Collaborative Filtering with Privacy

John Canny
Computer Science Division
UC Berkeley, CA 94720
jfc@cs.berkeley.edu

Abstract

Server-based collaborative filtering systems have been very successful in e-commerce and in direct recommendation applications. In future, they have many potential applications in ubiquitous computing settings. But today's schemes have problems such as loss of privacy, favoring retail monopolies, and with hampering diffusion of innovations. We propose an alternative model in which users control all of their log data. We describe an algorithm whereby a community of users can compute a public "aggregate" of their data that does not expose individual users' data. The aggregate allows personalized recommendations to be computed by members of the community, or by outsiders. The numerical algorithm is fast, robust and accurate. Our method reduces the collaborative filtering task to an iterative calculation of the aggregate requiring only addition of vectors of user data. Then we use homomorphic encryption to allow sums of encrypted vectors to be computed and decrypted without exposing individual data. We give verification schemes for all parties in the computation. Our system can be implemented with untrusted servers, or with additional infrastructure, as a fully peer-to-peer (P2P) system.

1 Introduction

Collaborative filtering has important applications in e-commerce, direct recommendations (such as Movielens and Ringo) and search engines. Personalized purchase recommendations on a web site are can significantly increase the likelihood over a customer making a purchase, compared to unpersonalized suggestions. In future ubiquitous computing settings, users will routinely be able to record their own locations (via GPS on personal computing devices and phones), and their purchases (through digital wallets or through their credit card records). Through collaborative filtering, users could get recommendations about many of their everyday activities, including restaurants, bars, movies, and interesting sights to see and things to do in a neighborhood or city. But such applications are infeasible without strong protection of individual data privacy.

Today's server-based collaborative filtering systems have a number of disadvantages. First of all, they are a serious threat to individual privacy. Most online vendors collect buying information about their customers, and make reasonable efforts to keep this data private. However, customer data is a valuable asset and it is routinely sold as such when companies have suffered bankruptcy. At this time this practice is supported by case law. A second disadvantage is that server-based systems encourage monopolies. There are correlations between customer purchase choices across product domains. So companies that can acquire preference data for many users in one product domain have a considerable advantage when entering another. Even within one market, a large established firm will have an advantage over any new competitor, because the latter will have a much smaller corpus of customer data to draw from, leading to less accurate (and less sucessful) recommendations. From the customer perspective, their purchase history is fragmented across many vendors reducing the quality of their recommendations.

Finally there is a subtle but important sociological disadvantage. Today's collaborative filtering algorithms are all based on ratings from the most similar users to a given user. In the language of diffusion of innovation [13], this is called homophilous diffusion. Homophilous diffusion allows rapid diffusion of innovations *within socio-economic groups*. But diffusion throughout society requires *heterophilous* diffusion, where individuals seek recommendations from more advanced peers who are *unlike* them. The lack of choice in today's systems defeats heterophilous diffusion. For instance, although I am not an expert in cooking or gardening or medicine, there are times when I would like recommendations from those communities, not from my own peers, or the population as a whole. We propose a system where communities create their own knowledge pools, and where they decide whether to share this information with outsiders. In some cases, access to this infor-

mation could be a service that the community provides to outsiders for a fee. Communities that do this will allow heterophilous diffusion.

We propose a "User-owned and operated" principle for data such as purchase or document access, or position logs: Users should have exclusive control and access to all data recorded about them. They should be able to control how and with whom the data will be shared. And they should be able to hide or restrict any part of the data. This provides us with an interesting algorithm design challenge: is there a practical algorithm for collaborative filtering using many users' data, which does not expose any individual's data?

In this paper we propose a solution, and argue that it is practical for an interesting class of application data, given recent developments in distributed computing infrastructure. Our scheme is based on distributed computation of a certain "aggregate" of all users' data. The aggregate is treated as public data. Each user constructs the aggregate and uses local computation to get personalized recommendations. The computation is designed to be done either on a single reliable server or in peer-to-peer fashion on unreliable, untrusted clients. The peer-to-peer architecture allows users to create and maintain their own recommender groups themselves.

This approach addresses several of the difficulties with traditional collaborative filtering systems. While it doesnt prevent vendors from gathering customer data, it provides the customers with the same recommendation services that vendors normally provide. Customers can therefore use anonymized purchase systems which are being developed elsewhere, and still obtain personalized recommendations. It blunts the monopoly trend because users can obtain recommendations directly from their peers, and those recommendations could then route through meta-storefronts (such as C-net, Yahoo etc.) that point to multiple vendors. And it addresses the homophily issue via a community-based model of recommendation. The system we propose makes it easy for users to set up communities, and to share their data within several communities. Because a community aggregate hides individual member data, we believe that many communities will be willing to share their aggregate information with outsiders (in some cases perhaps charging for this service). By listing a community with a portal like Yahoo, a community could encourage outsiders to use their data[1].

Some of the benefits of our scheme could be obtained by pseudo-identities. Users could use persistent online identities that do not link to their actual identity, and make anonymous purchases. A persistent pseudo-identity allows inference of purchase patterns, and supports collaborative filtering. However, unless the user data is further protected, pseudo-identities expose users to significant privacy risks. For example, an attacker who can observe a few of a user's transactions may then be able to uniquely identify that user from the available records. The attacker can then discover the rest of that user's purchase records.

We believe that this paper describes a practical application of multi-party computation, subject to the availability of certain distributed or peer-to-peer services (a blackboard and a source of trusted random bits). Both commercial and research versions of such services exist today. Beyond collaborative filtering, there are many other potential applications of these techniques in areas such as online surveys, usability studies, or censuses; where aggregate data is the goal.

1.1 Assumptions

Let n be the number of users in a community, and m be the number of items rated by them. In order to be practical, our method must have an overall complexity which is pseudo-linear in n and m separately (there is an obvious $\Omega(nm)$ lower bound). Typical values for the number of users and items can range from thousands to millions. We show later that total communication and work for our protocol is $O(mn \log n)$. In section 4.2 we expand the constants in the method to show that is practical if both n and m are at the low to middle part of their ranges.

Users' computers perform all the computation in the method. It would be simpler to do this using a set of servers as in [5], but this would not achieve our goal of a community-based system. Our peer-to-peer version includes additional verification steps that [5] does not (their protocol is "publicly verifiable" but verification is not included in it).

We assume that a fraction $\alpha > 1/2$ of the users' machines are uncorrupted. Our protocol proceeds in rounds, and we model corruption as a *static adversary* on each round [1, 12]. That is, the adversary can choose which machines to corrupt at the start of a round, but this choice stays fixed throughout the round. We believe this model is realistic. First of all, in our setting the greatest risk of corruption is from malicious users who may want to extract information or influence the aggregate through their own machine. The identities of these users will be static. For an external adversary to corrupt the protocol, signatures and other information would need to be generated which would require essentially full control of a large set of users' machines. Such corruption is realistically a static process. An adversary having broken into a machine is unlikely to relinquish control of it during the round, nor is the user likely to be able to detect the fault and repair it during the round.

Even if we could justify an adaptive adversary, it would not be practical to defend against it. The best multi-

[1] If the community is popular, the portal would most likely need to cache the aggregate.

party protocols for adaptive adversaries have complexity of $O(n^3)$ or higher [4]. This is well outside the realm of possibility for a large-scale peer-to-peer community. Fortunately, the static adversary does model the risks to our protocol sensibly. This adversary model allows us to use a simple and efficient means to verify intermediate steps in our protocol by sampling.

