"Amnesia" – Towards 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) techniques have become ubiquitous over the last years. Recent law requires companies and institutions 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 can be considered a lossy compressed version of the data, and therefore it can be argued that the user data must also be removed from them. Typically, this requires an inefficient and costly retraining of the affected ML models from scratch, as well as access to the original training data.

We address this perfomance issue by formulating the problem of "decrementally" updating trained ML models to "forget" the data of a user, and present efficient decremental update procedures for three popular ML algorithms. In an experimental evaluation on synthetic data, we find that our decremental update is two orders of magnitude faster than retraining the model without a particular user's data in the majority of cases.

1. INTRODUCTION

Software systems that learn from user data with machine learning (ML) techniques have become ubiquitous over the last years [13]. 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 [6]. However, 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 can be considered a lossy compressed version of the data, and it can therefore be argued that the user data must also be removed from them. While relational databases offer transactional deletes and corresponding updates for materialized views over the data [8], 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.

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 2, and describe efficient decremental update procedures for three ML algorithms from different ML domains (recommendation, classification, regression) in Section 2.1. Our approach is applicable to a family of simple ML models with non-iterative training, which either exhibit sparse computational dependencies between the model and the data, or have an efficient update procedure for their closed form solution. We experimentally evaluate our approach on synthetically generated datasets in Section 4.

In summary, we provide the following contributions:

- We introduce the problem of making trained ML models "forget" user data via decremental updates (Section 2).
- We describe decremental update procedures for three algorithms from different ML domains (recommendation, classification, regression) which do not require access to the original training data (Section 2.1).
- We conduct an experimental evaluation on synthetic data, and find that our decremental update is two orders of magnitude faster than model retraining in the majority of cases (Section 4).

2. APPROACH

We first formalize the problem of efficient "decremental" updates for having models forget data. Afterwards we describe decremental update procedures for three popular ML algorithms in detail.

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

1st International Workshop on Applied AI for Database Systems and Applications (AIDB'19), August 26, 2019, Los Angeles, California, CA, USA.

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 [1]. In summary, the decremental update procedure t_{learn} must satisfy the following property:

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

(and provide a runtime for t_{forget} that is much smaller than the runtime of t_{learn} for retraining). 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. We therefore focus on non-iterative ML algorithms which exhibit sparse computational dependencies [5, 15] or have a closed form analytical solution. Our approach is based on the general idea of having the algorithm retain intermediate results during the model computation, which can be efficiently updated in a decremental manner later on.

A major operational advantage of our proposed approach is that t_{forget} does not require access to the original training data $\mathbf{D}$, which simplifies the integration of our approach into complex real-world ML deployments.

2.1 Decremental Update Procedures

In the following, we detail three algorithms for different ML tasks (recommendation, regression, classification) and describe efficient decremental update procedures for them.

Recommender Systems: Item-Based Collaborative Filtering. Item-based collaborative filtering [12, 14] is a 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 & Inference. 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 cooccurrence counts. Many similarity measures between items can be computed from the cooccurrence matrix [14]. 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, and generate items to recommmend for a particular user by computing preference estimates via a weighted sum between item similarities and the corresponding user history [12].

Removal of user data. In the following we assume that we have an existing recommendation model with its intermediate data structures: the item cooccurrence matrix $\mathbf{C}$, the item interaction count vector $\mathbf{n}$ and the item similarity matrix $\mathbf{S}$. We now wish to remove the data of the u-th user from the model, which corresponds to the u-th row $\mathbf{H}_u$ of the user-item interaction matrix $\mathbf{H}$. Algorithm 1 details how to update the intermediates to remove the user data 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 j_1 in the user history and recompute its corresponding row $\mathbf{S}_{j_1}$ in the similarity matrix.

Algorithm 1 Decremental update procedure for removing the effects of the user history $\mathbf{H}_u$ from an item-based recommendation model.

