dot product between two rows is zero, and which can be computed from the dot product and norms of vectors. As a result, we can ignore pairs of users without a shared item interaction. This class involves many commonly used similarity measures, e.g., cosine similarity, computed as $\phi_{ij} = \boldsymbol{u}_i^\top \boldsymbol{u}_j / (\|\boldsymbol{u}_i\| \, \|\boldsymbol{u}_j\|)$ or Jaccard similarity. (ii) We *parallelize* the individual stages of our algorithm, and (iii) choose *appropriate data structures* that enable efficient low-level operations (e.g., compressed representations for sparse matrices and binary heaps for top-k lists).

Index layout. The memory layout of CABOOSE is shown in Figure 1. It enables efficient row-wise and column-wise access to $\boldsymbol{U}$, by holding $\boldsymbol{U}$ and $\boldsymbol{U}^\top$ in compressed sparse row form (CSR). CSR represents a matrix with three arrays: the array data contains its non-zero values, and the array indices holds the corresponding column indices, and both are accessed through the array indptr which denotes the range of columns belonging to each row. Also, CABOOSE contains an array norms for the norm of each row (e.g., the L^2-norm for cosine similarity). The top-k entries $\mathcal{T}_U$ are represented by the n-dimensional array topk, where each entry topk$[i]$ represents $\mathcal{T}_U(i)$, and contains a binary heap of length k, which stores tuples of row identifiers and the corresponding similarity scores. The tuple with the k-largest similarity score is at the heap root, which enables us to check in $O(1)$ time if a new similarity tuple is among the k-largest tuples, and insert such a tuple with complexity $O(\log k)$. Each entry additionally contains a sorted array sorted_keys of length k with the row identifiers from the heap to enable $O(\log k)$ membership tests for keys via binary search.

This design is inspired by [32] and we build our index accordingly: We precompute the norms for all rows, execute a sparse vector matrix multiplication $\boldsymbol{u}_i \boldsymbol{U}^\top$ for each row $\boldsymbol{u}_i$ in parallel to obtain its dot products with other rows, calculate the final similarity based on the precomputed norms and extract the top-k similar rows afterwards.

Unlearning algorithm. Next, we detail in Algorithm 1 how to efficiently conduct the four stages for unlearning an interaction u_{ig} of user i with item g from an existing index $\mathcal{T}_U$.

Algorithm 1 Unlearning an interaction u_{ig} from the index $\mathcal{T}_U$.

```
1:  function FORGET(row i, column g)
                          // Stage 1 – Updating U, U^⊤ and the precomputed norms
2:      norms[i] ← UPDATE_NORM(norms[i], u_ig)
3:      set entry corresponding to (i, g) in data to 0
4:      set entry corresponding to (g, i) in data_t to 0
5:      A ← ∅                 // Stage 2 – Parallel computation of (u_i − u_ig)(U − u_ig)^⊤
6:      parfor c ∈ [indptr[i], ..., indptr[i + 1]] in partitions of size m/#cores do
7:          a ← accumulator of capacity n
8:          for j ∈ c do SPARSE_MULT(a, j)
9:          A ← A ∪ a
10:     end parfor
11:     (R, t_new) ← MERGE_AND_COLLECT(i, k, norms, A)
12:     topk[i] ← t_new
13:     F ← ∅                 // Stage 3 – Parallel update of affected top-k entries
14:     parfor (r, φ_ir) ∈ R do
15:         if ¬FIND_WITH_BINARY_SEARCH(i, topk[r].sorted_keys) then
16:             if φ_ir > root of topk[r].heap then UPDATE_ROOT(topk[r], (i, φ_ir))
17:         else
18:             if φ_ir ≠ 0 then
19:                 if φ_ir < root of topk[r].heap then F ← F ∪ r
20:                 else UPDATE_TOPK(topk[r], (i, φ_ir))
21:             else
22:                 if |topk[r]| < k then REMOVE_FROM_TOPK(topk[r], i)
23:                 else F ← F ∪ r
24:     end parfor
                          // Stage 4 – Parallel recomputation of non-updatable top-k entries
25:     parfor c ∈ F in partitions of size 1024 do
26:         a ← accumulator of capacity n
27:         for r ∈ c do
28:             for j ∈ [indptr[r], ..., indptr[r + 1]] do SPARSE_MULT(a, j)
29:             topk[r] ← TOPK_AND_CLEAR(a, r, k, norms)
30:     end parfor
31: function SPARSE_MULT(accumulator a, pointer j)
32:     for l ∈ [indptr_t[indices[j]], ..., indptr_t[indices[j + 1]]] do
33:         ACCUMULATE(a, indices_t[l], data_t[l] · data[j])
```

Stage 1 – Updating U, $U^\top$ and the precomputed norms. Lines 2–4 in Algorithm 1 update the stored norm of u_i according to the norm required for the similarity (e.g., to $\texttt{sqrt}(\text{norms}[i]^2 - u_{ig}^2)$ for cosine similarity) and set the entries corresponding to u_{ig} in the CSR representations of U and $U^\top$ to 0.

