A Practical System for Privacy-Preserving Collaborative Filtering

Richard Chow[1]
Samsung Electronics, R&D
richard.chow@sisa.samsung.com

Manas A. Pathak[1]
Adchemy, Inc.
mpathak@adchemy.com

Cong Wang[1]
City University of Hong Kong
congwang@cityu.edu.hk

***Abstract*—Collaborative filtering is a widely-used technique in online services to enhance the accuracy of a recommender system. This technique, however, comes at the cost of users having to reveal their preferences, which has undesirable privacy implications. We propose a collaborative filtering system where the system does not observe the users' data and is still able to provide useful recommendations. Compared to prior systems, our emphasis is on building a practical system that can be reasonably used by a large number of users. Our approach involves creating a primitive to cluster similar users privately by modifying existing methods such as Locality Sensitive Hashing. Another technique we use is artificial ratings, as part of the process of privately predicting the rating for an item within a particular cluster. We evaluate our scheme on the Netflix Prize dataset, reporting the accuracy of our recommendations as a function of the privacy provided.**

I. Introduction

Recommender systems based on collaborative filtering [1] are widely used in online services to boost revenue and the user experience. However, the advantages of collaborative filtering are tempered by the potential cost to user privacy. As an example of the potential issues, in 2006 Netflix publicly released a dataset containing the movie ratings of more than 480,000 anonymized customers as part of the Netflix Prize contest [2]. The anonymization was broken by Narayanan and Shmatikov [3], prompting an FTC investigation and the lawsuit Doe v. Netflix [4] which alleges that the publication of the dataset potentially allows inference of sexual orientation. The lawsuit also notes that movie rental history may reveal personal issues such as domestic violence, adultery, alcoholism, or substance abuse. Because of such privacy concerns, Netflix cancelled the planned second stage of their contest [5].

In this paper we investigate whether one can obtain the advantages of collaborative filtering and yet protect end-user privacy. To this end, we argue that the most interesting architecture is that of a user connecting to an untrusted recommendation server. This is not only because a trusted server may compromise privacy by release of inadequately anonymized data as in the case of Netflix contest, but also because data may be released through compromised servers, unauthorized break-ins, and government subpoenas. User privacy is maximized when there is no need to trust the server with the data.

Previous work in this area has proposed algorithms that have been either impractical on large-scale data (*e.g.*, [6]), focused on weaker privacy models (*e.g.*, [7], [8]), or required additional network privacy infrastructure (*e.g.*, [9]). We propose an exceedingly practical and scalable privacy-preserving collaborative filtering framework which consists of a combination of user-side clustering and adding noise. Our framework requires no additional network infrastructure or trusted third-party.

As a running example, we describe our proposed algorithms using the Netflix dataset. We recognize that our algorithms may be of limited practical use with a company like Netflix: through its control of distribution and billing, Netflix knows the movies a user rents even without the ratings. As we describe later, the set of movies rated is the main privacy worry; the ratings themselves are secondary. Our system would be more useful for a pure recommendation site, say for news, products, or websites.

For our purposes though, the Netflix dataset offers a convenient way to demonstrate the prediction accuracy of our algorithms and compare with other (non-private) algorithms. The well-studied Netflix dataset has become the gold standard for recommendation datasets, with a well-defined measure of prediction accuracy: the root mean squared error (RMSE). We cannot expect comparable RMSE values in our case where the server does not have the actual data, but we are able to achieve RMSE values that are better than baseline algorithms in the non-private case. As far as we know, we are the first to test the recommendation accuracy of our algorithms on the Netflix dataset in the untrusted server case ([7] has a weaker privacy model of a trusted server but also tests against the Netflix dataset).

Our basic algorithm involves a clustering step and a recommendation step. The clustering step by itself may be interesting, as in some applications simply clustering users may be useful for, say, purposes of market segmentation.

Clustering Step: The server distributes a data-independent hashing algorithm to the set of users. The hashing function is fixed and public and operates on the space of movie ratings. The user computes on his private machine the hash value for his movie ratings and uploads the hash value to the server. The server partitions the set of users into buckets of similar users, where two users are in the same bucket if their hash value matches.

Recommendation Step: The user adds noise to his ratings vector and uploads this noisy ratings vector. The noise is in

[1]Work performed while at PARC.

the form of additional movies that are artificially rated. For a given user and given movie, the predicted rating is the average rating for that movie over the set of users similar to that user. The average is calculated from the noisy ratings vectors.

A. Privacy Properties

We postulate that in the case of an extremely sparse dataset, such as the Netflix dataset where 99% of possible ratings are missing, the main source of privacy concern is revealing the set of movies actually rated. The fact that a user has seen a given movie is more sensitive than the fact that the user has not seen a movie. The ratings themselves are also less important. This is because with these sparse datasets, the normal condition is for a user not to have rated an item and thus revealing this fact is generally not privacy-sensitive.