Our method assumes two services: a write-once, read-many or WORM storage system (a blackboard), and a trusted source of random bits. The WORM service is no longer a theoretical abstraction, but is becoming part of the core services for distributed computing. Small scale commercial implementations exist in the Groove system (`www.groovenetworks.com`). It is not clear whether Groove scales to thousands or even hundreds of sites. But other systems under development do. The Oceanstore system [8] is designed to provide global-scale services, potentially to millions of sites. Similar services are under development as part of the JXTA peer-to-peer API within Java.

We also assume the existence of a trusted source of random coin tosses. Our protocol requires $O(km \log^2 n)$ random bits per round. Given the availability of a WORM provider, generating and distributing these random bits is straightforward.

1.2 Outline and Contributions

The paper is structured as follows: Section 2 formalizes the collaborative filtering problem, explains how we compute an aggregate model of user preferences, and how we obtain recommendations from it. The model is a partial SVD (Singular Value Decomposition) of the matrix of user ratings. We show in section 2 that the SVD algorithm, which is iterative, requires only vector addition of certain user data in each iteration. This structure is necessary for the algorithm to work on encrypted data. In section 3 we show how to compute the sum of encrypted data vectors, in a peer-to-peer fashion. Since peers are untrusted, this section also describes checks for both the original data (via ZKPs) and the sums computed from it (via a sample majority). Then in section 4 we give the complete protocol, along with proofs of its cryptographic protections. Its statistical protections are discussed later in section 5. In section 4.1 we describe experiments with an implementation of the numerical part of the algorithm. The cryptographic protocols have not yet been implemented, but we give a detailed analysis of the running time and space requirements of the algorithm on a typical dataset using a standard cryptographic software toolkit (CRYPTO++). In section 5 we discuss the statistical vulnerability of the system. Finally, section 6 gives a discussion and conclusions.

The main contribution of the paper is to make a connection between cryptographic techniques and an application domain (collaborative filtering) which we believe makes economic and practical sense. It is interesting also as plausible application of encrypted multi-party computation. Collaborative filtering using SVD is not new, and was described in [14]. But that paper used simple inner products to generate recommendations, while we use the maximum likelihood formulation of section 2.1, which is novel. This improves the mean-square error of our method over [14].

We cannot use a "black-box" SVD algorithm with the limitations of encrypted computation, so we derive an iterative SVD using the conjugate gradient method of Polak-Ribiere [11]. This gives us an iteration with only vector additions of user data. The derivation is a standard application of numerical techniques, so we include it as Appendix I. For the cryptographic portion of the method, with use ideas from the voting algorithm of Cramer, Gennaro, and Schoenmakers [5]. For initial key generation, we assume either an honest dealer, or the distributed key generation scheme of [9]. While we can use Pedersen directly, Cramer et al.'s [5] is a server-based protocol. For our peer-to-peer application, we needed some modifications and extensions to [5]. The main extension is the data and tally verification section 3.5, which uses sampling and a trusted source of random bits to allow clients to compute reliably with mostly-trustworthy peers. Finally we modified the multiplication protocol from [3] for ZKP of products to work for squares (see Appendix II). The resulting non-interactive proofs are 7 integers rather than 10 integers long. This reduces the overall computation and communication cost by a small but significant constant factor.

2 Distributed Collaborative Filtering

Assume there are n users and m items that have been rated by them. Let P be the matrix of user preference data, where P_{ij} is the rating given by user i to item j, and $i \in \{1, \ldots, n\}$ $j \in \{1, \ldots, m\}$. We set $P_{ij} = 0$ if user i has not rated item j, and require that actual ratings are non-zero. P is a typically a sparse matrix with many missing ratings (density is 0.03 for the EachMovie dataset). We will speak of "clients" and "talliers" although the two functions may occur on the same machines. Each user is assumed to own a client, so there are n clients. A tallier computes a total of client data (transformed from client ratings). For simplicity, we will assume there are n talliers which is the least structured (fully peer-to-peer) case. We assume that a fraction $\alpha > 0.5$ of clients are talliers are trustworthy, in the sense of correctly following the protocols. However, no-one is considered trustworthy enough to see unencrypted user data.

Collaborative filtering methods generally use weighted combinations of nearest neighbor votes to extrapolate from a user's preferences. Call these methods "neighborhood

methods". Neighborhood methods ignore global relationships between user preferences. In fact global *linear* relationships between user ratings do exist and were used in the "eigentaste" algorithm by Goldberg et. al. [6]. The eigentaste method is still a neighborhood method, but it uses projections of actual user ratings into a low-dimensional space. This space is computed with a singular-value decomposition of the ratings matrix. Goldberg showed that this projection before neighbor matching improves performance, and describes the linear basis vectors as "eigentastes".

This suggests that rating prediction might be done using *only* a global linear approximation to the ratings set. In practice we have found that this works quite well. On tests with the "Eachmovie" database, the ratings from the linear model are as good as the best current algorithms. In a later section we compare it with neighborhood methods using surveys from Herlocher [7] and Breese et. al. [2].

We construct the k-dimensional linear space A that best approximates the user preference matrix P in a least-squares sense. Assume A is represented as a row-orthonormal matrix $A \in \mathbb{R}^{k\times m}$. Now $k \leq m$ where m the number of data items, and the orthonormality condition implies that $AA^T = I$. The projection of P onto A is PA^TA. The residual modeling error is $E = P - PA^TA$ and we want to minimize the sum of squares of the components of this error matrix, which is $e = \mathrm{tr}(EE^T)$. This simplifies to $e = \mathrm{tr}(PP^T) - \mathrm{tr}(PA^TAP^T)$ and the minimum error is obtained when $\mathrm{tr}(PA^TAP^T)$ is maximized. The optimization problem is then to find A such that

$$A = \sup_{U:\ UU^T=I} \mathrm{tr}(PU^TUP^T)$$

This optimization uses a conjugate gradient scheme which is discussed in detail in Appendix I. In fact we show that as well as A, we can obtain a partial singular value decomposition (SVD) of P using encrypted computation. Our algorithm is a straightforward application of the conjugate gradient method, although there is a non-trivial change of basis at each step. There are more efficient ways to compute an SVD, but our goal is to compute it in a reasonable amount of time using a cryptographic homomorphism. The conjugate gradient scheme allows us to reduce the calculation to series of vector additions of user data. In practice its convergence is fast, taking 40-60 iterations on typical data.

