“Amnesia” - A Selection of Machine Learning Models That Can Forget User Data Very Fast

Sebastian Schelter
New York University
sebastian.schelter@nyu.edu

ABSTRACT

Software systems that learn from user data with machine learning (ML) have become ubiquitous over the last years. Recent law requires organisations that process personal data to delete user data upon request (enacting the “right to be forgotten”).
However, it is not sufficient to merely delete the user data from databases. ML models that have been learnt from the stored data often resemble a lossy compressed version of the data. We therefore argue that the user data should also be removed from ML models due to potential privacy risks. Typically, this requires an inefficient and costly retraining of the affected ML models from scratch, where the training infrastructure has to re-access the original training data and redeploy the retrained model.
We address this performance and operational issue by formulating the problem of “decrementally” updating trained ML models to “forget” the data of a user, without the need to re-access the training data. We present efficient decremental update procedures for a selection of four popular ML algorithms, together with *Rust*-based single-threaded implementations.
In an experimental evaluation on nine real world datasets, we find that our decremental update approach is several orders of magnitude faster than model retraining in the majority of cases. We additionally describe parallel implementations of our update procedures in Differential Dataflow and discuss the limitations of our approach.

1. INTRODUCTION

Software systems that learn from user data with machine learning (ML) have become ubiquitous over the last years [24], and participate in many critical decision making processes, e.g., to decide about loans, job applications and medical treatments. Recent laws such as the General Data Protection Regulation (GDPR) and the “right to be forgotten” require companies and institutions that process personal data to delete user data upon request [11]. We argue that it is not sufficient to merely delete this user data from primary data stores such as databases. ML models that have been learnt from the stored data resemble a lossy compressed version of the data in many cases. Not updating these models might introduce a privacy risk, e.g., when the ML model can be leveraged to make elaborate guesses about the deleted data (or even to restore it). We discuss an example of such a case in Section 2.
While relational databases offer transactional deletes and corresponding updates for materialized views [14] over the data, there exists no such automated deletion mechanism for ML models derived from the data. Instead, the deletion of user data typically requires an inefficient and costly retraining of the affected ML models from scratch on the original training data, for example on a nightly basis. This retraining is especially complicated from an operational perspective, as the training infrastructure has to reaccess the training data, and redeploy the retrained model afterwards. We propose to alleviate this problem by describing how to “decrementally” update a selection of trained ML models in an efficient way, such that they “forget” the user data. In other terms, the decrementally updated model is identical to a model that would have been trained from scratch without the data of the user to remove.
We formalize this problem in Section 3, and describe efficient decremental update procedures for four ML algorithms from different ML domains (recommendation, classification, regression) in Section 3.2. We experimentally evaluate our approach on nine real world datasets in Section 6. Our approach is applicable to a family of ML models with non-iterative training, which exhibit sparse computational dependencies [3, 9] between the model and the data. We discuss the generality of our approach in Section 7.

In summary, we provide the following contributions:

- We introduce the problem of making trained ML models “forget” user data via decremental updates without reaccessing the training data (Section 3).
- We describe corresponding decremental update procedures for four algorithms from different ML domains (recommendation, classification, regression) (Section 3.2).
- We provide single-threaded and dataflow implementations our proposed algorithms (Section 4).
- We conduct an experimental evaluation on nine real world datasets, and find that our decremental update is several orders of magnitude faster than model retraining in the majority of cases (Section 6).

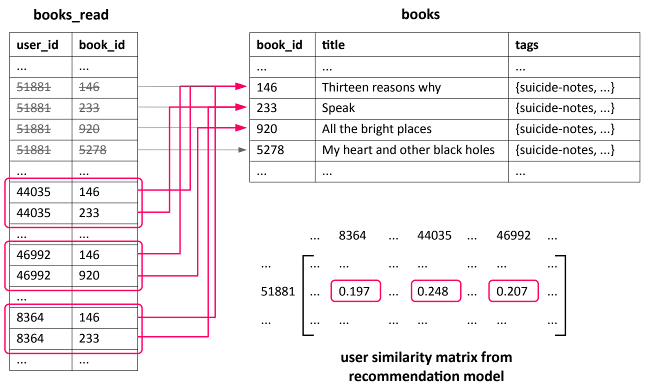
10th Annual Conference on Innovative Data Systems Research (CIDR ‘20) January 12-15, 2020, Amsterdam, Netherlands.

2. NEGATIVE PRIVACY IMPLICATIONS

We outline the potential privacy implications posed by not updating ML models after data removal on a fictitious example scenario, which leverages real world data. The *Goodbooks* dataset[1] contains six million ratings for ten thousand books given by fifty thousand (deidentified) users. Consider user `51881` who has read the following four books: *Thirteen Reasons Why*, *Speak*, *All the Bright Places*, *My Heart and Other Black Holes*. This information is potentially sensitive as all the books are dealing with the topic of teenage suicide (and are tagged with `suicide-notes` in the dataset). This user (or the parents of the user) might ask the data owner to remove this sensitive information from their database, e.g., under the legal umbrella of the "right-to-be-forgotten" as part of the European General Data Protection Regulation (GDPR) [11]. The data owner would implement this by removing the records for the user from the `books_read` table as shown in Figure 1.

However, the data owner might still have ML models that have been learnt from the version of the database prior to the removal of our user's records. An example could be a recommendation model built with user-based collaborative filtering [17]. This model would maintain a matrix of similarities between pairs of users, which can be leveraged to recommend books to other users (e.g., by choosing books from the reading histories of highly similar users). The similarity of two users in this scenario could for example be computed as the Jaccard similarity between their sets of reading histories.

Unfortunately, such a user similarity matrix can be used to make elaborate guesses about the (already deleted) reading history of user `51881` as we show in Figure 1. The mean similarity of our user to other users (as measured by a random sample of 100 users) is only about 0.04. We can however identify other users with significantly higher similarity by inspecting the recommendation model: examples are user `8364` with a similarity of 0.197, user `44035` with a similarity of 0.248 or user `46992` with a similarity of 0.207. If we inspect the intersection of the reading histories of these highly similar users, we again encounter many occurrences of the sensitive books that were contained in the deleted reading history of user `51881`: three occurrences of *Thirteen Reasons Why*, two occurrences of *Speak* and one occurrence of *All the Bright Places*. Based on this observation, we can conclude that it is very likely that these books also existed in the (now deleted) reading history of user `51881` and that the high similarity in the recommendation model is a result of the overlapping interest in books about teenage suicide.



user_id	book_id
...	...
~~51881~~	~~146~~
~~51881~~	~~233~~
~~51881~~	~~920~~
~~51881~~	~~5278~~
...	...
44035	146
44035	233
...	...
46992	146
46992	920
...	
8364	146
8364	233
...	...

book_id	title	tags
...	...	...
146	Thirteen reasons why	{suicide-notes, ...}
233	Speak	{suicide-notes, ...}
920	All the bright places	{suicide-notes, ...}
5278	My heart and other black holes	{suicide-notes, ...}
...	...	...

$$\begin{array}{c|ccccccc} & \dots & 8364 & \dots & 44035 & \dots & 46992 & \dots \\ \hline \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 51881 & \dots & \boxed{0.197} & \dots & \boxed{0.248} & \dots & \boxed{0.207} & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{array}$$

Figure 1: Example scenario for negative privacy implications of not updating trained ML models after data deletion. A recommendation model of user similarities enables elaborate guesses about sensitive deleted data (a reading history of books about teenage suicide).