The privacy properties of our system are data-dependent, and we do not offer a rigorous description and proof of any privacy guarantees for our system. Instead, we describe an informal privacy goal and offer empirical evidence that our goal is satisfied with the Netflix dataset. Our informal privacy goal is that *for any given movie, a server without auxiliary data about the user cannot reasonably (i.e., probabilistically) infer that the user has rated the movie*. Note that this goal is less important for very popular movies, as a large proportion of users may have rated the movie. For simplicity, we focus our arguments in the paper to the case of any single movie, although in practice the arguments will hold for any small set of movies.

B. Contributions

The main contribution of this work is to design a practical solution to collaborative filtering where the users do not trust the server with their recommendation data. This suggests the use of homomorphic encryption and other state-of-the-art cryptographic protocols. We have, however, avoided the use of such methods in our basic algorithm, as current cryptographic techniques are still not practical in terms of computational cost for large datasets.

In particular, we have concentrated our experimentation on the Netflix dataset, which are the movie ratings of 480,189 users for 17,770 movies. We present accuracy results for our algorithm on this well-studied dataset. An algorithm for collaborative filtering which preserves user privacy is not very useful if it does not achieve at least some level of accuracy. The RMSE of our algorithm can be as low as 0.9883 and go up to 1.0380, depending on the number of fake ratings added, while in the non-private case, a simple baseline algorithm of averaging the ratings from all users has an RMSE of around 1.0540.

We analyze the privacy of our algorithms using existing notions such as ℓ-diversity [10]. One step in our algorithms is the use of a data-independent hashing function, and the use of such functions is well-known in the context of collaborative filtering. To the best of our knowledge, however, we are the first to analyze its privacy properties. We empirically show that our privacy notion is satisfied due to the sufficient diversity present in the Netflix dataset.

Finally, we present a novel data-independent hashing algorithm that is inspired by the Locality-Sensitive Hash (LSH) construction for cosine distance proposed by Charikar [11]. Our algorithm eliminates much of the random nature of LSH functions and promotes more uniform clusters. In practice, our algorithm gives us slightly better accuracy than LSH with at least equivalent privacy.

II. Background and Notation

A. Collaborative Filtering Algorithms

We review some of the concepts for collaborative filtering (CF); see [12] for a detailed survey. Formally, we consider a set of n users and a set of m items, such as movies, rated on a scale of, say, 1 to 5. Each user provides ratings for a subset of the items. Hence, the entire recommendation data consists of a user-by-item matrix of ratings. The aim of a collaborative filtering algorithm is to predict the ratings for the items which have not been rated. For notational simplicity, we denote the ratings by r and users by suffixes $u, v, \ldots$ and items by suffixes $i, j, \ldots$. We denote the rating of user u for an item i by $r_{u,i}$, the set of all ratings of user u by r_u and the set of all ratings for an item i by r_i.

We focus on CF algorithms that focus on computing user-wise similarities, as in our privacy setting we require each user to have access to only their own ratings. We review some of the standard steps of collaborative filtering using user-wise similarities below.

1) **Standardizing Ratings**. Since users may rate items higher or lower on average compared to other users, we require the users to standardize their ratings by subtracting the mean. A user u with ratings $r_u = \{r_{u,1}, \ldots, r_{u,m}\}$ with the mean rating $\bar{r}_u$, computes the standardized ratings $\{\hat{r}_{u,1}, \ldots, \hat{r}_{u,m}\}. = \{r_{u,1} - \bar{r}_u, \ldots, r_{u,m} - \bar{r}_u\}$.
2) **Grouping Similar Users**. We identify similar users by computing the distance over the standardized data for all user pairs. Commonly used distance metrics include cosine and Euclidean distance. For a given user, we can select the users with top-k distance scores, *i.e.* the k-nearest neighbors into one group. Another popular strategy is to identify approximate nearest neighbors by using *Locality Sensitive Hashing* which can be carried out in linear time and yet achieve comparable accuracy.
3) **Making Predictions**. Once the users are grouped together, we obtain the predicted rating for a particular user and particular item by computing the average standardized rating for that item given by the other users in the same group as the given user, and adding it to the average rating for that user. Hence, the predicted rating of an item i for a user u belonging to a group S is given by

$$r_{u,i} = \bar{r}_u + \frac{1}{|S_i|} \sum_{v \in S_i} \hat{r}_{v,i},$$

where S_i is the subset of users in S who have provided a rating for the item i. If using k-nearest neighbors, we can compute the average of the ratings in the group, weighted by the distance.

B. Locality Sensitive Hashing

We briefly review Locality Sensitive Hashing (LSH) below. While we do not directly use LSH in our collaborative filtering algorithm, our clustering algorithm was inspired by LSH and has similar properties.

A family of LSH [13] functions is defined for a particular distance metric such as cosine or Euclidean distances. A hash function from this family has the property that data points close to each other as defined by the distance metric fall into the same bucket with high probability compared to distant data points.

We consider the LSH family defined over cosine distances. Charikar [11] proposed the following construction for the LSH family defined over cosine distance. By picking each component of an n-dimensional random vector r_i independently from $\mathcal{N}(0,1)$, the hash function $L_i(q)$ for a vector q is equal to 1 if $r_i \cdot q \geq 0$ and 0 otherwise.