Input: item cooccurrence matrix $\mathbf{C}$
Input: item interaction counts $\mathbf{n}$
Input: item similarity matrix $\mathbf{S}$
Input: user history to remove $\mathbf{H}_u$
1: $\mathbf{n} \leftarrow \mathbf{n} - \mathbf{H}_u$
2: **for** item pair $(i_1, i_2) \in \mathbf{H}_u$ **do**
3: $\quad C_{i_1 i_2} \leftarrow C_{i_1 i_2} - 1$
4: **end for**
5: **for** item $j_1 \in \mathbf{H}_u$ **do**
6: $\quad$ **for** item $j_2 \in \mathbf{C}_{j_1}$ **do**
7: $\quad\quad S_{j_1 j_2} \leftarrow C_{j_1 j_2} / (n_{j_1} + n_{j_2} - C_{j_1 j_2})$
8: $\quad$ **end for**
9: **end for**

Regression: Ridge Regression. Ridge regression [9] is a widely used technique to predict a continous 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 & Inference. A common way to compute a ridge regression model in the form of the weight vector $\mathbf{w}$ is to solve the normal equation $\mathbf{w} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}$, e.g., via QR or LU factorization. The regularization constant λ is typically selected via cross-validation. The regression estimate $\hat{y}_{\text{new}}$ for a new observation $\mathbf{x}_{\text{new}}$ can be computed via its dot product with the weight vector: $\hat{y}_{\text{new}} = \mathbf{w}^\top \mathbf{x}_{\text{new}}$.

Removal of user data. 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 in Algorithm 2.

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 $\mathbf{QR} = \text{qr}(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})$ 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 with $-\mathbf{X}_u^\top \mathbf{X}_u$ as argument to update the QR decomposition $\mathbf{Q}$, $\mathbf{R}$ [2]. Afterwards, we can efficiently solve for the updated model $\mathbf{w}$ by back substitution.

Algorithm 2 Decremental update procedure for removing the effects of user data $\mathbf{X}_u$ from a ridge regression model.

Input: $\mathbf{z} \leftarrow \mathbf{X}^\top \mathbf{y}$
Input: $\mathbf{QR} \leftarrow \text{qr}(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})$
Input: user observation $\mathbf{X}_u$ to remove
1: $\mathbf{z} \leftarrow \mathbf{z} - \mathbf{X}_u y_u$
2: $\mathbf{QR} \leftarrow \text{qr\_update}(\mathbf{Q}, \mathbf{R}, -\mathbf{X}_u, \mathbf{X}_u)$
3: solve $\mathbf{Rw} = \mathbf{Q}^\top \mathbf{z}$ for $\mathbf{w}$ via back substitution

Classification: k-Nearest Neighbors with Locality Sensitive Hashing. Our last 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) [7]. 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 & Inference. 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 the predicted label $\hat{y}_{\text{new}}$ to $\mathbf{X}_{\text{new}}$.

Removal of user data. Removing an existing obervation $\mathbf{X}_\text{u}$ for a user u from our index works as depicted in 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.

Algorithm 3 Decremental update procedure for removing the user data $\mathbf{x}_u$ from an index for approximate k-NN.

Input: hash tables $T_1, \ldots, T_h$
Input: projection matrices $\mathbf{\Omega}_1, \ldots, \mathbf{\Omega}_h$
Input: observation $\mathbf{X}_u$ to remove
1: **for** $h = 1 \ldots H$ **do**
2: $b_{uh} = \text{sgn}(\mathbf{X}_u\mathbf{\Omega}_h)$
3: remove $\mathbf{X}_u$ from bucket b_{uh} in hash table T_h
4: **end for**