3. APPROACH

In order to address the previously described privacy risks, we develop methods to efficiently update trained ML models after data removal. We first formalize the problem of efficient "decremental" updates for having models "forget" data, and describe decremental (and incremental) variants of four popular ML algorithms in detail afterwards.

3.1 Problem Statement

Let $t_{\text{learn}}(\mathbf{D}, \theta)$ denote a procedure to learn an ML model f (such as a classifier) from training data $\mathbf{D}$ with hyperparameters θ. Let $\mathbf{D} = \{\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_m\}$ denote the data of m users (our training data), and let $f = t_{\text{learn}}(\mathbf{D}, \theta)$ denote the ML model learnt from the user data $\mathbf{D}$. Now, assume that we are required to delete the data $\mathbf{d}_m$ of the m-th user from the model. That means we are interested in obtaining another ML model $f' = t_{\text{learn}}(\mathbf{D} \setminus \{\mathbf{d}_m\}, \theta)$, which never saw the data $\mathbf{d}_m$ of user m during its training. This new model can be trivially obtained by repeating the training procedure t_{learn} on $\mathbf{D} \setminus \{\mathbf{d}_m\}$.

Conducting a complete model retraining might however be inefficient and costly in many cases, and we can assume that the loss of data from a single user (or a handful of users) does not significantly change the predictive power of the model, or require a different choice of hyperparameters. Instead, we would be interested in an efficient procedure t_{forget} that can inspect $\mathbf{d}_m$ and update the existing model f to the desired model f', such that $f' = t_{\text{forget}}(f, \{\mathbf{d}_m\}, \theta) = t_{\text{learn}}(\mathbf{D} \setminus \{\mathbf{d}_m\}, \theta)$. This approach is refered to as "decremental learning" in the ML literature [5].

In summary, the decremental update procedure t_{forget} must satisfy the following property, and provide a runtime for t_{forget} that is much smaller than the runtime of t_{learn} for retraining:

$$t_{\text{forget}}(t_{\text{learn}}(\mathbf{D}, \theta), \{\mathbf{d}_m\}, \theta) = t_{\text{learn}}(\mathbf{D} \setminus \{\mathbf{d}_m\}, \theta)$$

This approach bears some resemblance to online learning, where we incrementally update an existing model with new observations. A major difference is that we not only need a merge operation to integrate updates into the global result, but also an inverse operation to remove such updates. This is problematic for learning algorithms such as gradient descent (which naturally supports online learning), as they compute a fixpoint using a series of global aggregations on all input tuples in every step, where the result of each iteration depends on all tuples and all previous results.

Therefore, we focus on non-iterative ML algorithms which exhibit sparse computational dependencies [9, 27]. We further discuss the generality of our approach in Section 7.

[1]http://fastml.com/goodbooks-10k

3.2 Decremental / Incremental Models

Our approach is based on the general idea of having the training algorithm for an ML model retain intermediate results during the model computation, which can be efficiently updated in both an incremental manner (to incorporate new user data) as well as a decremental manner (to remove user data) later on. In the following, we detail three algorithms for different ML tasks (recommendation, regression, classification) and describe efficient incremental / decremental update procedures for them. For each model, we describe the decremental update via the FORGET function, the incremental update via the PARTIAL-FIT function, and detail how to derive predictions from it via the PREDICT function.

3.2.1 Item-Based Collaborative Filtering (Recommender Systems)

Item-based collaborative filtering [23, 25] is a popular recommendation approach, which compares user interactions to find related items in the sense of: 'people who like this item also like these other items'. In a movie recommendation case for example, the system records which movies often cooccur in the viewing histories of users. These pairs of cooccurring movies are then ranked and form the basis for recommendations later on. In its simplest form, the input data for an item-based recommender comprises of a binary history matrix $\mathbf{H} \in \{0,1\}^{|U| \times |I|}$ which denotes the observed interactions between a set of users U and a set of items I. An entry H_{ji} equals 1 if user j interacted with item i, and 0 otherwise. Such binary observation data can be easily gathered by recording user actions (such as purchases in online shops or plays of a video on a movie platform).

Model & Prediction. The model of an item-based recommender comprises of a similarity matrix $\mathbf{S} \in \mathbb{R}^{|I| \times |I|}$ which denotes the interaction similarity between pairs of items. A common way to train this model is to first compute a cooccurrence matrix $\mathbf{C} = \mathbf{H}^\top \mathbf{H}$ which denotes how many users interacted with each pair of items. Additionally, we need a vector $\mathbf{n} = \sum_{j \in U} \mathbf{H}_j$ denoting the number of interactions per item (the row sums of $\mathbf{H}$). Next, we can compute the similarity matrix $\mathbf{S}$ by inspecting the cooccurrence counts. Many similarity measures between items can be computed from the cooccurrence matrix [25]. A popular choice is the Jaccard similarity $S_{i_1 i_2}$ between items i_1 and i_2, computed via $S_{i_1 i_2} = C_{i_1 i_2} / (n_{i_1} + n_{i_2} - C_{i_1 i_2})$. Recommendations can be retrieved by querying the similarity matrix $\mathbf{S}$. We retrieve item-to-item recommendations by querying for the most similar items per item (e.g., by invoking the PREDICT function in Algorithm 1), and generate items to recommend for a particular user by computing preference estimates via a weighted sum between item similarities and the corresponding user history [23].

Decremental and incremental updates. We require the following intermediate data structures to enable incremental and decremental updates for the item-based recommender: the item cooccurrence matrix $\mathbf{C}$, the item interaction count vector $\mathbf{n}$ and the item similarity matrix $\mathbf{S}$.