As using a single random hyperplane does not group the data points into fine-grained clusters, we use a set of k LSH functions and the final hash value is obtained by concatenating their output. This k-bit LSH function $L[x] = L_1(x) \cdots L_k(x)$ maps an n-dimensional vector into a k-bit string.

III. Operational Model

We assume that the users have access to only their individual ratings which are stored locally on their computing device. We assume that all users can communicate with an untrusted server over a secure channel, but do not need to communicate with each other. The latter condition is important to allow the system to scale to a large number of users.

We assume that the users and server are *semi-honest*, *i.e.*, they perform all the operations of the protocol faithfully, but try to extract as much information as possible from the intermediate results. The main assumption of this paper is that the users do not trust the server with their ratings: the unobscured ratings belonging to an individual user are never revealed to the server or other users.

While maintaining the semi-honest assumptions mentioned above, another source of vulnerability to the system is from the server colluding with some users directly or by creating fake users. As one of the steps in collaborative filtering is to group similar users together, the server can potentially know the ratings of multiple colluding users similar to a genuine user, and can use this to gain information about the ratings of the genuine user. In Section V, we analyze the loss of privacy, and find that it is minimal for the Netflix dataset.

IV. System Design

Our system follows the general scheme of collaborative filtering algorithms described in Section II-A where there are two main steps: clustering similar users and computing average ratings. In this section we present the design of our system, along with a discussion of how user privacy is preserved at each step.

A. Clustering via Hash Function Evaluation

As a first approximation to our hash function, we use a set of k hash functions chosen from the LSH function family for cosine similarity proposed by Charikar [11] (see Section II-B). In essence, we choose k random vectors, and split the entire space by the hyperplanes defined by these vectors. If $k = 8$, we split the space of ratings into 256 "cones", and users in the same cone have the same hash value and thus are considered similar.

First, we remark that we cannot directly use an LSH scheme. The random nature of the scheme can be a potential problem for privacy. If one of the random vectors has a large component in the direction of a particular movie, the hash value reveals the user's preference for that movie. The server can also use this to its advantage by choosing such vectors to represent the LSH function. We therefore modify the usual LSH hash function scheme by ensuring our k random vectors do not have this property.

Secondly, if we assume a uniform distribution of user ratings, it seems reasonable to use equal-sized cones, rather than those of disparate sizes. This can be achieved by choosing the k vectors to be orthogonal. In practice, as compared to LSH, this strategy of choosing orthogonal vectors makes a small but measurable difference in the variance of the cluster sizes (decreased by 2%) and also increased slightly our overall accuracy (see Section VI).

Our hash function is thus constructed as follows. We generate k random vectors and apply the Gram-Schmidt process [14] to ensure that each vector is orthogonal. After normalizing to unit length, we also check that there is no component of any vector larger than, say, 0.03. In the unlikely event that some component is larger than 0.03, we simply replace it with another vector. We experimented with several values of k for the Netflix dataset, and $k = 8$ gave us reasonable accuracy and at the same time provided good privacy (see Section V). Note that there could potentially be a set of k vectors that result in higher accuracy on the Netflix data than our set of k vectors. In practice, these could be found by extensive experimentation, but in an instantiated system, the server would not be able to perform these experiments while preserving user privacy. As we require a data independent scheme to identify similar users, we choose random orthogonal vectors while satisfying the constraints mentioned above.

We first describe the clustering algorithm. We describe the averaging step in Section IV-B. The mechanism used in our system only involves a simple two-step interaction between users and the server as illustrated in Figure 1. It incurs little additional communication and computation overhead compared to current collaborative filtering systems.

Initialization:

1) Server generates a set of k random orthogonal vectors as described above. This defines the hash function $L[\cdot]$ to

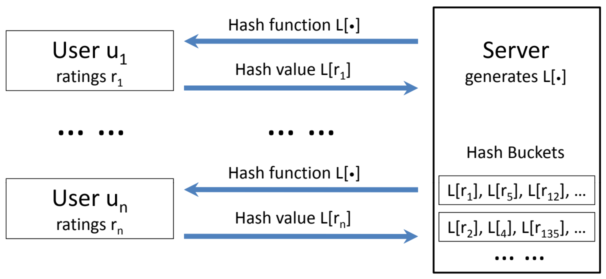


Fig. 1: Clustering algorithm. The server generates the hash function and pushes it down to all users. The users each compute a hash value, which are sent back to the server and used to cluster the users.

be used for the clustering. Each of the k vectors defines a bit in the hash value as in Section II-B. The hash function is considered public.

2) Server transfers the hash function $L[\cdot]$ to all users.
3) Users compute their standardized ratings $\hat{r}_u$ as described in Step 1 of Section II-A.

Applying hash function:

1) All users apply the hash function $L[\cdot]$ to their standardized ratings $\hat{r}_u$ to get the hash value $L[\hat{r}_u]$, a k-bit binary vector. All users upload their hash value to the server.
2) The server clusters users by hash value.