2.1 Generating Recommendations

Each user seeking a recommendation will already have constructed the public matrix A in the course of running the protocol described in section 4. The user can then generate recommendations for themselves using A. User i has a $1 \times m$ matrix of preferences P_i. Many of these will be missing (represented by zeros in P) for items the user has never rated. Underlying our linear approximation is a probabilistic latent variable model. We assume that each user i has a static preference (row) vector $x_i \in \mathbb{R}^k$. Let D and V be the matrices derived from the SVD of P as described in appendix I. Then each user's ratings vector is given by

$$P_i = c_1 x_i (DV^T) + n_i$$

where $n_i \in \mathbb{R}^m$ is a "noise" random vector and c_1 is a constant. Both x_i and n_i are assumed to have gaussian distributions. The probability density of a given pair (x_i, n_i) is given by

$$c_2 \exp(-|x_i|^2/(2\sigma_x^2))\exp(-|n_i|^2/(2\sigma_x^2))$$

Given a vector of user preferences P_i, the most likely pair (x_i, n_i) is the pair that minimizes

$$|x_i|^2/(2\sigma_x^2) + |n_i|^2/(2\sigma_n^2)$$

where $n_i = P_i - c_1(x_i^T(DV^T))$ which is a quadratic minimization over x_i. The solution is easily shown to be

$$x_i = P_i B^T (I + BB^T)^{-1}$$

where B is the restriction of $c_1 DV^T$ to the columns containing known preferences for user i. The constant c_1 is given by $\sigma_n \sqrt{k|D|^2}$, and σ_n can be estimated from the dataset. Given x_i, the estimate of the user's preferences for other data items is given by $c_1 x_i (DV^T)$.

2.2 Updating the Aggregate

The numerical method for updating the aggregate is derived in appendix I. It is an iterative conjugate-gradient algorithm, using the Polak-Ribiere recurrence [11]. There are two phases to each iteration. First each user first computes their contribution to the gradient of A, which is

$$G_i = AP_i^T P_i (I - A^T A) \tag{1}$$

where P_i is the vector of preferences for the i^{th} user, and A is the aggregate from the previous iteration. Then all users add their gradient contributions using the protocol discussed in the next section. This gives a total gradient $G = \sum G_i$, shared by all the users. The next phase of conjugate gradient is "line minimization," where each user computes the scalar quantities:

$$\begin{aligned} c_i &= -2\mathrm{tr}(P_i G^T A P_i^T) \\ a_i &= -\mathrm{tr}(P_i G^T G P_i^T) \\ b_i &= \mathrm{tr}(P_i A^T G G^T A P_i^T) \end{aligned} \tag{2}$$

these values are also tallyed using the vector addition protocol in the next section to produce global values $(c, a, b) = \sum(c_i, a_i, b_i)$. Finally, from (c, a, b) the new aggregate is computed as described in appendix I. There are a few extra steps implement the Polak-Ribiere method but they do not require communication. They are covered in the appendix.

3 Vector Summation of Encrypted Data

We assume that each of n users has a vector of data $G_i \in \mathbb{R}^{k\times m}$ for $i = 1, \ldots, n$ representing their contribution to the gradient of A. For convenience, we treat each G_i as a vector with km coordinates $G_i = (G_{i1}, \ldots, G_{i(km)})$. We assume that every user data item G_{ij} is integer, and restricted to a small number of bits, say 10 bits. We assume that a fraction $\alpha > 0.5$ of clients and talliers are honest. The goal is to compute the vector sum $G = \sum_{i=1}^{n} G_i$ at all the honest talliers [2]. The privacy goals are that:

1. The tallier should gain no information about an individual user's data G_i, except that:
2. User data is almost surely valid. Almost surely valid means that $|G_i| < L$ with high probability. This is in spite of malicious behavior by some talliers or clients.
3. The total G should be verified. We will rely on multiple tallier computations and the trusted coin source to do this.

For our method to be practical we must meet some efficiency goals:

1. Typical collaborative filtering domains have hundreds to millions of items (this is the range of m). The dimension k is typically less than ten. Secondly, the number of users could range from 10 to 10^7. Clearly $\Omega(knm)$ is a lower bound on the total work that must be done. If each user contributes a processor, then the lower bound per machine is $\Omega(km)$. To be practical, the work per machine should stay within a polylog factor of $O(km)$.
2. The validity proof for each user's data should be "small" compared to the representation of G_i, and the time to check it should be small compared to the time to add G_i to the sum.
3. It should be possible to efficiently check the computation done by the talliers. This will turn out to be the most expensive step, and it requires a trusted global source of random bits.

Our scheme follows the general architecture of the election scheme of Cramer, Gennaro, and Schoenmakers [5]. There are several differences between our scheme and theirs. First, we are computing a sum of vectors of user preferences instead of binary user votes. Because of this, the ZKP of vote validity is different. Second, instead of voters+authorities, we have clients and one or more talliers. In our scheme, a private key is secret-shared among all clients, not just the authorities. Talliers perform a tallying function like the authorities, but are not assumed to be secure. So in fact our scheme could be implemented as a pure peer-to-peer system where clients and talliers are the same.

As mentioned earlier, we assume the parties share a blackboard to which they can all write and read, but such that one party's data cannot be erased or changed by anyone (a WORM store). We also assume the existence of a trusted source of random coin tosses.

3.1 Key Sharing

The goal of this step is to create a globally-known El-Gamal public key, and a matching private key which is held by no-one and instead secret-shared among all the clients. The key generation protocol of Pedersen [9] does this. The result is that each player has a share s_i of the decryption key s, and s can be linearly reconstructed from a sufficient number of shares. More precisely, let p and q be large primes such that $q|p-1$, and let G_q denote the subgroup of $\mathbb{Z}_p^*$ of order q. In normal El-Gamal encryption, a recipient chooses a $g \in G_q$ and a random secret key s, and publishes $g, h = g^s$ as their public key. In our case, we want the secret key to be held by no-one and instead secret-shared among all the players. After applying Pedersen's protocol, each client has a share s_i of the decryption key s, and s can be linearly reconstructed from any set Λ of $t+1$ shares via appropriate Lagrange coefficients:

$$s = \sum_{i\in\Lambda} s_i L_{i,\Lambda} \qquad L_{i,\Lambda} = \prod_{j\in\Lambda, j\neq i} \frac{j}{j-i}$$

These shares can also be used for threshold decryption of messages encrypted with the public key (g, h). We assume that p, q, g, h are known to all participants after Pedersen's protocol, as well as another generator $\gamma \in G_q$ needed for homomorphisms. We also assume that each user publishes a public key corresponding to s_i, which is needed to verify their decryption of data.

We choose the encryption threshold to be greater than the number of untrusted users, which is $(1-\alpha)n$. Taking $\alpha = 0.8$ for instance, gives us a threshold of $t = 0.2n$ which allows the scheme to work correctly even when a significant fraction of trusted clients are offline.

3.2 Value Encryption/Homomorphism

Each user has km data values $G_{ij}, j = 1, \ldots, km$. To encrypt, user i chooses km random values $r_{ij}, j = 1, \ldots, km$ from $\mathbb{Z}_q$. The encryption of the data is then

$$\Gamma_{ij} = (x_{ij}, y_{ij}) = (g^{r_{ij}}, \gamma^{G_{ij}} h^{r_{ij}}) (\mathrm{mod}\ p)$$

[2] There is also the second phase of totalling the 3-element vectors (c_i, a_i, b_i) but clearly the first phase dominates

for $j = 1, \ldots, km$. In other words, each value is a standard El-Gamal encryption of the exponentiation of a vote: $\gamma^{G_{ij}}$. User i sends these km values to the write-once blackboard. Notice that this map is a homomorphism. Define

$$\begin{aligned} H : \mathbb{Z}_q \times \mathbb{Z}_q &\rightarrow \mathbb{Z}_p \times \mathbb{Z}_p \\ h(v, r) &= (g^r, \gamma^v h^r) (\text{mod } p) \\ h(v_1 + v_2, r_1 + r_2) &= h(v_1, r_1) * h(v_2, r_2) (\text{mod } p) \end{aligned}$$

where the multiplication on the right side is element-wise. We will assume element-wise multiplication from here on. This homomorphism allows us to compute the encryption of a sum of votes by simply multiplying the encryptions. That is:

$$h(\sum_{i=1}^{n} G_{ij}, \sum_{i=1}^{n} r_{ij}) = \prod_{i=1}^{n} h(G_{ij}, r_{ij}) (\text{mod } p)$$

for $j = 1, \ldots, km$. This tally and all partial totals are El-Gamal encryptions, and provide computational hiding of the data.