The FORGET function in Algorithm 1 details how to remove the data of the u-th user (which corresponds to the u-th row $\mathbf{H}_u$ of the user-item interaction matrix $\mathbf{H}$) from the model. We loop over all pairs of items (i_1, i_2) in the user history $\mathbf{H}_u$ and decrement the corresponding cooccurrence count $C_{i_1 i_2}$. Finally, we iterate over each item i_1 in the user history and recompute its corresponding row $\mathbf{S}_{i_1}$ in the similarity matrix. The incremental update of the model, illustrated in the PARTIAL-FIT function in Algorithm 1 works analogously, with the difference that we increment (instead of decrement) the cooccurrence counts.

Space and complexity. In general, an item-based recommendation model consists of a similarity matrix $\mathbf{S} \in \mathbb{R}^{|I| \times |I|}$ which requires space quadratic in the number of items. The intermediate data structures for the incremental/decremental variants double the required memory, as we additionally need to maintain the cooccurrence matrix $\mathbf{C} \in \mathbb{N}^{|I| \times |I|}$ and the vector $\mathbf{n} \in \mathbb{N}^{|I|}$. An update requires $|\mathbf{H}_u|^2$ adjustments of the cooccurrence matrix, and (in the worst case) the recomputation of $|\mathbf{H}_u| \cdot |I|$ entries in the similarity matrix $\mathbf{S}$, which means that an update has a quadratic complexity of $\mathcal{O}(|I|^2)$ in the worst case. The vast majority of users will however have only interacted with a very small number of items in real world data. Additionally, both the memory required for the intermediate data structures as well as the update complexity can be reduced by only retaining the top-k entries per item in $\mathbf{S}$ and by introducing bounds on the maximum number of interactions to consider per user and item, an approach which we discussed and evaluated in detail in previous work [26].

Algorithm 1 Decremental / incremental update procedures for an item-based recommendation model.

```
Input: item cooccurrence matrix $\mathbf{C}$, item interaction counts $\mathbf{n}$,
    item similarity matrix $\mathbf{S}$, user history $\mathbf{H}_u$

 1: function PARTIAL-FIT($\mathbf{H}_u$, $\mathbf{C}$, $\mathbf{n}$, $\mathbf{S}$)
 2:     $\mathbf{n} \leftarrow \mathbf{n} + \mathbf{H}_u$
 3:     for item pair $(i_1, i_2) \in \mathbf{H}_u$ do
 4:         $C_{i_1 i_2} \leftarrow C_{i_1 i_2} + 1$
 5:     for item $i_1 \in \mathbf{H}_u$ do
 6:         for item $i_2 \in \mathbf{C}_{i_1}$ do
 7:             $S_{i_1 i_2} \leftarrow C_{i_1 i_2} / (n_{i_1} + n_{i_2} - C_{i_1 i_2})$

 8: function FORGET($\mathbf{H}_u$, $\mathbf{C}$, $\mathbf{n}$, $\mathbf{S}$)
 9:     $\mathbf{n} \leftarrow \mathbf{n} - \mathbf{H}_u$
10:     for item pair $(i_1, i_2) \in \mathbf{H}_u$ do
11:         $C_{i_1 i_2} \leftarrow C_{i_1 i_2} - 1$
12:     for item $i_1 \in \mathbf{H}_u$ do
13:         for item $i_2 \in \mathbf{C}_{i_1}$ do
14:             $S_{i_1 i_2} \leftarrow C_{i_1 i_2} / (n_{i_1} + n_{i_2} - C_{i_1 i_2})$

15: function PREDICT($j$, $k$, $\mathbf{S}$)
16:     return top-$k$ items from $\mathbf{S_j}$
```

Restoration of deleted data from a stale model. Next, we discuss how deleted user data could be restored from a stale model. We look at the case where the data of a single user has been deleted from the database. That means that the interactions data now consists of a matrix $\hat{\mathbf{H}}$ which is the original database $\mathbf{H}$ with the row corresponding to the deleted user removed. If we still have access to the (now stale) model $\mathbf{S}$ computed from $\mathbf{H}$, we can restore the deleted user data as follows. We compute the similarity matrix $\hat{\mathbf{S}}$ corresponding to the updated database $\hat{\mathbf{H}}$, and compare it to the stale model $\mathbf{S}$. We identify all items i for which we observe differences in entries of the similarity matrices (e.g., $\exists j \; \mathbf{S}_{ij} \neq \hat{\mathbf{S}}_{ij}$). The set of these items i is exactly the items that were contained in the interaction history of the deleted user.

3.2.2 *Ridge Regression (Regression)*

Ridge regression [15] is a widely used technique to predict a continuous target variable in regression scenarios. The input data to the regression model is a matrix $\mathbf{X} \in \mathbb{R}^{m \times d}$ of m d-dimensional observations, and a corresponding numeric target variable $\mathbf{y} \in \mathbb{R}^{m}$. Without loss of generality, we assume that the data of a particular user i is captured in the i-th row vector $\mathbf{X}_i$ of the input (if more than one row would denote data about a particular user, we could simply run the decremental update procedure several times).

Model & Prediction. A common way to compute a ridge regression model in the form of the weight vector $\mathbf{w}$ is to solve the normal equation $\mathbf{w} = (\mathbf{X}^\top\mathbf{X}+\lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$. The regularization constant λ is typically selected via cross-validation. The regression estimate for a new observation $\mathbf{x}_{\text{new}}$ can be computed via its dot product with the weight vector: $\hat{y}_{\text{new}} = \mathbf{w}^\top\mathbf{x}_{\text{new}}$ (as shown in the PREDICT function in Algorithm 2).

Decremental and incremental updates. We detail how to remove the data of a user u (in the form of the u-th row $\mathbf{X}_u$ of the observation matrix $\mathbf{X}$) from a ridge regression model. We aim for an efficient way to compute:

$$\mathbf{w} = (\mathbf{X}^\top\mathbf{X} - \mathbf{X}_u^\top\mathbf{X}_u + \lambda\mathbf{I})^{-1}(\mathbf{X}^\top\mathbf{y} - \mathbf{X}_u y_u)$$

We therefore maintain the following intermediates from the computation: the vector $\mathbf{z} = \mathbf{X}^\top\mathbf{y}$ and a QR factorization[2] $\mathbf{QR} = \text{qr}(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})$ of the regularized gram matrix. We can now recompute $\mathbf{z}$ by subtracting $\mathbf{X}_u y_u$, and we invoke a fast rank-one update algorithm [30] with $-\ \mathbf{Q}^T\mathbf{X}_u$ and $\mathbf{X}_u$ as argument to update the QR decomposition $\mathbf{Q}$, $\mathbf{R}$. Afterwards, we can efficiently solve for the updated model $\mathbf{w}$ (see the FORGET function in Algorithm 2). If we substitute the subtractions with additions, this gives us an incremental update algorithm as well, as illustrated by the PARTIAL-FIT function of Algorithm 2.