Remark. The server is assumed to be able to observe the hash values provided by all users (see Section III). We therefore do not need to protect the hash values in the open system and rely on the diversity within our clusters to support our privacy requirements (see Section V).

Performance. With the Netflix data set of 17,770 movies and k = 8, the hash function consists of 8 vectors each with 17770-dimensions, comprising about 1.08 MB in total. The hash value to be transmitted to the server is just one byte. The evaluation of $L[\cdot]$ over the individual user's ratings is a simple matrix-vector multiplication requiring only a few milliseconds.

B. Averaging Step via Obfuscated Ratings Vector

As discussed in Section II-A, the server needs to compute the average rating of all the users in the same hash bucket, without observing the ratings. This is a well-established example of a secure multiparty computation problem, and various protocols (*e.g.,* [15]), have been developed using cryptographic techniques such as homomorphic encryption. However, most of these techniques are computationally impractical or require a trusted party.

We instead propose having users send obfuscated ratings (not standardized) to the server, which can then compute the average rating over the obfuscated ratings. Perturbing ratings for preserving privacy is a standard approach (see, for instance, [16]), but there are concerns about the degree of privacy offered [17], [18]. Thus, compared to a cryptographic approach this method has the drawback of a potential degradation in both accuracy and privacy. However, the advantage is a much more efficient system. There are many potential schemes for obfuscating the ratings, and we present just one simple technique; better techniques in terms of privacy and accuracy are certainly possible. We analyze the loss in accuracy and privacy of our simple technique in Section V. In particular, we quantify the loss of accuracy in Figure 3 and show that reasonable accuracy is still attainable.

In our technique, the obfuscation is in the form of adding some fake ratings to randomly selected movies. The ratio of the number of fake ratings over true ratings is a parameter of the system, and is chosen based on a privacy-utility trade-off. A higher ratio increases the user's privacy via plausible deniability. Namely, for any given movie, the server can only conclude that the movie has been watched by the user with probability $N/(N+M)$, where N movies are actually rated by the user and M movies are artificially rated, and $N+M \ll$ total $\#$ of movies. This analysis does not take movie popularity into account, but see Section IV-C below. We also require $N+M$ to be at least 80, as we need to ensure that the set of rated movies returned to the server belongs to a space of size at least 2^{80}, which is impractical to iterate over. Note that for small N, the server does not know the value of N. Each subset of the rated movies may constitute the set of actual rated movies, so if $N+M$ is small, the server can compute the hash value of every subset to see if the hash value matches the user's actual hash value. In the extreme case, there may be only one match, in which case the server knows with certainty the set of actual rated movies. If $N+M$ is large, then even if the server finds some subsets that match the user's hash value, the server cannot iterate through all subsets and hence cannot analyze these subsets for common properties (such as common movies).

Experimentally, we tried adding fake ratings in two different ways on the Netflix dataset. The first way is to simply choose ratings uniformly over $\{1, ..., 5\}$. The second way roughly preserves the user's own average rating. The user's true average $\bar{r}_u$ is calculated, and the ratings are drawn from $\mathcal{N}(\bar{r}_u, 1)$, rounding to the nearest rating in $\{1, ..., 5\}$. As expected (see Figure 3), the second way leads to more accurate recommendations.

When we add the same number of fake ratings as real, we expect an approximate doubling of the number of ratings in a cluster to average over for any particular movie, with half the ratings being fake. The accuracy is still found to be reasonable as there is an averaging out effect for the fake movies.

Finally, we note that in operation a user may change some of his ratings over time. There is negligible performance cost to calculating a new hash value and noisy ratings vector. However, knowledge of both the old values and new values may be revealing. We make two suggestions here: (1) fake ratings should be maintained as much as possible (else a movie going from rated to not seen would indicate it was most likely a fake, since a user deleting an actual rating is rare), and (2) uploading should be in batch (e.g., after at least a few ratings changes), to minimize possibilities of analyzing what changes might move a ratings vector moving from one hash value to another.

C. Alternatives

We considered data perturbation as a primitive for data obfuscation, i.e., perturbing all user's ratings, including the 0 rating. This approach would offer differentially private [19] guarantees and has a minimal computational overhead. However, the sparsity of datasets such as the Netflix dataset argues against this approach, since the noise that must be added would quickly outweigh the signal from the actual ratings.

We also considered perturbing the actual ratings in addition to adding artificial ratings to further protect the user, but the gain in privacy would be small, as the user is more concerned with protecting the set of rated movies more than the individual ratings as we have reasoned in Section I-A.

An obfuscation method worth consideration is to randomly set a fraction of rated movies to the 0 rating, *i.e.*, consider them as not rated. This introduces plausible deniability for movies not rated and in some applications it may be important for the server to have some uncertainty as to whether a movie has in fact not been seen. In our obfuscation algorithm, the server knows with certainty that the user has not seen certain items.

We could also take into account movie popularity when adding fake ratings. This would give the user a higher degree of plausible deniability for popular movies (see Section V-B). However, the system would become more complex, as users would need to be able to gauge movie popularity.