3.3 ZK Proof of User Data Validity

Each user should give a ZKP that their encryptions $(\Gamma_{i1}, \ldots, \Gamma_{i(km)})$ represent a valid input, namely one that is not "too large". An expensive way to do this is to give a ZKP for every element G_{ij} that bounds its size. This is neither efficient nor desirable. The amount of influence a single user has over the aggregate can be bounded by the 2-norm of G_i. The squared 2-norm is just the sum of the squares of the elements of G_i. We can bound the 2-norm in zero-knowledge by bounding a single quantity which we prove is the sum of the squares of the elements of G_i. The bound uses ideas from [3], suitably adapted as described in Appendix II.

For each Γ_{ij} which encrypts a G_{ij}, we first generate a W_{ij} which encrypts G_{ij}^2. The prover and verifier multiply together W_{ij} for $j = 1, \ldots, km$ which will be an encryption of the sum of the squares of elments of G_i, i.e. the squared 2-norm of G_i. Then prover gives a ZKP that this value, call it ν is at most L^2, where L is the desired bound on the 2-norm of G.

The bound on ν is built by expressing ν as a weighted sum of $2 \log_2 L$ binary-valued variables (bits), and then showing in zero knowledge that each bit has value 0 or 1. Cramer and Damgård [3] give zero-knowledge, honest verifier proof that a given encrypted value is 0 or 1. Their method can be implemented non-interactively by hashing the verifier's response, which is best for our application. In that case each ZKP comprises 7 long (mod p) integers. So to prove that ν has at most t bits requires $7t$ large (typically 160- or 1024-bit) integers. The length required for our protocol is $t = O(\log mk)$. Note that there is only one such proof for each gradient vector G_i.

It remains for the prover to show in zero knowledge that each W_{ij} is the encryption of the square of the value encrypted by Γ_{ij}, for $j = 1, \ldots, km$. This requires a modification of the multiplication protocol from [3] which we give in Appendix II. Since we only need to deal with squares, the proof in Appendix II is shorter than the general multiplication proof. When implemented non-interactively, each proof requires a fixed number of integers from $\mathbb{Z}_p$. While the original protocol from [3] required ten large integers per multiplication proof, the protocol we give in Appendix II requires seven. This is a useful saving in the overall communication cost of the protocol, which is dominated by these ZKPs. Putting the two proofs together (for ν and the W_{ij}) shows that total size of the proof of validity of G_i is

$$7km + O(\log km) \qquad \text{large integers}$$

.

3.4 Tallying and Threshold Decryption

The tallier computes for each j the product of all the homomorphic images that it receives:

$$X_j = \prod_{i=1}^{n} x_{ij} \qquad Y_j = \prod_{i=1}^{n} y_{ij} \pmod{p}$$

and we notice that $Y_j = \gamma^{T_j} h^{R_j}$ and $X_j = g^{R_j}$ where

$$T_j = \sum_{i=1}^{n} G_{ij} \qquad \text{and} \qquad R_j = \sum_{i=1}^{n} r_{ij}$$

so (X_j, Y_j) is an El-Gamal encryption of the desired sum T_j. To decrypt, we broadcast X_j to all clients.

Each client that receives X_j should apply their share of the secret key to it, and send $X_j^{s_i} (\text{mod } p)$ to the tallier. Assume that for each j, the tallier receives at least $t + 1$ responses from some set Λ of clients. Then tallier computes:

$$P_j = \prod_{i \in \Lambda} (X_J^{s_i})^{L_{i,\Lambda}} = g^{sR_j} = h^{R_j} (\text{mod } p)$$

Finally, the tallier computes: $Y_j P_j^{-1} = \gamma^{T_j} \pmod{p}$. Although computing T_j requires taking a discrete log, the values of T_j will be small enough (10^6 to 10^9) that a baby-step/giant-step approach will be practical. This can be done by many of the clients in parallel to speed up the process. In $\sqrt{|T_j|}$ steps, the value of T_j will be found, and the client can send this info directly to the tallier for verification, since it is public.

3.5 Checking Inputs and Tallys

The ZKPs and calculations done by each tallier in our scheme are "publicly" verifiable, as in [5]. However, [5]

gave no scheme for explicitly checking tallys. Checking totals appears to be difficult without seeing the inputs, which is clearly not efficient. A simple approach to efficient checking is to use randomly sampled redundant talliers, and take the majority for each tally. First, assuming $km \leq n$, we compute each of the km totals with a different group of talliers. In the second case of $km > n$, we distribute the km values into n groups so that each group has at most $\lceil km/n \rceil$ totals to compute. The number of groups in either case is $\min(km, n)$.

To choose which talliers lie in which group, we rely on the global coin toss. The number of random bits needed to allocate tallys to groups of talliers is $O(\min(km, n) \log^2(km + n))$. The majority value(s) among the talliers in a group will determine the value(s) used in subsequent calculations. Let α be the fraction of honest talliers. When n_r talliers are chosen at random using global coin tosses, the majority scheme will succeed if most of talliers are honest. Now $\alpha > 0.5$, and the expected number of honest talliers n_h in the sample is $\mathrm{E}(n_h) = \alpha n_r$. The scheme will fail if the number of honest talliers in the sample $n_h < 0.5 n_r$. Taking $\alpha = 0.8$ and using Chernoff bounds, we find that

$$\Pr[n_h < 0.5 n_r] < 0.922^{n_r}$$

Since there are $\min(km, n)$ groups, and we need a total probability of failure of $O(1)$, the probability of failure in any single group should be $\Pr[n_h < 0.5 n_r] = O(1/\min(km, n))$. Then it follows that $n_r = \Omega(\log \min(km, n))$. If p is a bound on the probability of failure in any group, the number of checkers needed for $\alpha = 0.8$ is

$$n_r > 8.5(\log_2 \min(km, n) + \log_2(1/p)) \qquad (3)$$

If we choose instead $\alpha = 0.7$ then the constant above increases from 8.5 to 15. Choosing $\alpha = 0.6$ causes the constant to increase to 50.

4 Protocol

Here we summarize the entire method. As before there are n clients and m items, and A has dimensions $k \times m$. First the procedure for computing a least-squares fit and partial SVD of the training data. Assume A has been initialized to a random matrix in $\mathbb{R}^{k \times m}$. All users know this matrix. Repeat `numiter` times:

1. All clients compute their contribution to the gradient vector, which is $AP_i^T P_i$ for client i. They compute ZK proofs that their data are valid and write all of this to the blackboard. $O(km)$ computation and communication cost per client. The total number of integers written by each client is $8km + O(\log km)$.