Space and complexity. The decremental variant of the model requires us to maintain two additional matrices $\mathbf{Q} \in \mathbb{R}^{d \times d}$ and $\mathbf{R} \in \mathbb{R}^{d \times d}$ as well as the vector $\mathbf{z} \in \mathbb{R}^{d}$. This means that the required space is quadratic in the number of features d (which is typically much smaller than the number of examples m), but independent of the number of examples. An update requires $2d$ operations to scale and add to $\mathbf{z}$, d^2 operations for the matrix vector multiplication $\mathbf{Q}^T\mathbf{X}_u$, $26d^2$ operations for the rank-one QR update [30], another d^2 operations for the matrix vector multiplication $\mathbf{Q}^T\mathbf{z}$ and again d^2 operations for solving for $\mathbf{w}$ via back substitution. In summary, an update has a complexity of $\mathcal{O}(d^2)$, which is an improvement over retraining which requires $\mathcal{O}(md^2)$ operations.

Restoration of deleted data from a stale model. In contrast to the other approaches presented here, it is more challenging to infer information about a deleted user feature vector $\mathbf{X}_d$ from a regression model $\mathbf{w}$. If we still had access to the complete target variable $\mathbf{y}$, we could constrain candidate vectors to the subspace defined by $\mathbf{w}^\top\mathbf{X}_d = y_d$ (with some uncertainty due to the contained prediction error). Additional knowledge about $\mathbf{X}$ (e.g., about the ranges of certain features or about row-wise normalisation techniques), would further restrict the space of candidate vectors.

[2]The $\mathbf{QR}$-factorisation is a decomposition of a matrix $\mathbf{A}$ into a product $\mathbf{A} = \mathbf{QR}$ of an orthogonal matrix $\mathbf{Q}$ and an upper triangular matrix $\mathbf{R}$, which is particularly useful to solve linear least squares problems [30].

Algorithm 2 Decremental / incremental update procedures for a ridge regression model.

```
Input: $\mathbf{z} \leftarrow \mathbf{X}^\top\mathbf{y}$, $\mathbf{QR} \leftarrow \text{qr}(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})$, user observation $\mathbf{X}_u$

 1: function PARTIAL-FIT($\mathbf{X}_u$, $y_u$, $\mathbf{Q}$, $\mathbf{R}$, $\mathbf{z}$)
 2:     $\mathbf{z} \leftarrow \mathbf{z} + \mathbf{X}_u y_u$
 3:     $\mathbf{QR} \leftarrow \text{qr\_update}(\mathbf{Q}, \mathbf{R}, \mathbf{Q}^\top\mathbf{X}_u, \mathbf{X}_u)$
 4:     solve $\mathbf{Rw} = \mathbf{Q}^\top\mathbf{z}$ for $\mathbf{w}$

 5: function FORGET($\mathbf{X}_u$, $y_u$, $\mathbf{Q}$, $\mathbf{R}$, $\mathbf{z}$)
 6:     $\mathbf{z} \leftarrow \mathbf{z} - \mathbf{X}_u y_u$
 7:     $\mathbf{QR} \leftarrow \text{qr\_update}(\mathbf{Q}, \mathbf{R}, -\mathbf{Q}^\top\mathbf{X}_u, \mathbf{X}_u)$
 8:     solve $\mathbf{Rw} = \mathbf{Q}^\top\mathbf{z}$ for $\mathbf{w}$

 9: function PREDICT($\mathbf{x}_{\text{new}}$,$\mathbf{w}$)
10:     return $\mathbf{w}^\top\mathbf{x}_{\text{new}}$
```

3.2.3 *k-Nearest Neighbors with Locality Sensitive Hashing (Classification)*

Our next example leverages a nearest neighbor-based classification algorithm, which assigns the label of the k nearest neighbors to an unknown observation. It applies a common approximation technique to speed up the search for the nearest neighbors called locality sensitive hashing (LSH) [13]. The input data for the classifier comprises of a matrix $\mathbf{X} \in \mathbb{R}^{m \times d}$ of m d-dimensional observations, and the target variable $\mathbf{y} \in \{1, \ldots, c\}^m$ which denotes the assignments of the corresponding c categorical labels. We again assume that each row $\mathbf{X}_i$ corresponds to observations for a particular user i.

Model & Prediction. The algorithm leverages approximate similarity search in a high-dimensional space by building an index over the data. This index comprises of several hash tables where observations which are close in terms of Euclidean distance have a high chance of ending up in the same hash bucket. We leverage random projections as hash function to compute the bucket indexes. We compute H hash tables $T_1, \ldots, T_H$ with b-dimensional bucket indexes. We generate a Gaussian random matrix $\mathbf{\Omega}_h$ for each hash table T_h, and leverage this matrix to conduct a random projection of our data via $\text{sgn}(\mathbf{X}\mathbf{\Omega}_h)$. The resulting bit vectors denote the hash keys, and we assign each row $\mathbf{X_i}$ from $\mathbf{X}$ to its corresponding bucket with key $\text{sgn}(\mathbf{X}_i\mathbf{\Omega}_h)$ in the hash table T_h. In order to assign a label to an unseen observation $\mathbf{X}_{\text{new}}$, we collect its nearest neighbors as follows. For each hash table T_h, we compute the bucket index $b_h = \text{sgn}(\mathbf{X}_{\text{new}}\mathbf{\Omega}_h)$, and collect all observations from this bucket. Next, we compute the k exact nearest neighbors of $\mathbf{X}_{\text{new}}$ in the retrieved observations, and assign the majority label from these as the predicted label $\hat{y}_{\text{new}}$ to $\mathbf{X}_{\text{new}}$. This can be efficiently implemented by maintaining a binary heap with the k closest examples, as shown in the PREDICT function of Algorithm 3.

Decremental and incremental updates. Removing an existing observation $\mathbf{X}_\text{u}$ for a user u from our index works as depicted in the FORGET function of Algorithm 3. We compute the bucket index b_{uh} of $\mathbf{X}_\text{u}$ via the random projection $\text{sgn}(\mathbf{X}_\text{u}\mathbf{\Omega}_h)$, and remove $\mathbf{X}_u$ from bucket b_{uh} in hash table T_h. We repeat this procedure for all hash tables. An incremental update works analogously, as listed in the PARTIAL-FIT function in Algorithm 3.