Another possibility is to calculate our hash on the obfuscated ratings rather than the true ratings. In this approach, the architecture is simpler as the server could actually perform the hash calculation. However, accuracy suffers. For instance, we tried this approach with $M = N$ and the accuracy was 1.0452.

Finally, we consider an environment where some subset of users are less privacy-sensitive and willing to reveal their actual ratings. For some applications, this seems a reasonable assumption, as for instance seen by the number of movie reviews posted in IMDB and product reviews on Amazon. If these users are numerous enough, we do not need the obfuscated ratings vector from users who are more privacy-sensitive, as we can simply average the ratings from the less privacy-sensitive users.

V. Privacy Analysis

Our basic scheme involves a clustering step and a recommendation step. We consider the potential loss of privacy from each of these below. Again, we focus on intuitive and empirical evidence rather than a rigorous description and proof of privacy properties.

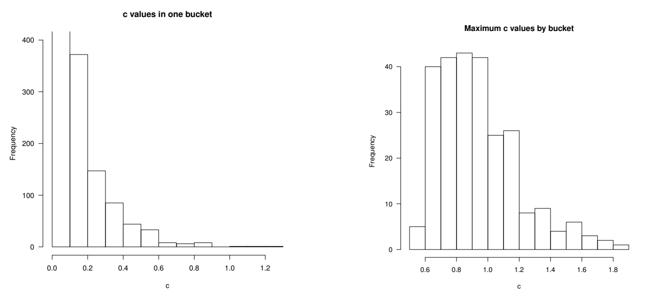
A. Privacy Loss from Clustering Step

We consider how much information is revealed from knowledge of a user's hash value. Clearly, this question is data-dependent. For instance, there is a loss of privacy if the individuals mapped to the same hash bucket have similar ratings, allowing inferences about one user's ratings by another user in that hash bucket.

We can state our privacy notion by adopting the language of ℓ-diversity [10]. We require that for any cluster and any particular movie, the ratio of the number of people who have seen the movie divided by the number who have not seen the movie is bounded by c:

$$\#\text{people who have seen} < c(\#\text{people who have not seen}).$$

In the language of ℓ-diversity, each hash bucket is $(c, 2)$-diverse for each movie attribute, with possible values "seen" or "unseen." (The "2" means the number of possible values of the attribute.) The seen value is considered sensitive. The unseen value is not considered sensitive, and hence we do not need to bound the reciprocal of the ratio. Hence, a user hashed to any particular bucket can deny having seen any particular movie where c measures the plausibility of this denial.



(a) Typical bucket's c values. The histogram is truncated for clarity; there are over 17000 values in the range 0.0 and 0.1.

(b) Histogram of maximum c values. There are 256 values, one for each bucket. Each value is the maximum c in the bucket.

Fig. 2: Diversity of c values

Figure 2a shows the distribution of 17770 c values for a typical bucket. The vast majority (over 17000) of c values are near 0, with a handful of movies having c values over 1. Note that the c value encapsulates external information about the movie, such as popularity.

Figure 2b shows the distribution of the 256 maximum c values in each bucket over all movies. In general, most buckets have a maximum c value of less than 1. The bucket with largest c value has a maximum of about 1.8. Hence, in the worst case (over all movies and all buckets), about $1/3$ of the users in the bucket have not seen the movie.

Finally, we remark that there is inherent tension between privacy vs. accuracy in a clustering-based collaborative filtering algorithm. Homogeneous buckets correspond to high accuracy, and heterogeneous buckets to high privacy, since revealing which users are similar is necessary to make the recommendation. We expect users with the same hash to be somewhat similar, else the recommendations would be no good at all. On the other hand, if users with the same hash are too similar, knowledge of the hash value can be privacy invasive. Our experiments show a high degree of heterogeneity for our buckets in the attributes important for privacy; in Section VI we also show reasonable accuracy.

B. Privacy Loss from Recommendation Step

We consider how much is revealed by our simple obfuscation of the ratings vector. For concreteness, we describe the case of N movies actually rated by the user and M movies artificially rated, where $N + M \ll$ number of total movies. If the server sees a 0, then the server can conclude that the user didn't see the movie. If the server sees an actual rating, then the server can guess that the user saw the movie with probability about $N/(N + M)$. Here we do not take into consideration movie popularity, where it may be more likely that one movie has been seen than another.

In general, one must consider a server having the obfuscated ratings vector for an entire hash bucket of users. First, we observe that for popular movies, our scheme does not prevent inference that the user has rated the movie. Consider a prototypical attack. The server observes obfuscated rating vectors from a cluster. Each movie will have a varying number of ratings, and since fake ratings are assigned to random movies, the number of fake ratings will be about the same for each movie. Popular movies will have a large number of ratings, relative to the number of fake ratings. Any rating of a popular movie will hence most likely be an actual rating.

To continue this analysis, this attack is not possible for movies where the number of actual ratings is comparable to the number of fake ratings within a cluster. In the case the number of fake ratings added is the same as the number of actual ratings, this will be true for movies with a small value of c, say $c \leq 0.1$. For a sample data point, in the histogram of Figure 2a, there are 706 movies out of the total 17770 with $c > 0.1$. For these relatively popular movies, there is a high probability that a user has seen the movie if it appears in his rated set. For most movies, however, the number of ratings received is comparable to the number of expected fake ratings, and so no inference can be made. In Section IV-C, we mention the possibility of adjusting the fake ratings for popularity, at a cost of a more complex system.