2. Using the global coin toss, each tallier chooses a subset of $O(\log n)$ clients. The tallier checks the ZKPs of these clients and posts the results either "OK" or "not-OK" to the blackboard. This requires $O(km \log n)$ computation and computation per tallier.

3. Each tallier reads the results of ZKPs checks in the previous step. For each client with a majority of OK votes, the tallier commits to add that client's data to its total. The tallier reads the global coin toss and chooses a subset of items to total. The total for the chosen items and the valid clients is then written to the blackboard. This requires $O(km \log n)$ communication and computation cost per tallier.

4. Clients compare encrypted totals from approved talliers (those selected by the global coin toss) and if there is a clear majority for a total, they decrypt it using their share of the secret key. They write these to the blackboard. Cost is $O(km \log n)$ computation and communication per client.

5. Talliers collect partial decryptions from clients (which are easily verified using each client's public key) for the data items for which they are responsible, which is $O((km/n) \log n)$ items per tallier. They combine these to produce decrypted totals, still as exponentials. Each tallier then computes the discrete logs of those totals using an $O(\sqrt{n})$ baby-step/giant-step method, and writes these values to the blackboard. These are now fully decrypted coefficients of the gradient of A. Total cost is dominated by $O(km \log n)$ per tallier.

6. Talliers read the blackboard and take the majority vote among approved talliers for gradient coefficients for which they were *not* responsible. At the end of this process, every honest tallier should have a complete copy of the new gradient. This process takes $O(km \log n)$ steps per tallier.

7. The conjugate gradient algorithm also requires a line minimization step (Appendix I). This part of the protocol is a repeat of steps 1-5 above, except that there are only 3 line coefficients (c_i, a_i, b_i) instead of km gradient coefficients. We assume this has been done, and now every honest tallier has a copy of the new gradient and the line coefficients.

8. Talliers update the estimate of A using the decrypted line coefficients and conjugate gradient as described in Appendix I. They also compute the partial SVD matrices D and V. These are written to the blackboard. For efficiency, each tallier does this only for the coefficients for which it is responsible. Cost is $O(km \log n)$ per tallier.

9. Talliers take majority vote for items for which they were not responsible. The result is that all honest talliers have updated values for A, D and V. Cost is once again $O(km \log n)$ per tallier.

As we mentioned earlier, the typical number of iterations is 40-60 for convergence on real collaborative filtering data.

Note that at each step of the protocol, incoming data is checked for validity. In step 2, this is done using ZKP. In step 4 this is through a user's public key which immediately verifies their decryption of the data. In the other steps, verification is through majority vote of approved talliers using the global coin toss. If a sufficient majority of talliers is honest, this yields the correct result with high probability. By totaling the computational effort, we arrive at the following:

Lemma The total computation per client/tallier during one round of the protocol is $O(km \log n)$.

We have not said anything yet about sychronizing this protocol. The shared blackboard makes this a fairly simple process. We can declare each round complete when a pre-specified fraction (e.g. 70%) of clients or talliers have written their data to the blackboard. All honest clients and talliers would then always work on the same data, no matter what was written later. This fraction would need to be determined experimentally, once it was known how many of the possible clients typically participate.

4.1 Experiments

We did not implement the cryptographic protocols above. This would have been reasonably straightforward, but tedious. Since we can prove their desired privacy properties, we would not have learned anything by implementing them. Performance is fairly easy to estimate without implementation, because all of the cryptographic operations have well-characterized running times using, e.g. the CRYPTO++ toolkit, and are much more expensive than other operations.

But it was far from clear whether the numerical method was practical. How fast would it converge on typical data? Would it be sensitive to noise? How large should k be for good predictions? Is it competitive with existing collaborative filtering schemes? Therefore we focussed our implementation on the numerical method.

We tested the numerical method on the EachMovie dataset, a well-known test dataset for collaborative filtering algorithms [2]. This dataset comprises ratings of 1648 movies by 74422 users. Each rating is an integer in the range $0, \ldots, 5$. We normalized the ratings to $-2.5, \ldots, 2.5$ so that there was no zero rating to be confused with an absent rating. We chose 40% of the users at random as a training set. The ratings of these users became the P matrix on which we ran the iterative least-squares procedure.

The implementation was done as a Matlab script file. It was run on a 500MHz processor with 256MB of memory. There were no clients, so all calculations were done on this machine. The dimension of the linear space A was $k = 8$ for these experiments. This was found to give best performance in cross-validation experiments. The average time per iteration was about 1 second, and 40 iterations - the entire training phase for 74422 users - was completed in under one minute. Typical convergence rates were very fast: 10-fold residual error reduction every 10 iterations. The error reductions for 40 iterations ranged from 10^3 to 10^6. For the Eachmovie dataset, if the residual error reduction is at least 10^2 there is no measurable change in the quality of predictions.

The remaining 60% of users were used for cross-validation. For each user, 10 of the items they had rated were set aside, and the remainder used to generate predictions using the method of section 2.1. The average time to generate a recommendation was 0.05 seconds, or 20 ratings per second. Accuracy was very good. The Mean Absolute Error (MAE) is the average of the absolute difference between a prediction and the actual rating of an item by a user. The MAE for our scheme was was 0.96. In [7], several collaborative filtering schemes were compared on the Eachmovie dataset. The best performance by any of the algorithms they studied was an MAE of 0.96 - equal to our method.

Finally we studied the robustness of the scheme by simulating a fraction of clients "dropping out" of various steps of the computation. At each iteration a different random subset of 50% of the clients were discarded from the gradient total. A different random subset of 50% of clients was dropped during the line minimization step. The least-squares algorithm still converged, albeit more slowly and could not achieve residual errors below 0.01 of the initial error. Fortunately, this made no measurable difference to prediction accuracy during cross-validation. This is probably because the dataset is very noisy.

4.2 Communication and Computation Time Estimates

We also estimated the running time and communication costs of the cryptographic protocols based on recent benchmarks for the primitives. Efficient cryptographic tools such as Crypto++ provide all the basic operations we need. The size of the aggregate for the EachMovie dataset is $8 \times 1648 = 13184$ array elements, each element being a pair of field elements. With El-Gamal encryption, typical field elements are 1024-bit integers, while for ECC they are 168-bit integers. The total storage required for the EachMovie

aggregate is 3.4 MB for EG or 500 kB for ECC. Each user's encrypted gradient has the same size as the aggregate, but adding ZKPs to the gradients increase their size to 15 MB (EG) and 2.2 MB (ECC).

There are several steps with complexity $O(km \log n)$ in the protocol, but the dominant step in terms of constants is step 2. To derive the cost of this step, we pick a typical value of $n = 10^5$, which was the case for the Eachmovie dataset. We set $\alpha = 0.8$ as before. Picking an error probability for checking of 10^{-6} allows us to compute the size of the random sample $n_r = 300$ from equation 3. The total amount of communication per client during step 2 is the product of the ZKP size given above by the redundancy n_r. That takes the total communication per client to 4 GB (EG) or 600 MB (ECC).