Space and complexity. Our approach requires no additional space, as it simply leverages the hash tables $T_1, \ldots, T_h$ and the projection matrices $\mathbf{\Omega}_1, \ldots, \mathbf{\Omega}_h$, which are required for deriving predictions from the model anyways. A decremental update requires H matrix-vector multiplications between a projection matrix of size $d \times b$ and the feature vector of dimensionality d for determining the bucket indices, plus H subsequent hashmap inserts, leading to a complexity of $\mathcal{O}(Hdb)$.

Algorithm 3 Decremental / incremental update procedures for approximate k-NN.

```
Input: hash tables $T_1, \ldots, T_h$, projection matrices $\mathbf{\Omega}_1, \ldots, \mathbf{\Omega}_h$,
    observation $\mathbf{X}_u$

 1: function PARTIAL-FIT($\mathbf{X}_u$, $\mathbf{\Omega}$, $T$)
 2:     for $h = 1 \ldots H$ do
 3:         $b_{uh} = \text{sgn}(\mathbf{X}_u \mathbf{\Omega}_h)$
 4:         add $\mathbf{X}_u$ to bucket $b_{uh}$ in hash table $T_h$

 5: function FORGET($\mathbf{X}_u$, $\mathbf{\Omega}$, $T$)
 6:     for $h = 1 \ldots H$ do
 7:         $b_{uh} = \text{sgn}(\mathbf{X}_u \mathbf{\Omega}_h)$
 8:         remove $\mathbf{X}_u$ from bucket $b_{uh}$ in hash table $T_h$

 9: function PREDICT($\mathbf{x}_{\text{new}}$, $\mathbf{\Omega}$, $T$)
10:     $P \leftarrow$ heap with capacity $k$ for closest examples
11:     for $h = 1 \ldots H$ do
12:         $b_{uh} = \text{sgn}(\mathbf{X}_u \mathbf{\Omega}_h)$
13:         update $P$ for all examples from bucket $b_{uh}$
14:     return majority label from $k$ examples in $P$
```

Restoration of deleted data from a stale model. The restoration of deleted data from the model is trivial as the model stores copies of the feature vectors.

3.2.4 Multinomial Naive Bayes (Classification)

The last algorithm we discuss is again a classification algorithm called Naive Bayes [22], where we focus on a variant for categorical data called Multinomial Naive Bayes (MNB). The input data for the classifier comprises of a matrix $\mathbf{X} \in \mathbb{N}^{m \times d}$ of m d-dimensional observations, and the target variable $\mathbf{y} \in \{1, \ldots, c\}^m$ which denotes the assignments of the corresponding c categorical labels. We again assume that each row $\mathbf{X}_i$ corresponds to observations for a particular user i.

Model & Prediction. The Naive Bayes model is built on the assumption that features are conditionally independent, given the class label. If we use a uniform prior estimate for the sake of simplicity, then the MNB classifier assigns its class prediction $\hat{y}$ as follows: $\hat{y} = \text{argmax}_y \sum_j \log X_{ij} \theta_{yj}$, where the probability $P(x_j = k|y) = \theta_{yi}$ of encountering the value k for feature j in class y is estimated using a smoothed version of the maximum likelihood estimate $\hat{\theta}_{yj} = \frac{N_{yj} + \alpha_j}{N_y + \alpha}$. Here, N_{yj} denotes the number of times the value for feature j occurs in class y, $N_y = \sum_j N_{yj}$, a uniform prior is used for the sake of simplicity, α_j is the smoothing term per feature, and α the sum of the smoothing terms.

Decremental and incremental updates. Adding or removing a user feature vector $\mathbf{X}_\text{u}$ with a given class label y for MNB is trivial, as we just need to increment or decrement the corresponding feature counts N_{yj} and N_y for all non-zero features with indexes $j \in \mathbf{X}_\text{u}$. Note that this approach also works for the Gaussian variant of Naive Bayes for continuous data, where we have to maintain the mean and variance per feature and class label instead of the raw counts.

Space and complexity. Our incremental/decremental variant requires us to maintain the raw feature counts $\mathbf{N}$ with space in $O(dc)$ which requires the same space as the probability vectors θ. The number of operations for incremental and decremental updates linear in the number of non-zero features of the vector $\mathbf{X}_\text{u}$ to add or remove.

Algorithm 4 Decremental / incremental update procedures for Multinomial Naive Bayes .

```
Input: conditional feature counts $\mathbf{N}$

 1: function PARTIAL-FIT($\mathbf{X}_u$, $c_u$, $\mathbf{N}$)
 2:     for $(i, X_{uj}) \in \mathbf{X}_u$ do
 3:         $N_{y_u j} \leftarrow N_{y_u j} + X_{uj}$
 4:         $N_{y_u} \leftarrow N_{y_u} + X_{uj}$

 5: function FORGET($\mathbf{X}_u$, $\mathbf{\Omega}$, $T$)
 6:     for $(i, X_{uj}) \in \mathbf{X}_u$ do
 7:         $N_{y_u j} \leftarrow N_{y_u j} - X_{uj}$
 8:         $N_{y_u} \leftarrow N_{y_u} - X_{uj}$

 9: function PREDICT($\mathbf{x}_{\text{new}}$, $\mathbf{N}$)
10:     return $\text{argmax}_y \sum_{(j, x_j) \in \mathbf{x}_{\text{new}}} \log x_j \frac{N_{yj} + \alpha_j}{N_y + \alpha}$
```

Restoration of deleted data from a stale model. We again discuss how deleted user data could be restored from a stale model, and focus on the simplified case where the data of a single user has been deleted from the database. In this case, we can recompute all paramater estimates $\hat{\theta}_{yj}^{(new)}$ and detect the features contained in the deleted data by identifying all differences $\hat{\theta}_{yj}^{(new)} \neq \hat{\theta}_{yj}$ in the new and old parameter estimates.

4. IMPLEMENTATION

We provide single-threaded and parallel implementations of our proposed algorithms in *Rust*.[3]

Single-Threaded. We depend on the `fnv` crate for fast hashmaps, the `ndarray` crate with bindings to *OpenBLAS* for the random projections in approximate k-NN, as well as on the `GSL` crate bindings to the *GNU Scientific Library* which provides the routines for updating the QR factorizations required for ridge regression. Note that we implement mini-batch versions of the PARTIAL-FIT function for efficient retraining and apply a 'user interaction cut' [26] of 500 interactions to our recommender implementation.

Differential Dataflow. We additionally implement dataflow versions of our algorithms in Differential Dataflow [20, 19] (DD) to enable parallelisation and distribution, as well as to benefit from its automatic incrementalisation and decrementalisation. We find that three out of four of the algorithms fit well into the dataflow model.