VI. Accuracy

The recommendation accuracy for the Netflix Prize is measured by the root mean squared error (RMSE). For instance, the Netflix prize winner had an RMSE of 0.8572 while the original Netflix algorithm Cinematch had an RMSE of 0.9525. The relatively simple baseline algorithm of averaging the rating from all users has an RMSE of around 1.0540 (see [20]). In our case we cannot expect comparable RMSE values since the server does not have the actual data. Our algorithms have RMSE values starting from 0.9883.

Table I shows the accuracy of the clustering step of our proposed algorithm compared with a baseline privacy-preserving algorithm and an algorithm where the usual LSH random vectors are used. Our hash function does slightly better than the LSH scheme, and both do much better than the baseline algorithm. The baseline privacy-preserving algorithm is simply to predict the user's mean rating; this does not need to interact with the server and is of course privacy-preserving.

To compute our accuracy figures, we assume here that the server can take the clusters output by the algorithms and, by some privacy-preserving process, generate predicted ratings by averaging the ratings of users in the cluster who have seen the movie. In practice, getting this average rating will reduce the accuracy. For example, we may get only a noisy average after adding fake ratings to the original ratings.

Algorithm	**RMSE**
User means	1.1325
LSH	0.9932
Proposed Hash	0.9883

TABLE I: Clustering Accuracy. The "user means" algorithm is the non-collaborative algorithm of always predicting the user's mean rating. The "LSH" algorithm means using the usual LSH algorithm. "Proposed Hash" is our proposed hashing scheme.

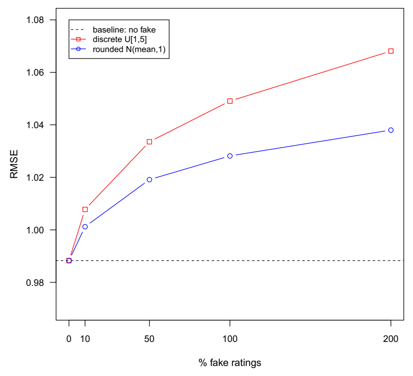


Fig. 3: Privacy vs. Accuracy Tradeoff. The plot shows the decrease in accuracy as relatively more fake ratings are added. We picked a rating from a uniform distribution for U[1,5] and from a Gaussian distribution for N(mean,1).

Figure 3 shows the results of experiments with adding fake ratings to the original user ratings. We see that even after adding as many as double the number of actual ratings, the prediction RMSE stays as low as 1.0380, below the baseline.

As described in Section IV-B, we compared two strategies to generate fake ratings, using a uniform distribution and a Gaussian distribution. We observe that the latter strategy performs much better than the former. We hypothesize that this is because the distortion of user characteristics is less. Along these lines, another interesting strategy that we did not try would be to generate fake ratings that preserve the user's hash value.

VII. Related Work

There is considerable previous work in the intersection of privacy-preserving data mining and collaborative filtering. Canny [6], [21] studies the same problem that we do and may be the first to pose the problem. He proposes a scheme where each user computes a public aggregate of their data using a public blackboard provided by the server. Using homomorphic encryption and a private computation, the user is able to get personalized recommendations. Our approach is much more practical than Canny's and involves considerably less work on the client side. In particular, our basic approach is easily implemented and involves no cryptography, only a matrix multiplication and adding noise.

Polat and Du [8], [22] study similar problems but propose different approaches. In [8], their algorithm adds noise to ratings on the client side and computes a weighted average of all users' ratings on the server side for the predicted rating. However, their privacy model is weaker as the user only adds noise to items the user has rated; this is problematical because the set of rated items can be as privacy-invasive as the actual ratings. They identify similar users through the noisy ratings, while we rely on our data-independent hash function. The algorithms proposed in [22] add fake ratings in the case of binary ratings, where they study the problem of recommending items rather than predicting ratings. Huang et al. [17] and Zhang et al. [18] have shown that basic noise addition schemes in many cases leak private information. Parra-Arnau combines noise with suppression of ratings in [16]. In general, the technique of adding fake ratings to enhance privacy is a known technique, sometimes called "randomized response," after Warner [23] proposed the method in the context of sensitive statistical surveys.

Nandi et al. propose privacy-preserving personalization middleware in [9]. Their idea is to locally compute the user's profile on the device and then anonymously aggregate the data from similar users at a middleware node. The anonymous aggregation depends on an anonymization network, such as the TOR onion router, and a Distributed Hash Table. The method to compute the user's profile locally is in spirit similar to ours, but we do not rely on any network privacy infrastructure to aggregate data.

McSherry and Mironov [7] study a weaker privacy model than ours. In their collaborative filtering model, a trusted server holds actual user data and perturbs the output to prevent the leak of inferences about user data from the recommendations made by the server. They also perform experiments on the Netflix Prize dataset. Since the server in their model can see actual data, the RMSE numbers are not comparable to ours.