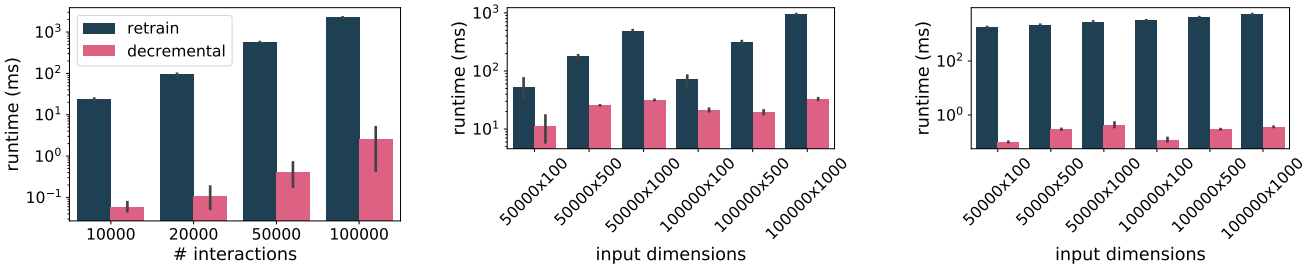
3. RELATED WORK

Ensuring that data processing technology adheres to legal and ethical standards in a fair and transparent manner [16] is an important research direction, which starts to gain a lot of attention from law makers and goverments.. The machine learning community has pioneered some work on removing data from models under the umbrella of "decremental learning"for support vector machines [1, 10]. In contrast to our work, these approaches need to re-access the training data for updating the model. An orthogonal technique to protect the privacy of user data in machine learning use cases is differential privacy [4]. In contrast to our presented approach, it is very difficult to design differentially private algorithms (even for experts [3]), and this approach requires a difficult decision on the limit on the acceptable privacy loss in practice. On the technical side, the approach used in this paper bears resemblance to general approaches for incremental data processing [5, 11, 15, 8] with the difference that it requires inverse operations for removing data, which are not covered in many existing approaches.

4. EVALUATION

We implement our algorithms together with their decremental update procedures using numpy and Python data structures. We put a more efficient C or Rust implementation aside for future work, as our main aim is to showcase the runtime difference originating from the different algorithmic approaches (retraining vs. decremental update). We generate various synthetic datasets for each presented algorithm from Section 2.1, and train a model on each such 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 six times and report the mean runtime as well as its standard deviation in Figure 1. Note that the runtime on the y-axis is displayed on a logarithmic scale.

Item-Based Collaborative Filtering. We generate synthetic interaction data by sampling from two independent two-parameter Poisson–Dirichlet distributions with $\alpha = 6000$ and $d = 0.3$, which are helpful for simulating powerlaw distributed data. We generate datasets with 10,000, 20,000, 50,000, and 100,000 interactions, and compare the time to retrain the model without a random user to the decremental update time for removing the interactions of the particular user (Figure 1(a)). In each experiment, the decremental update is at least two orders of magnitude faster than retraining the model as a whole, and the runtime difference increases with the size of the input data.



(a) Item-Based Collaborative Filtering. (b) Ridge Regression. (c) k-Nearest Neighbors.

Figure 1: Comparison of the 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 "decremental" updates the trained model to forget the user data. The runtime is displayed on a logarithmic scale.

Ridge regression. We generate synthetic data in different shapes (varying the number samples from 50,000 to 100,000 and the number of features from 100 to 500 to 1,000) using the `datasets.make_regression` function from the scikit-learn library to generate synthetic regression problems. We compare the time to retrain the model without a randomly picked user to the decremental update time for removing this particular user (Figure 1(b)). In each experiment, the decremental update is at least one order of magnitude faster than retraining, and we observe that the difference increases with growing data sizes. Additionally, we see that the runtime depends on both the number of samples and the number of features.

k-Nearest Neighbors. We again generate synthetic data in different shapes (varying the number samples from 50,000 to 100,000 and the number of features from 100 to 500 to 1,000). We leverage the `datasets.make_classification` function from scikit-learn to generate synthetic classification problems. We again compare the time to retrain the model without the data of a randomly chosen user to the decremental update time for removing this particular user's data (Figure 1(c)). We find that the runtime of the decremental update is two to three orders of magnitude less than the time required to retrain the model. Additionally, the runtime is independent of the number of samples, and only depends on number of features (which dictates the cost of the random projection).

Summary. In short, the results confirm our expectations for decremental updates: the updates are at least two orders of magnitude faster than retraining in cases where the algorithms exhibit sparse computational dependencies (item-based collaborative filtering and k-nearest neighbors), and one order of magnitude faster for algorithms where we need to update the whole model such as ridge regression.

5. CONCLUSION & FUTURE WORK

We described decremental update procedures for three trained ML models to make them efficiently "forget" the data of a user to remove. In future work, we aim to formalize the required algorithmic properties for our approach using the framework of differential computation and algebraic structures like monoids [11]. We will quantify the size of the required data structures as a function of the input data and describe the complexity of the update operations. Additionally, we will present decremental update procedures for more algorithms, evaluate them on real-world data and discuss how to integrate our approach with relational databases.

6. REFERENCES

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

[2] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart. Reorthogonalization and stable algorithms for updating the gram-schmidt qr factorization. *Mathematics of Computation*, 30(136):772–795, 1976.

[3] Z. Ding, Y. Wang, G. Wang, D. Zhang, and D. Kifer. Detecting violations of differential privacy. *CCS*, 2018.

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

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

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

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

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

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

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

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

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

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

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

[15] 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.

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