To determine running times, we use benchmarks for the CRYPTO++ tookit from `www.eskimo.com/~weidai/benchmarks.html`. Their experiments show that EG 1024-bit exponentiations take approximately 10ms. Checking a ZKP as per appendix II requires 11 exponentiations. Multiplying these numbers by the number of proofs to check gives a total time of $13k \times 300 \times 11 \times 10ms$ which comes to about 50k seconds, or 15 hours. The times for ECC are very similar.

These times and communications totals are large, but even without improvement it should be feasible to run one round of the protocol over several days as a background process. Since the user ratings data are changing slowly, a few days latency does not diminish the value of the aggregate.

Finally, the local storage demands of the protocol are quite modest. A client need only work on a single copy of a gradient or the aggregate at a time. Including ZKPs, local storage of 10-50MB should be enough.

5 Statistical Vulnerabilities

While the scheme we described gives good data hiding in a cryptographic sense (beyond disclosure of A), there is still the potential for leakage of information. Such leakage may be "static" or "dynamic" arising respectively from one snapshot of the aggregate (static leakage), or from several snapshots of the aggregate over time (dynamic leakage). We discuss static leakage first.

5.1 Static Leakage

We have treated the entire aggregate A as public data because our scheme for generating ratings (section 2.1) allows the aggregate A to be constructed from a sufficient number of queries. So as long as a recommendation service is running on a model A, that model can be extracted through the query interface. This is also true of the SVD CF scheme published in [14]. The positive aspect of this is that mining the query interface will *only* reveal information about the model A, and not the underlying data. In order to avoid overfitting, the model A is at least a 10-fold compression (size measured as number of elements) of the original ratings data. By adding a small amount of noise to each rating, we can achieve a similar compression in an information-theoretic sense. Typical compression ranges from 10-100 times. So even though it is easy to access A, the amount of information about an individual user's ratings is very limited, at least in an average sense. Most of the original information has simply been lost. But we must guard against concentration of the information.

Although the aggregate includes data from many individuals, some items may have been rated by just a few individuals, and those items can be correlated with others using the SVD data. For instance, if A encoded ratings of web sites based on user visits, personal web sites would leak information because their owners frequent them more than anyone else. Other sites that correlate strongly with a personal site are strong candidates to have been visited by the owner of the site. Highly selective data such as personal web site visits should be filtered out from a scheme like this, as their potential for leaking information is too great. We assume that users are able to exclude any chosen site or locations from their data, and that the system advises them to do so.

The second source of static leakage is sites that have been rated by very few users. If an item has very few raters, correlations between this item and others will disclose much information about those raters' choices. It is therefore desirable to remove items with few raters from the aggregate. A second reason for doing this is that extrapolated ratings for such an item are likely to be inaccurate. The accuracy of ratings of an item will be quite poor unless the number of raters of that item is larger than the dimension of the linear model. If there are fewer, then there is not enough information to localize the item in the k-dimensional ratings space. To deal with this, we suggest using a dynamically-maintained "frontier" of items.

5.2 The Frontier

As well as the model A, we employ an integer-valued vector F called the *frontier*. For elements in the frontier, we maintain only a count of the number of users that have rated them, not a model of user ratings. So let F_i be the count of the number of users who have rated item i. The set of items in the frontier F is typically much larger than the set of items modeled in A. For instance, if F contains k times as many items as A, then the vector F and the matrix A will both contain km elements. With easy extensions to the protocol we described in section 4, the counts in F can all be maintained without disclosing user data. Then the set of items actually handled in the aggregate A at each iter-

ation would be the subset of the m most frequently-rated items from among the items counted in F. It would be an even smaller subset if there are fewer than m items whose count lies above a cardinality threshold (e.g. $2k$) for accuracy purposes. In this way, A would only model ratings of reasonably popular items (items with at least $2k$ raters).

As well as protecting privacy and avoiding inaccurate extrapolations, this scheme allows a much larger set of items to be handled by the system with a small impact on storage and computation. For instance, for the Eachmovie dataset with 1600 items and $k = 20$, maintaining a frontier with $km = 32000$ items would only double the storage needed, and less than double the computation, compared to the basic protocol. Given the typically Zipf-like distributions of number of raters of items, most of the items in the frontier will have very few raters, and would not meet the cardinality threshold. Thus we could not provide accurate extrapolations for them. We can recognize this fact from the values in F, and advise the user of it.

5.3 Static Leakage in other CF systems

First of all, we note that the SVD scheme described in [14] has quite similar properties to ours. Namely, it provides high compression of the original data, and therefore good protection of user data if we guard against the two "information concentration" mechanisms described above. Like ours, it is straightforward to construct the linear model from a sufficient number of queries with [14].

It is more complicated to analyze other schemes. But schemes which do not create an intermediate model like ours are probably very dangerous. For instance, Pearson correlation [7] and personality diagnosis [10] use the entire user dataset to generate new recommendations. What's more, Pearson correlation makes use of a subset of "neighbors" of the current user who have rated several of the same items. The neighbor subset may be extremely small if the querying user has rated only a few items so far. Pearson schemes may simply refuse to return a rating if the neighbor set is empty or too small. An adversay can easily use this to advantage by choosing their number of rated items so that it is just large enough to avoid a "no ratings" message. That means there are just enough items to give an adequate neighbor set, but this neighbor set will be very small, and the ratings the adversary sees will be a weighted average of that very small set of neighbors. It is easy to come up with artificial (and unrealistic) datasets where the entire user dataset can be extracted via queries. Just how well one can do at extracting information from realistic datasets is a matter of some concern, since memory-based methods are in use in some real websites today.

So in summary, schemes based on low-dimensional linear models of ratings data (e.g. SVD) offer quite good protection against static leakage of individual information. For these methods, it makes no difference whether the model is exposed directly, or only via queries. For memory-based methods, there is no intermediate model that limits the information leakage. The potential for leakage via query mining for such methods appears to be severe.

5.4 Dynamic Leakage

The iterative least-squares scheme makes repeated use of user data. Suppose a user contributes to one iteration but not the next. There will be slight numerical differences in the gradient which may not mask the difference caused by that user. The best defence against this problem is to add more randomness. We tested a modification of the numerical method where each user tosses a coin to decide whether to contribute their actual gradient, or a zero vector at each iteration. As we noted earlier, the iterative method still converges with this disturbance. Such an approach should make it very difficult for an adversary to isolate individual data by "sniffing" the changes in A over time. This method of randomization is valid wrt the SVD calculation, because it amounts to a sub-sampling of the dataset. Other randomization methods, such as additive noise, do not have this property[3]

6 Discussion

An important pragmatic issue with our scheme is the management of the recommender community. In order to succeed, this scheme must have a majority of honest clients and talliers. That implies some authentication of the members. Without it, a malicious user could join a community masquerading as many individuals. Ideally, the community might be formed from individuals who actually know each other. This can be extended to include individuals that are *vouched for* by a core community member, etc. The design implications are subtle and we have not explored them. The next most reliable method would be to restrict membership to a known community. For instance, campus or company email might be used in the key setup phase, ensuring that each user has a valid email address within the organization. Beyond social and organizational bounds, community setup and maintenance is more problematic. There are a variety of creative solutions, e.g. having individuals fill out surveys, or receive a password by phone, etc., but these are beyond the scope of our present work. We believe that this problem is a basic one in peer-to-peer computing, and is e.g. being studied in the development of the Java peer-to-peer API, JXTA.