There has been some previous work on collaborative filtering and a data-independent hashing technique, but without privacy analysis of the hashing technique. For instance, Das et al. [24] uses the LSH technique with the Jaccard similarity measure in order to recommend articles to Google News readers.

VIII. Conclusion

We have presented a practical scheme for privacy-preserving collaborative filtering. Our basic scheme is easily implemented and scalable and is a combination of user-side clustering and adding noise. The user is able to receive recommendations from an untrusted server without revealing actual data.

Our experiments with the Netflix Prize dataset reveal that the recommendation accuracy of our algorithms is considerably better than baseline algorithms, but, as expected, is worse than algorithms where actual, unobfuscated data is available. We empirically demonstrate the privacy property of ℓ-diversity for our algorithms on the Netflix dataset.

Acknowledgment

The authors would like to thank Elaine Shi, Oliver Brdiczka, and Bo Begole for many helpful discussions.

References

[1] Wikipedia, "Collaborative filtering," http://en.wikipedia.org/wiki/Collaborative_filtering.

[2] J. Bennett and S. Lanning, "The Netflix Prize," in *KDD Cup and Workshop*, 2007.

[3] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets." in *IEEE Symposium on Security and Privacy*, 2008, pp. 111–125.

[4] R. Singel, "Netflix Spilled Your Brokeback Mountain Secret, Lawsuit Claims," http://www.wired.com/threatlevel/2009/12/netflix-privacy-lawsuit/.

[5] N. Hunt, "Netflix prize update," http://blog.netflix.com/2010/03/this-is-neil-hunt-chief-product-officer.html.

[6] J. Canny, "Collaborative filtering with privacy," in *IEEE Symposium on Security and Privacy*, 2002.

[7] F. McSherry and I. Mironov, "Differentially private recommender systems: Building privacy into the Netflix prize contenders," in *KDD*, 2009, pp. 627–636.

[8] H. Polat and W. Du, "Privacy-preserving collaborative filtering using randomized perturbation techniques," in *ICDM*, 2003, pp. 625–628.

[9] A. Nandi, A. Aghasaryan, and M. Bouzid, "P3: A privacy preserving personalization middleware for recommendation-based services," in *Hot Topics in Privacy Enhancing Technologies Symposium*, 2011.

[10] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam, "ℓ-diversity: Privacy beyond k-anonymity," *ACM Transactions on Knowledge Discovery from Data*, vol. 1, no. 1, 2007.

[11] M. Charikar, "Similarity estimation techniques from rounding algorithms," in *34th Annual ACM Symposium on Theory of Computing*, 2002.

[12] X. Su and T. M. Khoshgoftaar, "A survey of collaborative filtering techniques," *Advances in Artificial Intellegence*, 2009.

[13] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Communications of the ACM*, vol. 51, pp. 117–122, 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327494

[14] G. Strang, *Introduction to linear algebra*, 4th ed. Wellesley-Cambridge Press, 2009.

[15] E. Shi, T-H. Hubert Chan, E. Rieffel, R. Chow, and D. Song, "Privacy-preserving aggregation of time-series data," in *NDSS*, 2011.

[16] J. Parra-Arnau, D. Rebollo-Monedero, and J. Forné, "A privacy-protecting architecture for collaborative filtering via forgery and suppression of ratings," in *DPM/SETOP*, 2011, pp. 42–57.

[17] Z. Huang, W. Du, and B. Chen, "Deriving private information from randomized data," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/1066157.1066163

[18] S. Zhang, J. Ford, and F. Makedon, "Deriving private information from randomly perturbed ratings," in *In Siam Conference on Data Mining*, 2006.

[19] C. Dwork, "Differential privacy," in *International Colloquium on Automata, Languages and Programming*, 2006.

[20] Wikipedia, "Netflix Prize," http://en.wikipedia.org/wiki/Netflix_Prize.

[21] J. Canny, "Collaborative filtering with privacy via factor analysis," in *SIGIR*, 2002, pp. 238–245.

[22] H. Polat and W. Du, "Achieving private recommendations using randomized response techniques." in *PAKDD'06*, 2006, pp. 637–646.

[23] S. L. Warner, "Randomized response: A survey technique for eliminating evasive answer bias," in *Journal of the American Statistical Association*, vol. 60(309), 1965, pp. 63–69.

[24] A. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *WWW*, 2007, pp. 271–280.

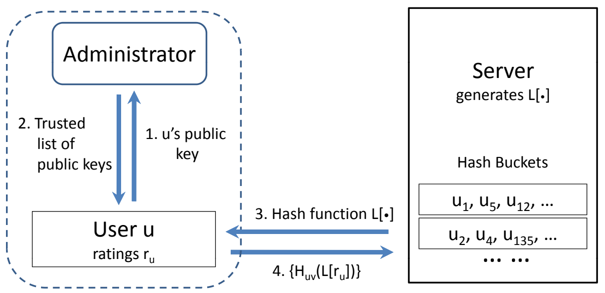


Fig. 4: Clustering algorithm in the closed system. Users send their public keys to an administrator, who is trusted to maintain a public list of users. The server generates the hash function and pushes it down to users. For each other user v in the list of public keys, user u applies a salted, one-way function $H_{uv}(\cdot)$ to the hash value and sends the result to the server. The one-way function is specific to u and v. u and v are in the same bucket if the values of $H_{uv}(L[r_u])$ and $H_{uv}(L[r_v])$ match.