A simplified implementation of our incremental/decremental item-based recommender is shown in the following. The most expensive operations is the matrix matrix multiplication $\mathbf{C} = \mathbf{H}^\top \mathbf{H}$ of the binary user interaction history matrix $\mathbf{H}$ to obtain the cooccurrence matrix $\mathbf{C}$. We implement this with a self-join followed by a count aggregation. We would

[3]https://github.com/schelterlabs/projects-amnesia

like to highlight that our actual implementation contains a set of performance improvements not included in the listing, e.g., the use of 'arranges' [18] to re-use intermediate results, the application of an interaction-cut [26] to handle 'power users', and thresholding of the similarity matrix.

```
// Maintain interactions per item n
let num_interactions_per_item = interactions
 .map(|(_user, item)| item)
 .count_total();

// Maintain the cooccurence matrix C
let cooccurrences = interactions
 .join(&interactions,
      |_user, &item_a, &item_b| (item_a, item_b))
 .count();

// Maintain the similarity matrix S
let jaccard_similarities = cooccurrences
 .map(|record|, key_by_item_a(record))
 // Find the number of interactions for item a
 .join(&num_interactions_per_item,
      |record| collect_for_item_a(record))
 .map(|record|, key_by_item_b(record))
 // Find the number of interactions for item b
 .join(&num_interactions_per_item,
      |record| collect_for_item_b(record))
 // Compute Jaccard similarty
 .map(|record| compute_jaccard(record));
```

Listing 1: Implementation of the proposed item-based recommender in DD (simplified).

For the k-nn classifier, we form the cartesian product between the projection matrices and features, and execute the projections afterwards to obtain the bucket keys. The Naive Bayes implementation only requires us to maintain grouped counts by label and feature per label.

```
let features_per_label =
 examples.explode(|example| {
  let label = example.label;
  example.features.into_iter()
   .map(|(index, value)| ((index, label), value)));

let feature_per_label_counts =
 features_per_label.count();

let label_counts = features_per_label
 .map(|(_, label)| label)
 .count();
```

Listing 2: Implementation of the proposed multinomial naive bayes classifier in DD (simplified).

5. RELATED WORK

Ensuring that data processing technology adheres to legal and ethical standards in a fair and transparent manner [29] is an important research direction, which starts to gain attention from law makers and governments [10]. The machine learning community has pioneered some work on removing data from models under the umbrella of "decremental learning" for support vector machines [5, 16].
In contrast to our work, these approaches need to re-access the training data for updating the model though, which introduces operational complexity as model training (and the corresponding data) and serving is typically handled in different systems and infrastructures [21]. Cao et al. [4] propose an approach similar to ours to remove adversarial inputs from ML models, however they again reaccess the training data for model updates. For that reason, their approach also works for iterative, gradient-descent based learning algorithms, as they basically just restart gradient descent from the iteration where the sample to forget was used first. They adopt a popular summation form of ML algorithms (developed as a general model to parallelize training on multicore architectures), which for example does not capture our QR factorization-based update methods for linear regression well. Ginart et. al. [12] explore a problem setting similar to ours for stochastic algorithms, in particular for variants of k-Means clustering.
An orthogonal technique to protect the privacy of user data in machine learning use cases is differential privacy [8]. However, it is very difficult to design differentially private algorithms (even for experts [7]), and this approach requires a difficult decision on the limit of the acceptable privacy loss in practice.
The security community studies inference attacks against ML models, e.g., to infer whether a given record was part of the training data of a model [28] or to infer statistical information about the training data [2].
The approach used in this paper bears resemblance to general approaches for incremental data processing [9, 19, 27, 14] with the difference that it requires inverse operations for removing data, which are not covered by many existing approaches.

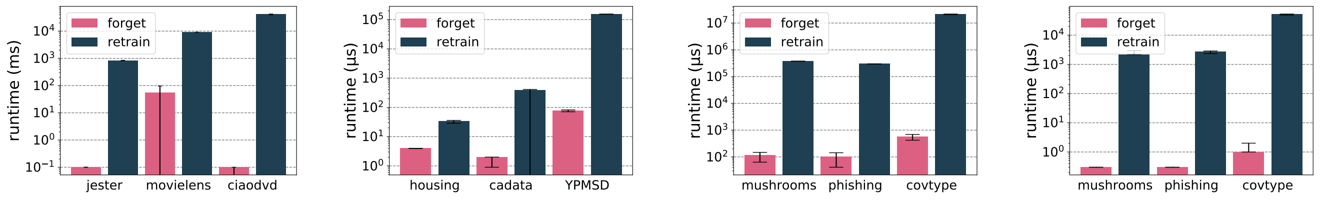
6. EVALUATION

Datasets. We run experiments[4] on nine real-world datasets, more precisely on three differently sized datasets for each proposed algorithm, which are detailed in Table 1. For our recommendation algorithm, we leverage joke ratings (*jester*), movie ratings (*movielens*), and DVD ratings (*ciaodvd*), which are publicly available via the *Konect* network repository.[5] We use data about *mushrooms*, *phishing* websites, and cartographic forest data (*covtype*) for our classification experiments. Finally, we train our regression models on data about house prices (*housing*, *cadata*) and music (*YearPredictionMSD*). The classification and regression datasets are available via the *LibSVM* dataset repository.[6]

6.1 Benefits of Decremental Updates

Setup. We experiment using our single-threaded implementations in Rust 1.38 on a machine with an Intel i7-7700HQ CPU and 16GB of RAM, running Ubuntu Linux 16.04. We load the input data into memory and train a model on each dataset. Next, we measure the time to update the model to "forget" a random user versus the time to retrain the model from scratch without the data of that particular user. We repeat every experiment for twenty randomly chosen users and report the median runtime as well as the 90th and 10th percentile of the runtime distribution in Figure 2. Note that the runtime on the y-axis is displayed on a logarithmic scale, and that the runtimes for retraining are overly optimistic as we do not include the time to parse and read the training data into the measurements (which would depend on the infrastructure).

[4]Code to reproduce our experiments is available at https://github.com/schelterlabs/projects-amnesia.
[5]http://konect.cc/networks/
[6]https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/



(a) Collaborative Filtering. (b) Ridge Regression. (c) k-Nearest Neighbors. (d) Multinomial Naive Bayes.

Figure 2: Comparison of the median runtimes for removing a random user's data from a trained ML model. "Retrain" retrains a model from scratch without this user's data, while "forget" decrementally updates the trained model to forget the user data. The runtime is displayed on a logarithmic scale and the error bars denote the 90-th and 10-th percentiles of the observed runtime distribution.

dataset	#examples	#features
mushrooms	8,124	112
phishing	11,055	68
covtype	581,012	54
housing	506	13
cadata	20,640	8
YearPredictionMSD	463,715	90

dataset	#interactions	#users	#items
jester	1,728,847	50,692	140
movielens	1,000,209	6,040	3,706
ciaodvd	1,625,480	21,019	71,633

Table 1: Datasets for evaluation.