Stage 2 – Parallel dot product updates $(u_i - u_{ig})(U - u_{ig})^\top$. Next, we recompute the dot products for the updated row u_i by computing $(u_i - u_{ig})(U - u_{ig})^\top$ via sparse parallel vector matrix multiplication in Lines 5–10. From the result, we obtain the updated top-k similarities t_{new} (representing $\mathcal{T}_{U-u_{ig}}(i)$), updated similarities and affected rows R (which have a non-zero dot product with u_i) (Line 11).

Stage 3 – Parallel updates of affected top-k entries. Next, we inspect and potentially change each entry $\mathcal{T}_U(r)$ for an affected row $u_r \in R$ with an updated similarity ϕ_{ir} between u_i and u_r (Lines 13–24). Note that we need to distinguish several cases here:

- Not-in-top-k: In Line 15 we first test whether the row u_i is already part of the top-k entry $\mathcal{T}_U(r)$ of u_i via binary search on topk[r].sorted_keys. If u_i is not contained, we check whether the unlearning operation changed u_i to be part of the top-k entry of u_i by comparing the new similarity ϕ_{ir} to the heap root of topk[r] (the k-largest similarity to u_r). If this holds, we update the corresponding top-k entry via UPDATE_ROOT in Line 16.
- Already-in-top-k: If u_i is already in the top-k of u_r and the updated similarity ϕ_{ir} is non-zero, then we have to distinguish two cases: (i) if ϕ_{ir} is not smaller than the current heap root, we can simply recreate the corresponding heap to reflect the changed similarity (Line 20). Otherwise, (ii) the $k+1$th element (which is not contained in the index) might actually be larger than the updated one for u_i, therefore we need to recompute the top-k entry for u_r from scratch. We add r to the set F to schedule this recomputation later (Line 19).
- To-be-removed-from-top-k: The trickiest case is when u_i has been in the top-k of u_r already, but the unlearning operation results in a zero similarity ϕ_{ir} (Line 21). This happens if g is the last shared non-zero column between u_i and u_r. We distinguish two cases here: (i) if there are less than k similar rows for u_r in the data anyway, we can just delete u_i from the top-k entry of u_r via REMOVE_FROM_TOPK (Line 22). However, (ii) if the top-k entry of u_r has k elements, than we would need to remove u_i and replace it with the $(k+1)$-st most similar row (which is not stored in the current index). As a consequence, we again need to recompute the top-k entry for u_r from scratch, and add r to the set F to schedule this recomputation later (Line 23).

Stage 4 – Parallel recomputation of non-updatable top-k entries. We finally recompute the non-updatable top-k entries contained in the set F from scratch via parallel sparse vector matrix multiplications in Lines 25–30. We empirically find this to happen rarely.

Due to lack of space, we refer to our shared code repository for details on the accumulators and update functions.

Table 1: Sparse interaction datasets used for evaluation.

Dataset	Domain	#Users	#Items	#Interactions
movie [12]	movie	69,879	10,678	10,000,055
lastfm [21]	band	993	174,078	19,150,868
syn [29]	synthetic	100,000	50,000	50,000,000
spotify [33]	song	1,000,000	2,262,292	66,346,428
yahoo [39]	song	1,000,991	624,962	256,804,236

4 EXPERIMENTAL EVALUATION

We implement CABOOSE in Rust and evaluate it on five sparse interaction datasets listed in Table 1. We use cosine similarity for the index, and if not reported otherwise, run experiments on a machine with a four-core Intel i7-8569U CPU @2.80GHz, 16GB of RAM and MacOS 12.6. Our experiment code is available at https://github.com/amsterdata/caboose.