Appendix

In the case collusion between a user and the server can be prevented, we present a stronger, "closed" variant of the system. The "closed" system has a restricted and known set of users. One example scenario is centrally managed users, such as a corporate network consisting of employees utilizing an external cloud-based collaborative filtering service. In this scenario, it is considered impractical for the external recommendation server to collude with individual users.

The advantage of the closed system is that we can prevent the server from learning the actual hash value. That is the reason for the use of the one-way function. The knowledge gained by the server is precisely the partitioning of the users into their hash buckets. Knowledge about a user can only come from auxiliary knowledge about other users and which buckets these users fall into, that is, which users are considered similar.

Below we present a design where using cryptographic techniques the server is restricted to observe only the set of users that are similar to a given user, not the actual ratings of any user. In the closed system, the server is assumed to be unable to collude with users. By applying a one-way function, we leverage this assumption to hide from the server the actual hash values provided by the user. While the server can create fake users with known hash values, it is unable to match these with any real users. Our protocol uses a one-way hash function based on the discrete logarithm problem and is similar to a Diffie-Hellman key exchange.

We assume user accounts are managed by a trusted administrator within the closed system. This trusted administrator cannot observe user ratings, but is able to observe user hash values as it has the ability to create accounts.

Initialization. Similar to the open system, the users standardize their data and the server generates the hash function $L[\cdot]$. In addition, the users and the administrator in the closed the system perform the following steps.

1) The administrator inside the closed system picks a safe prime p, where $p = 2q + 1$ and q also prime, and then generates a subgroup $\mathbb{Z}_p^*$ of prime order q and generator g. The administrator makes g and p public to all the users.
2) When user u joins the system, he generates a random number s_u from $\mathbb{Z}_q$, and sends a public key $g^{s_u} \bmod p$ to the administrator.
3) The administrator releases the list of all public keys $\{g^{s_1} \bmod p, \ldots, g^{s_n} \bmod p\}$ to the users. There is no harm done if the adversary is able to see this list, but the adversary must not be able to add his own public key to the list.

Applying hash function Similar to the open system, the server transfers the hash function $L[\cdot]$ to all users, who then apply it to their standardized ratings to obtain the hash value $L[\hat{r}_u]$. We describe the protocol for applying the one-way function to the hash value below.

1) For each public key g^{s_j}, $j \neq u$, in the list $\{g^{s_1} \bmod p, \ldots, g^{s_n} \bmod p\}$, each user u computes the one-way hash value
$$g^{s_u s_j + L[\hat{r}_u]} \bmod p = (g^{s_j})^{s_u} g^{L[\hat{r}_u]} \bmod p.$$
Here s_u is the user's previously chosen random number and $L[\hat{r}_u]$ is the clustering hash value. The expression
$$g^{s_u s_v + L[\hat{r}_u]} \bmod p$$
is the $H_{uv}(L[r_u])$ of Figure 4.
2) Each user u sends the set of pair-wise one-way hash values $\{g^{s_u s_j + L[\hat{r}_u]}\}$, for $i \neq u$, to the server.
3) The server clusters users that share the same one-way hash value in a pair-wise manner. For example, for the three users u, v, w, if $g^{s_u s_v + L[\hat{r}_u]} = g^{s_v s_u + L[\hat{r}_v]}$, and $g^{s_u s_w + L[\hat{r}_u]} = g^{s_w s_u + L[\hat{r}_w]}$, then all these three users are clustered into the same bucket.

Remark. We assume computational hardness for the discrete log problem, and hence the server and, similarly, the users cannot reverse $g^{s_u} \bmod p$ to obtain s_u for other users. Thus, $g^{s_u s_i}$ is the pairwise secret established between user u and any other user i. As a result, the one-way hash value of $\{g^{s_u s_i + L[\hat{r}_u]}\}$, where $i \neq u$, reveals nothing about user u's clustering hash value $L[\hat{r}_u]$. That is, the server knows nothing beyond the clustering of users.

Performance. We analyze performance on the Netflix dataset. Since the one-way hash function value is of size 1024-bits or 128 bytes and there are 480,189 users, the communication load for each user in the system is 58.6 MB, which would require a few minutes to transmit on a broadband network. Similarly, each user needs to evaluate the one-way hash function 480,188 times, which required 11 min 36 s with our implementation.[2] It is important to note that both of these costs are only for the one-time initialization. Once the clustering is done, the server can provide recommendations without requiring further interaction with the users.

[2]We implemented the protocol in C++ on a desktop machine running 64-bit Ubuntu 11.04 with 3 GHz Intel Core 2 Duo processor and 3.7 GB RAM.