Results. The results for our recommendation algorithm applied to the *jester*, *movielens*, and *ciaodvd* data are shown in Figure 2(a). The decremental update is three to five orders of magnitude faster than retraining for the *jester* and *ciaodvd* data, and two orders of magnitude faster for the *movielens* dataset, where the update time varies much stronger. This difference is due to the dependence of the update time on the user history length which is very low for the *jester* and *ciaodvd* data with 20 and 34 interactions on average, and much higher for the *movielens* data with a mean of 143 interactions.
The results for our ridge regression experiments on the *housing*, *cadata* and *YearPredictionMSD* datasets are illustrated in Figure 2(b). We observe that the update is one to two orders of magnitude faster than retraining for the small *housing* and *cadata* datasets, and more than three orders of magnitude faster for the *YearPredictionMSD* data, which has more than 500K observations. This illustrates the fact that the update runtime depends on the number of features only and is independent of the number of examples (as outlined Section 3.2). Therefore, we observe a larger runtime difference for datasets with more examples.
We show the results for the approximate k-NN model on the *mushrooms*, *phishing* and *covtype* datasets in Figure 2(c). We used 20 hash tables, set k to 10, and apply random projections as hashing function to approximate euclidean distance. The decremental update is more than three orders of magnitude faster in all cases, which is an expected result as the update just involves a fixed number of hash table modifications.

We show the results for the Multinomial Naive Bayes model on the *mushrooms*, *phishing* and *covtype* datasets in Figure 2(d). We binarized the input features. The decremental update is more than three orders of magnitude faster in all cases, and can often be conducted in less than a microsecond, which is an expected result as the update just involves a fixed number of updates to a dense array.
In short, the experimental results confirm that our algorithms satisfy the requirement for the decremental update procedures: the update is several orders of magnitude faster than retraining in the majority of cases, and can be conducted in sub-second time for the datasets from our evaluation.

6.2 Benefits of Parallelisation

Next, we evaluate the scalability our implementation in Differential Dataflow. We focus on the item-based recommender, as we found the other approaches to have update times of less than a millisecond in our previous experiments (which obliviates the need for parallelisation). We train a recommender model for the *jester*, *movielens*, and *ciaodvd* datasets, and measure the time to remove 100 randomly chosen user histories. We repeat each experiment 20 times for an increasing number of workers (threads), and plot the resulting runtimes in Figure 3. Note that we fix the random seeds for comparability as the runtime is extremely sensitive to the length of the (randomly) chosen user histories.

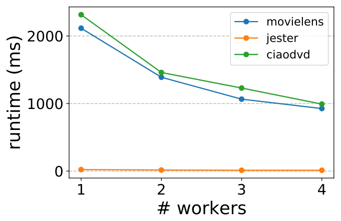


Figure 3: Scalability of our item-based recommender implementation in differential dataflow for a decremental update of 100 random users.

We observe that the parallelisation effectively reduces the runtime for the *movielens*, and *ciaodvd*. The updates for the *jester* dataset can be conducted in less than 25 milliseconds in all setups, due to the extremely low number of items in the dataset.

7. DISCUSSION

We discuss operational implications of our approach, and describe limitations in its generality as basis for further research.

7.1 Operational Implications

A major operational implication of our proposed approach is that our t_{forget} procedure does not require access to the original training data $\mathbf{D}$, which simplifies the integration of our approach into complex real-world ML deployments. In such deployments, model training typically requires an offline training process on a cluster or a powerful machine in separate infrastructure with access to the training data (which typically must not be accessible from serving systems for security reasons). After the completion of the training, the updated model has to be re-deployed to a model serving system [21, 1] via complex deployment pipelines. As our approach does not require access to the training data, we enable a setup with much lower operational complexity because the model can be updated in-place in the serving systems.
This operational perspective and the goal to avoid having to reaccess training data is a novel and differentiating factor of our approach in constrast to existing work which falls back to reaccessing training data and retraining models in most cases [5, 16, 4].

7.2 Generality of our Approach

Our approach is based on the idea of recomputing the parts of the model that have been affected by the user data which we aim to remove. This is efficient if (*i*) only a small part of the model is affected, and (*ii*) the recomputation is asymptotically cheaper than full retraining. These conditions are the basis for the experimental results we observed in the previous section. A general property that a model must have for our approach to work are so-called *sparse computational dependencies* [9, 27]. A formal way to reason about these is via a bipartite dependency graph $G(V_i, V_m, E)$ [3] where the vertex set V_i denotes partitions of our input data, the vertex set V_m denotes partitions of the final model, and an edge $(v_i, v_m) \in E$ between a vertex v_i and a vertex v_m denotes that the input i is necessary to compute the m-th part of the model. This graph allows us to determine which parts of the model need to be recomputed if we remove a particular input v_u. If the graph is sparse (e.g., the vertices in V_i have a low degree), only a small part of the model must be recomputed. If the graph was fully connected, the removal of a single input would trigger a complete retraining of the model.

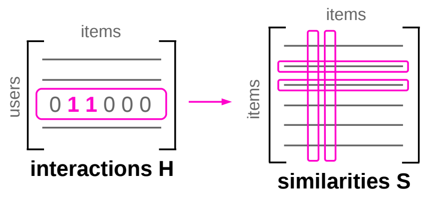


Figure 4: Illustration of the sparse computational dependencies in item-based collaborative filtering. Each (sparse) row of the input H affects only a fraction of the model S, indicated by the non-zero entries in the row.

The algorithms discussed in this paper exhibit such sparse computational dependencies: For approximate k-NN, the vertex set V_i comprises of the user observations (e.g., the rows of the input matrix $\mathbf{X}$, and the vertex set V_m consists of the buckets of the hash tables $T_1, ..., T_H$. The resulting dependency graph is sparse, because we hash each input $\mathbf{X}_u$ to H buckets only, which means the degree of each $v \in V_i$ is exactly H, and upon removal of an input, only H buckets must be updated.
The dependency graph between the inputs (the user histories $\mathbf{H}_u$) and the cooccurrences (entries of the cooccurrence matrix $\mathbf{C}$ for item-based collaborative filtering is sparse as well, as shown in Figure 4. Each similarity $\mathbf{S}_{ij}$ depends on all user histories where the item pair (i, j) occurs. That means that a single user history contributes to a quadratic number of cooccurrences (all possible item pairs). As a result, the resulting dependency graph is sparse as well, because the majority of users only interact with a very small subset of the overall items.
The dependency graph model also explains why our approach is not applicable to models learnt with gradient descent, as this learning algorithm exhibits *dense computational dependencies*. In general, we learn a supervised model $\mathbf{w}$ on labeled examples $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$ as follows using gradient descent:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \lambda \sum_{\mathbf{x},y} \nabla_{\mathbf{w}^{(t)}} \ell(\mathbf{x}, y, \mathbf{w}^{(t)})$$