Experiment 1: Index building time. In our first experiment, we showcase that CABOOSE can be built in a time that is competitive with existing approaches. We compute the top-k similar users for all datasets with $k = 50$ using our index and two baselines: `similaripy` [32] from the RecSys Challenge 2018, and unsupervised nearest neighbors from `sklearn` [25]. We repeat each run seven times and report the mean time to build the index. Note that we run the experiments for `spotify` and `yahoo` on a larger machine with 64 AMD EPYC 7H12 2.6 GHz cores and AlmaLinux 8.6, as the experiment would take several days otherwise.

Results and discussion. Figure 2 plots the mean runtimes in milliseconds on a logarithmic scale. We had to cancel the experiments for `sklearn` on the large datasets, as this implementation did not manage to make proper use of all cores of the system. We find that CABOOSE outperforms `sklearn` in all cases by a factor of up to 2.26 and `similaripy` in three out of five datasets by a factor of up to 2.27. We attribute the runtime differences with `similaripy`

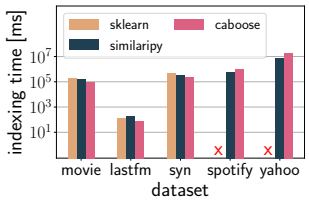

Figure 2: Indexing times for different datasets.

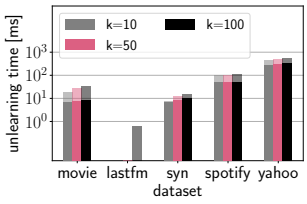

Figure 3: Unlearning latencies for different datasets.

to different optimization choices in the underlying parallelization libraries `OpenMP` and `rayon`. We conclude that our index building time is competitive, even though CABOOSE has to conduct extra work for its unlearning capability. Additionally, we see that the unlearning functionality becomes crucial when scaling to large datasets, where indexing can take several hours.

Experiment 2: Low-latency unlearning. The goal of our next experiment it to show that CABOOSE can unlearn interactions with sub-second latency, and is thereby fast enough for interactive use. We build indexes with $k \in [10, 50, 100]$ for our datasets. Next, we randomly choose 500 interactions to forget and measure the median (`p50`) and 90-th percentile (`p90`) of the unlearning time per interaction. We cannot include `similaripy` or `sklearn` in this experiment, as they lack unlearning functionality.

Results and discussion. Figure 3 plots the unlearning latencies in milliseconds on a logarithmic scale. CABOOSE unlearns interactions very fast with a median latency of less than 100ms for all datasets, except for `yahoo`. Even for this large dataset with over one million users, our index still achives sub-second unlearning latencies with a `p50` of 317ms and a `p90` of 550ms. In the majority of cases, unlearning an interaction is about four orders of magnitude faster than rebuilding the index from scratch. For `lastfm`, unlearning always takes less than a millisecond, due to the low number of users in this dataset. Additionally, we observe that the range of k has a low impact on the unlearning latency, the `p50` latencies for `spotify` and `yahoo` only increase by 4.7% and 11.9% when we increase k from 10 to 100. These findings confirm that CABOOSE is suitable for interactive use, as end users in general perceive response latencies below 500ms as instantaneous [2].

Experiment 3: Prediction adjustment via unlearning. Next, we integrate CABOOSE into the Python implementations of the three recent state-of-the-art NBR approaches TIFU-KNN [14], PerNIR [3] and UP-CF@r [7], to enhance them with unlearning functionalities. Note that PerNIR is originally proposed for within-basket recommendation, but can be used as a NBR model assuming the current basket is empty. We implement an algorithm-specific `forget` method for each approach, which uses CABOOSE to unlearn interactions and updates algorithm-specific internal data structures.

In our last experiment, we showcase that unlearning selected interactions provides a simple way for users to adjust their predictions from recommendation models, e.g., to remove sensitive items from their recommendations. There is no hard guarantee for this to work, as the impact of removals depends on the model details and co-occurrence structure in the data. In the worst, a user could simply unlearn all interactions stored for them to get rid of any personalized recommendations. We evaluate a simple strategy in an e-commerce setting: We assume that a user does not want to see products from a certain sensitive product category, even though they interacted with such products in the past. If the user is exposed to recommendations with such sensitive items, they ask the system to unlearn their past interactions with products from this sensitive category. We use the instacart30k [15] dataset for grocery shopping as a basis for the experiment. We define three types of sensitive product categories: (*i*) *baby items* motivated by a report on a recent traumatizing case [36], (*ii*) *meat*-related categories to mimic a person changing to a vegetarian diet, and (*iii*) *alcohol*-related categories to mimic a person suffering from alcohol addiction.

Table 2: Impact of unlearning sensitive items from user histories to remove sensitive items from their predictions.