[3]To see this, imagine the dataset is drawn from an elliptical gaussian distribution. Adding sufficient noise will produce a spherical gaussian.

The last question is whether this kind of key-sharing among peers is a good model of security. After reflecting on this work for some time, we believe that the model is not only acceptable, but is very good in many respects. The goals of any privacy scheme should be to protect individuals from unreasonable scrutiny or search without cause. At the same time, it is not socially desirable that criminals be protected from scrutiny once guilt has been established or there is probable cause. A scheme which provides perfect data hiding also provides criminals with effective means to communicate and perhaps perform other kinds of distributed computation. A key escrow scheme like this one places the ability to decrypt information in the hands of individuals. If a single individual or agency has this power, then the possibilities for abuse are many. If a few individuals within an organization have this capability, the protections are better, but there is still the prospect of coercion by outsiders, or communication of these few powerful keys to others. On the other hand, escrow in the hands of many places the community's privacy in the hands of the community. They can also make a judgement about using their keys to decrypt private data in situations where there is a compelling reason to do so, such as suspicion of criminal behavior. Coercion of a large community would be impractical in most situations. Any abuse from within the community would be highly visible to other members, which is itself a strong deterrent.

To summarize, we described in this paper a practical and useful example of computation on encrypted data. Our method reduces a non-linear computation (SVD) to a series of linear steps. It can be implemented fully peer-to-peer. We showed by experiment that the algorithm compares well in accuracy and speed with traditional collaborative filtering methods. We believe that it points the way to a class of practical algorithms that work on encrypted data.

References

[1] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for noncryptographic fault-tolerant distributed computations. In *20th ACM STOC*, pages 1–10, 1988.

[2] Breese, Heckermen, and Kadie. Empirical analysis of predictive algorithms for collaborative filtering. Technical report, Microsoft Research, October 1998.

[3] R. Cramer and I. Damgård. Zero-knowledge for finite field arithmetic. or: Can zero-knowledge be for free? In *Proc. CRYPTO '98*, volume 1462, pages 424–441. Springer Verlag LNCS, 1998.

[4] R. Cramer, I. Damgard, S. Dziembowski, M. Hirt, and T.Rabin. Efficient multiparty computations secure against an adaptive adversary. In *Proc. EuroCrypt '99*, volume LNCS 1592, pages 311–326. Springer-Verlag, 1999.

[5] R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. *European Transactions on Telecommunications*, 8(5):481–490, 1997.

[6] K. Goldberg, D. Gupta, M. Digiovanni, and H. Narita. Jester 2.0 : Evaluation of a new linear time collaborative filtering algorithm. In *22nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, August 1999. Poster Session and Demonstration.

[7] J. Herlocker, J. Konstan, A. Borchers, and J. Riedl. An algorithmic framework for performing collaborative filtering. In *Proc. ACM SIGIR*, 1999.

[8] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[9] T. Pedersen. A threshold cryptosystem without a trusted party. In *Eurocrypt '91*, volume 547, pages 522–526. Springer-Verlag LNCS, 1991.

[10] D. Pennock and E. Horvitz. Collaborative filtering by personality diagnosis: A hybrid memory- and model-based approach. In *IJCAI Workshop on Machine Learning for Information Filtering, International Joint Conference on Artificial Intelligence (IJCAI-99)*, Stockholm, Sweden, August 1999.

[11] E. Polak. *Computational Methods in Optimization*. Academic Press, New York, 1971.

[12] T. Rabin and M. Ben-Or. Verifiable secret-sharing and multiparty protocols with honest majority. In *21st ACM STOC*, pages 73–85, 1989.

[13] E. M. Rogers. *Diffusion of Innovations, Fourth Edition*. The Free Press, 1995.

[14] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. Riedl. Application of dimensionality reduction in recommender system – a case study. In *ACM WebKDD 2000 Web Mining for E-Commerce Workshop*, 2000. Full length paper.

Appendix I: SVD via Vector Addition

Recall that the matrix A minimizes the squared error $e = \mathrm{tr}(PP^T) - \mathrm{tr}(PA^TAP^T)$. To compute A, we use a conjugate gradient algorithm. This allows us to compute A in a number of rounds which equals the number of iterations of the conjugate gradient algorithm (typically 40-60 for our application). This requires only *summation* of individual user data, which can be done using the cryptographic homomorphism described earlier.

Since $\mathrm{tr}(PP^T)$ is fixed, minimizing e amounts to maximizing $\mathrm{tr}(PA^TAP^T)$. Since A is an orthonormal representation of a vector space, it is subject to the constraint $AA^T = I$. Using Lagrange multipliers, we need to maximize

$$\mathrm{tr}(PA^TAP^T + \Lambda(AA^T - I))$$

where Λ is a $k \times k$ matrix of multipliers. The gradient of this function is

$$G = 2AP^TP + (\Lambda + \Lambda^T)A = 2AP^TP + SA$$

where S is a symmetric matrix. For a vector G to lie on the tangent manifold defined by $AA^T = I$, we must have

$$\lim_{\epsilon \to 0}((A + \epsilon G)(A^T + \epsilon G^T) - I)/\epsilon = 0$$

so

$$AG^T + GA^T = 0$$

substituting for G and solving gives

$$S = -2AP^T PA^T$$

and therefore:

$$G \ = \ 2AP^T P(I - A^T A)$$

Now suppose that A is the current approximation to the best linear fit to P. Let P_i denote the $1 \times m$ matrix of data from the i^{th} user. Then $P^T P$ can be computed as $P^T P = \sum_{i=1}^{n} P_i^T P_i$ and the gradient as:

$$G \ = \ \sum_{i=1}^{n} AP_i^T P_i(I - A^T A)$$

or $G = \sum_{i=1}^{n} G_i$ where $G_i = AP_i^T P_i(I - A^T A)$ is the i^{th} user's contribution to the gradient. This is a key point. The fact that the gradient is expressible as a sum of contributions from each user makes it possible to compute A using only addition of user data (and therefore using cryptographic homomorphism). We assume that the current approximation $A^{(j)}$ to A is known to everyone at the j^{th} iteration. Then user i computes their contribution $G_i^{(j)}$ to the gradient $G^{(j)}$ using the expression above. The user sends an encrypted copy of $G_i^{(j)}$ to the tallier(s), which then sums all user contributions to yield the encryption of $G^{(j)}$. The work to compute each user's contribution is $O(km)$, assuming manifold correction is done on the tallier.

The next phase of conjugate gradient is calculation of the extremum along the gradient direction. For this we need a quadratic approximation to the value of the error function $e(t)$ a distance t along the gradient direction. Strictly speaking, we actually move along a quadratic curve that tracks the curvature of the manifold. The derivation up to second order terms of $e(t)$ is tedious but straightforward. The quadratic approximation is $e(t) \approx e_0 + e_1 t + e_2 t^2$ and the extremum occurs at $t \approx -e_1/(2e_2)$. We need the two terms e_1 and e_2. For convenience, we will break e_2 into two components $e_2 = a + b$, and let $e_1 = c$. Then the three quantities we need are:

$$c = -2\mathrm{tr}(PG^T AP^T)$$
$$a = -\mathrm{tr}(PG^T GP^T)$$
$$b = \mathrm{tr}(PA^T GG^T AP^T)$$

and we note that each term has the form $\mathrm{tr}(PXP^T)$, and involves private data P. All the data in each X is public, involving the current approximation A and the gradient G. Now notice that all products can be computed as

$$\mathrm{tr}(PXP^T) = \sum_{i=1}^{n} P_i X P_i^T$$

where P_i is user i's data as before. So just as with the gradient computation, the line minimization step can be done with summation of encrypted data. Each user computes their contributions to (c, a, b) and sends them encrypted to the tallier(s). The computation per client is $O(km)$ for these terms.