We obtain $\mathbf{w}^{(t+1)}$ by updating $\mathbf{w}^{(t)}$ (according to the learning rate λ) in the negative direction of the gradient of the model's loss function, which we compute as the sum of the gradients over the losses ℓ of the individual examples $(\mathbf{x}, y)$. That means that each intermediate model $\mathbf{w}^{(t)}$ is dependent on all inputs with a non-zero gradient and recursively depends on all previous model versions $\mathbf{w}^{(t-1)}, \dots, \mathbf{w}^{(0)}$, giving rise to a dense computational dependency graph, as sketched in Figure 5.

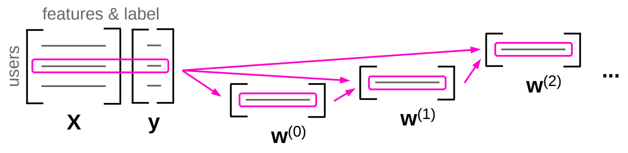


Figure 5: Illustration of the dense computational dependencies in stochastic gradient descent-based learning. Each model iterate $\mathbf{w}^{(t)}$ depends on the whole input and the previous iterate $\mathbf{w}^{(t-1)}$.

This illustrates that our approach works best in case of non-iterative learning procedures with sparse computational dependencies. While such methods are conceptually simpler than currently popular approaches (e.g., deep neural networks), they are nevertheless widely used in industry (e.g., in the Amazon SageMaker platform[7], which offers k-nearest neighbors, linear models, and k-Means). Furthermore, it has recently been shown [6] that classical itembased collaborative filtering outperforms recurrent neural networks for a wide variety of recommendation datasets.

[7]https://docs.aws.amazon.com/sagemaker/latest/dg/algos.html

7.3 Future Directions

The next directions for this line of research are two-fold: we will explore decremental updates for more ML algorithms, and extend the decremental updates to the whole ML pipeline (e.g., provide decremental/incremental data preprocessing and feature encoding operations). We additionally intend to explore changes to the training procedures of gradient-descent-based models to also enable decremental updates for them.

We would furthermore like to stress that challenges with respect to reliably deleting user data are not constrained to ML models and pipelines; they extend to the entire data management lifecycle, which often includes data replication when run in cloud environments.

Acknowledgements. *We would like to thank Frank McSherry and Ted Dunning for helpful comments on improving implementations and algorithms in this paper.*

8. REFERENCES

[1] P. Andrews, A. Kalro, H. Mehanna, and A. Sidorov. Productionizing machine learning pipelines at scale. *ML Systems workshop at ICML*, 2016.

[2] G. Ateniese, G. Felici, L. V. Mancini, A. Spognardi, A. Villani, and D. Vitali. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *arXiv preprint arXiv:1306.4447*, 2013.

[3] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods*, volume 23. Prentice hall Englewood Cliffs, NJ, 1989.

[4] Y. Cao and J. Yang. Towards making systems forget with machine unlearning. *IEEE Symposium on Security and Privacy*, pages 463–480, 2015.

[5] G. Cauwenberghs and T. Poggio. Incremental and decremental support vector machine learning. *NeurIPS*, pages 409–415, 2001.

[6] M. F. Dacrema, P. Cremonesi, and D. Jannach. Are we really making much progress? a worrying analysis of recent neural recommendation approaches. *RecSys*, pages 101–109, 2019.

[7] Z. Ding, Y. Wang, G. Wang, D. Zhang, and D. Kifer. Detecting violations of differential privacy. *CCS*, pages 475–489, 2018.

[8] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. *Theory of Cryptography*, pages 265–284, 2006.

[9] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *PVLDB*, 5(11):1268–1279, 2012.

[10] Executive Office of the President. White House Report on Big Data: Seizing Opportunities, Preserving Values, 2014.

[11] GDPR.eu. Recital 65: Right of rectification and erasure. https://gdpr.eu/recital-65-right-of-rectification-and-erasure

[12] A. Ginart, M. Y. Guan, G. Valiant, and J. Zou. Making AI forget you: Data deletion in machine learning. *CoRR*, abs/1907.05012, 2019.

[13] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. *VLDB*, pages 518–529, 1999.

[14] A. Gupta, I. S. Mumick, et al. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[15] A. E. Hoerl and R. W. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[16] M. Karasuyama and I. Takeuchi. Multiple incremental decremental learning of support vector machines. *NeurIPS*, pages 907–915, 2009.

[17] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl. Grouplens: applying collaborative filtering to usenet news. *Communications of the ACM*, 40(3):77–87, 1997.

[18] F. McSherry, A. Lattuada, and M. Schwarzkopf. K-pg: Shared state in differential dataflows, 2018.

[19] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. *CIDR*, 2013.

[20] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. *SOSP*, pages 439–455, 2013.

[21] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *ML Systems workshop at NeurIPS*, 2017.

[22] J. D. Rennie, L. Shih, J. Teevan, and D. R. Karger. Tackling the poor assumptions of naive bayes text classifiers. *ICML*, pages 616–623, 2003.

[23] B. M. Sarwar, G. Karypis, J. A. Konstan, J. Riedl, et al. Item-based collaborative filtering recommendation algorithms. *WWW*, 1:285–295, 2001.

[24] S. Schelter, F. Biessmann, T. Januschowski, D. Salinas, S. Seufert, and G. Szarvas. On challenges in machine learning model management. *Data Engineering*, 2018.

[25] S. Schelter, C. Boden, and V. Markl. Scalable similarity-based neighborhood methods with mapreduce. *ACM RecSys*, pages 163–170, 2012.

[26] S. Schelter, U. Celebi, and T. Dunning. Efficient incremental cooccurrence analysis for item-based collaborative filtering. *SSDBM*, 2019.

[27] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl. All roads lead to rome: optimistic recovery for distributed iterative data processing. *CIKM*, pages 1919–1928, 2013.

[28] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. Membership inference attacks against machine learning models. pages 3–18, 2017.

[29] J. Stoyanovich, S. Abiteboul, and G. Miklau. Data, responsibly: Fairness, neutrality and transparency in data analysis. *EDBT*, 2016.

[30] C. F. Van Loan and G. H. Golub. *Matrix computations.* Johns Hopkins University Press, 1983.