Sensitive	Users affected			Removal success		
category	TIFU	PerNIR	UP-CF	TIFU	PerNIR	UP-CF
baby items	22.4%	23.5%	21.5%	100.00%	100.00%	99.80%
meat	36.3%	37.5%	35.3%	100.00%	100.00%	99.79%
alcohol	33.1%	33.2%	35.3%	99.65%	99.62%	97.90%

For each sensitive category, we sample the baskets of 1,000 users who bought at least one item from this category, determine the other non-sensitive categories in their baskets, and sample 1,000 additional users who bought items from these categories but not from the sensitive category. We train the models on this dataset of 2,000 users (with default hyperparameters), and inspect the top-10 predicted items for each user with sensitive purchases. If such a user has sensitive items in their predictions, we make the model forget all the user's historical interactions with items from the sensitive category, recompute the predictions for the user and check if any sensitive items remain in the top-10 predictions. We repeat this experiment with seven random seeds for the three models and three categories, and report the mean number of affected users (who initially had sensitive items in their recommendations) and the fraction of such users where the simple unlearning strategy successfully removed the sensitive items from their recommendations.

Results and discussion. We detail the results of this experiment in Table 2. Across all models and categories, a large fraction (between 21.5% and 37.5%) of users who interacted with sensitive items also get exposed to sensitive items in their top-10 predictions. The simple removal strategy of forgetting past interactions with sensitive items is very effective, and removes almost all sensitive items from the predictions. It fails in a small number of cases (mostly for alcohol), which we attribute to the strong coocurrence between an alcoholic beverage and non-alcoholic items like certain dishes in the data; here the user would have to forget these interactions as well.

5 CONCLUSION

We formalized the problem of unlearning for kNN models on sparse data, and detailed how to unlearn interactions in a fast and exact manner from a custom in-memory top-k index. For future work, we plan to integrate GraphBLAS [6] for accelerating our sparse operations. We aim to avoid the potential full recomputation of individual top-k entries from a "budget" of more than the required k most similar users computed at indexing time.

Acknowledgements. This work was supported by Ahold Delhaize. All content represents the opinion of the authors, which is not necessarily shared or endorsed by their respective employers and/or sponsors.

REFERENCES

[1] California Privacy Protection Agency. [n.d.]. California Consumer Privacy Act - Frequently Asked Questions. https://cppa.ca.gov/faq.html#faq_res_1

[2] Ioannis Arapakis, Xiao Bai, and B Barla Cambazoglu. 2014. Impact of response latency on user behavior in web search. *SIGIR* (2014), 103–112.

[3] Mozhdeh Ariannezhad, Ming Li, Sebastian Schelter, and Maarten de Rijke. 2023. A Personalized Neighborhood-based Model for Within-basket Recommendation in Grocery Shopping. *WSDM* (2023), 87–95.

[4] Yinzhi Cao and Junfeng Yang. 2015. Towards Making Systems Forget with Machine Unlearning. *IEEE Symposium on Security and Privacy* (2015), 463–480.

[5] Chong Chen, Fei Sun, Min Zhang, and Bolin Ding. 2022. Recommendation Unlearning. *WWW* (2022), 2768–2777.

[6] Timothy A Davis. 2019. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)* 45, 4 (2019), 1–25.

[7] Guglielmo Faggioli, Mirko Polato, and Fabio Aiolli. 2020. Recency Aware Collaborative Filtering for Next Basket Recommendation. *UMAP* (2020), 80–87.

[8] Hnswlib fast approximate nearest neighbor search. [n.d.]. Does this version of hnsw support sparse matrices? https://github.com/nmslib/hnswlib/issues/13

[9] GDPR.eu. [n.d.]. Article 17: Right to be forgotten. https://gdpr.eu/article-17-right-to-be-forgotten.

[10] GDPR.eu. [n.d.]. Recital 74: Responsibility and liability of the controller. https://gdpr.eu/recital-74-responsibility-and-liability-of-the-controller/.

[11] Antonio Ginart, Melody Y. Guan, Gregory Valiant, and James Zou. 2019. Making AI Forget You: Data Deletion in Machine Learning. *NeurIPS* (2019).

[12] Grouplens. [n.d.]. MovieLens 10M Dataset. http://konect.cc/networks/movielens-10m_rating/

[13] Chuan Guo, Tom Goldstein, Awni Hannun, and Laurens Van Der Maaten. 2020. Certified Data Removal from Machine Learning Models. *ICML* (2020), 3832–3842.