The tallier sums all the contributions and computes final encrypted totals (c, a, b). When these totals are decrypted (next section), the tallier(s) can compute the gradient step size[4] $t_s = -c/(2(a + b))$. The tallier then increments the current estimate $A^{(i)}$ by marching along in the gradient direction. When corrected for curvature, the new estimate is

$$A_0^{(i+1)} = A^{(i)} + t_s G - \frac{1}{2} t_s^2 GG^T A^{(i)}$$

Because of numerical error, $A_0^{(i+1)}$ will not have orthonormal rows. To obtain $A^{(i+1)}$ we apply a standard orthonormalization scheme, such as Gramm-Schmitt, to $A_0^{(i+1)}$.

Conjugate Gradient

A simple gradient scheme such as described above will have slow (sub-quadratic) convergence. Conjugate gradient is a good way to accelerate a minimization, and gives quadratic convergence. The conjugate gradient is a moving average of gradient directions. It requires a "one-step" memory of the previous gradient, and requires only slight modification of the tallier code (no changes are needed to inter-processor communication). We used the Polak-Ribiere formula [11] to compute a generalized gradient H based on G and the H, G pairs from earlier iterations. This is a standard technique and we do not describe it here.

There is one complication with applying conjugate gradient. We are working in a "moving" coordinate system. That is, every gradient G or H has coordinates which are based on the current approximation $A^{(i)}$. Gradients at two different values of A cannot be compared or combined because they will not satisfy the condition $AG^T + GA^T = 0$ at the other A. Conjugate gradient requires a weighted sum of gradients from two different time steps. These gradients are in different coordinate systems and therefore cannot be combined. Fortunately, the set of orthonormal A has a Lie Algebra structure and there is a simple way to transform

[4] For large steps t_s which occur early in the optimization, the estimator $e_2 = a + b$ is ill-conditioned. Instead we use the less accurate but more stable estimator $e_2 = b - a$. The switch to $e_2 = a + b$ should be made when convergence slows

between one coordinate system and another. Let $A^{(i)}$ and $A^{(i+1)}$ be two orthonormal matrices (and coordinate systems). To transform a gradient G expressed in $A^{(i)}$'s coordinate system to $A^{(i+1)}$, we compute:

$$G' = A^{(i+1)}(A^{(i)T}G - G^T A^{(i)})$$

when both the standard and conjugate gradients from the previous step are transformed in this way, they can be used in the Polak-Ribiere formula [11].

Computing Singular Value Decomposition

We can extend the least-squares scheme to compute a partial singular value decomposition (SVD) of P. Recall that an SVD is a factorization of P as $P = UDV^T$ where U and V have orthonormal columns and D is a diagonal matrix with real non-negative entries sorted in descending order. It is known that the first k columns of U give the optimal k-dimensional approximation to the columnspace of P, while the first k rows of V^T give the best k-dimensional approximation to the rowspace of P. It follows that if A has been computed as in the previous sections, then rowspan(A) equals the rowspan of the first k rows of V^T. We can recover these rows by computing the eigenvectors and eigenvalues of a small ($k \times k$) matrix.

Let $B = AP^T PA^T$ and form the eigendecomposition of B as $B = WEW^T$ where E is a diagonal matrix of eigenvalues, and W is the matrix whose columns are the corresponding eigenvectors. Notice that B is positive semidefinite, and assume the real eigenvalues in E are sorted in descending order. Then

$$D^2 = E \qquad V = W^T A$$

The information needed to compute D and V is available at the tallier. The matrix B is easily computed from the gradient G as $B = \frac{1}{2}G(I - A^T A)^{-1}A^T$. Since B is $k \times k$ and k is small (typically $k < 20$), the eigenvalue calculation is inexpensive. Computing D and V from the eigenvalue decomposition is also inexpensive, and requires $O(k^2 m)$ steps.

Note that it is not possible to compute the matrix U in the SVD. This is intentional. U contains information about specific users. The i^{th} row of U encodes user i's preferences in the k-dimensional subspace. We do not store information that would allow U to be recovered. Both D and V however, contain useful information about patterns of user preferences mapped onto the data items.

Appendix II

To simplify notation, suppose A, B respectively encrypt a and b, then we give here a ZKP that $b = a^2$. This protocol is a straightforward adaption of the multiplication prototocol from [3]. The idea is to show that if the "message" encypted in A is the a^{th} power of γ, then the message in B is the a^{th} power of the message in A. We have to do this without revealing a of course.

1. Prover knows s_a, s_b such that

   $$A = (g^{s_a}, \gamma^a h^{s_a}) \bmod p \qquad B = (g^{s_b}, \gamma^b h^{s_b}) \bmod p$$

   Prover chooses $x, r_a, r_b \in \mathbb{Z}_q$ uniformly at random and sends

   $$C_a = (g^{r_a}, \gamma^x h^{r_a}) \qquad \text{and} \qquad C_b = A^x (g^{r_b}, h^{r_b})$$

   to Verifier.

2. Verifier chooses $c \in \mathbb{Z}_q$ uniformly at random and sends it to Prover.

3. Prover computes

   $$v = ca + x \pmod q$$
   $$z_a = cs_a + r_a \pmod q$$
   $$z_b = c(s_b - as_a) + r_b \pmod q$$

   and sends v, z_a, z_b to verifier.

4. Verifier checks that

   $$(g^{z_a}, \gamma^v h^{z_a}) = A^c C_a$$
   $$A^v (g, h)^{z_b} = B^c C_b$$

   and accepts iff both identities hold.

This protocol is honest-verifier zero-knowledge. The simulation is: Choose $c, v, z_a, z_b \in \mathbb{Z}_q$ independently and uniformly at random, and compute C_a and C_b to satisfy step 4 above. Both the protocol and the simulation have the same probability distribution on conversations $(C_a, C_b, c, v, z_a, z_b)$.

From two accepting conversations $(C_a, C_b, c, v, z_a, z_b)$ and $(C_a, C_b, c', v', z'_a, z'_b)$ we can recover a and s_a via:

$$a = (v - v')(c - c')^{-1} \pmod q \qquad \text{and}$$
$$s_a = (z_a - z'_a)(c - c')^{-1} \pmod q$$

and then s_b as

$$s_b = (z_b - z'_b)(c - c')^{-1} + as_a \pmod q$$

finally by equating powers of γ in the last test in step 4, we see that: $ca^2 + ax = cb + ax$ from which it follows that $b = a^2$.

If this protocol is implemented non-interactively, both prover and verifier will use a secure hash function to compute c from A, B, C_a, C_b. Then the non-interactive proof consists of C_a, C_b, v, z_a, z_b, which is a total of seven $\bmod p$ or $\bmod q$ integers.