[14] Haoji Hu, Xiangnan He, Jinyang Gao, and Zhi-Li Zhang. 2020. Modeling Personalized Item Frequency Information for Next-basket Recommendation. *SIGIR* (2020), 1071–1080.

[15] Instacart. [n.d.]. Instacart Market Basket Analysis. https://www.kaggle.com/c/instacart-market-basket-analysis

[16] Zachary Izzo, Mary Anne Smart, Kamalika Chaudhuri, and James Zou. 2021. Approximate Data Deletion from Machine Learning Models. (2021), 2008–2016.

[17] Thorsten Joachims. 2002. Optimizing Search Engines Using Clickthrough Data. *KDD* (2002), 133–142.

[18] Barrie Kersbergen and Sebastian Schelter. 2021. Learnings from a Retail Recommendation System on Billions of Interactions at bol.com. *ICDE* (2021), 2447–2452.

[19] Barrie Kersbergen, Olivier Sprangers, and Sebastian Schelter. 2022. Serenade - Low-Latency Session-Based Recommendation in e-Commerce at Scale. 150–159.

[20] Pang Wei Koh and Percy Liang. 2017. Understanding Black-box Predictions via Influence Functions. *ICML* (2017), 1885–1894.

[21] Last.fm. [n.d.]. Last.fm bands. http://konect.cc/networks/lastfm_band/

[22] Sara Latifi, Noemi Mauro, and Dietmar Jannach. 2021. Session-aware Recommendation: A Surprising Quest for the State-of-the-art. *Information Sciences* 573 (2021), 291–315.

[23] Malte Ludewig and Dietmar Jannach. 2018. Evaluation of Session-based Recommendation Algorithms. *User Modeling and User-Adapted Interaction* 28, 4-5 (2018), 331–390.

[24] Malte Ludewig, Noemi Mauro, Sara Latifi, and Dietmar Jannach. 2021. Empirical Analysis of Session-based Recommendation Algorithms. *User Modeling and User-Adapted Interaction* 31, 1 (2021), 149–181.

[25] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *JMLR* 12 (2011), 2825–2830.

[26] Pinecone. [n.d.]. The Pinecone vector database - Limits. https://docs.pinecone.io/docs/limits

[27] Facebook Research. [n.d.]. About Sparse Vectors in FAISS. https://github.com/facebookresearch/faiss/issues/754

[28] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. 1994. Grouplens: An Open Architecture for Collaborative Filtering of Netnews. *CSCW* (1994), 175–186.

[29] Sebastian Schelter. [n.d.]. 50 million synthetic interactions. https://www.dropbox.com/s/96j7q6yd43lm3w1/synthetic-100000-50000-0.01.npz

[30] Sebastian Schelter, Stefan Grafberger, and Ted Dunning. 2021. HedgeCut: Maintaining Randomised Trees for Low-Latency Machine Unlearning. *SIGMOD* (2021), 1545–1557.

[31] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. 2020. Understanding and Benchmarking the Impact of GDPR on Database Systems. *PVLDB* (2020).

[32] Boglio Simone. [n.d.]. bogliosimone/similaripy. https://doi.org/10.5281/zenodo.2583851

[33] Spotify. [n.d.]. Spotify Million Playlist Dataset. https://www.aicrowd.com/challenges/spotify-million-playlist-dataset-challenge

[34] DigiChina Stanford University. [n.d.]. Internet Information Service Algorithmic Recommendation Management Provisions. https://digichina.stanford.edu/work/translation-internet-information-service-algorithmic-recommendation-management-provisions-opinon-seeking-draft/

[35] Benjamin Longxiang Wang and Sebastian Schelter. 2021. Efficiently Maintaining Next Basket Recommendations under Additions and Deletions of Baskets and Items. *Workshop on Online Recommender Systems and User Modeling at RecSys* (2021).

[36] Vincent Warmerdam. 2021. koaning.io: Beyond Broken. https://koaning.io/posts/beyond-broken/

[37] Yinjun Wu, Edgar Dobriban, and Susan B. Davidson. 2020. DeltaGrad: Rapid Retraining of Machine Learning Models. *ICML* (2020).

[38] Yinjun Wu, Val Tannen, and Susan B Davidson. 2020. Priu: A Provenance-based Approach for Incrementally Updating Regression Models. *SIGMOD* (2020), 447–462.

[39] Yahoo. [n.d.]. Yahoo songs. http://konect.cc/networks/yahoo-song/