The TROOTH Recommendation System

Keno Albrecht, Roger Wattenhofer
Computer Engineering and Networks Laboratory
ETH Zurich, 8092 Zurich, Switzerland
{kenoa, wattenhofer}@tik.ee.ethz.ch

Abstract

In this paper we present TROOTH *as a robust, partially decentralized, collaborative, and personalized recommendation system. We examine a voting scheme where users have to assess a continuous stream of items to be either* good *or* bad*—either manually or, assisted by* TROOTH*, automatically based on previous votes. For this purpose,* TROOTH *implicitly creates* special interest groups *containing users who share similar opinions expressed by assenting votes. To evaluate an item with* TROOTH*, a user trusts those votes most which have been cast by other users in the same group.* TROOTH *has been implemented in the* SPAMATO *spam filter system where it is used in the context of collaborative spam filtering.*

1 Introduction

Internet users of today's online shops and information portals are often stunned by the vast amount of product and service alternatives, unable to decide which product to buy or service to approve. Recommendation systems promise to help customers find a way in this information overload. By tracing and analyzing explicit or implicit data on user actions, such as the visit of web pages, the order of products, or the assessment of experienced services, it is possible to recommend particular objects which are assumed to match a user's expectation.

E-commerce sites as well as their customers both benefit from a recommendation system. Online shops, such as Amazon.com, greatly profit by suggesting individually chosen books to users being satisfied even with unexpected purchases. Ebay.com employs a recommendation scheme to comfort people buying articles from strangers by having both rate each other after their transaction.

But recommendation systems are not restricted to online business. They can be applied to a much broader application domain, including social bookmarking, software evaluation, and spam filtering—thus, in all areas where users can annotate items and contribute their opinions to a collaborative environment.

Recommendations systems have been influenced by the work on *collaborative filtering* techniques. Generally, a sparse $m \times n$ matrix for m users and n items is considered in which only a few entries reflect the users opinions on the items. The goal of a collaborative filtering system is to ease the task of manually choosing a new item by automatically recommending suitable items to users who have not rated them previously. For this purpose, this technique derives future decisions from assenting opinions in the past.

In this paper, we adopt this notion in that we also assume users to vote for items, *votes* being either "good" or "bad." Additionally, we explicitly introduce *trust values* between users; a higher value denotes more confidence in a user. By this, we implicitly define *special interest groups* (SIGs) which contain users with assenting opinions. In contrast to collaborative filtering, we are not particularly interested in recommendations for arbitrary items—instead, our system computes *evaluations* of specific items, being either *good* or *bad* (or *unknown* if an item cannot be evaluated). Furthermore, we assume a continuous stream of items which a varying number of users want to have assessed. Thus, the number of users and items is not predefined or bounded to any m and n respectively.

These ideas are incorporated in TROOTH which we present as a robust, partially decentralized, collaborative, and personalized recommendation system. It is robust as it withstands malicious users who are trying to cheat the system, partially decentralized as clients are explicitly involved in the voting and evaluation processes, and collaborative and personalized as users interact with each other for collective benefits.

As an example, we present the collaborative spam filter in the SPAMATO spam filter system (see Section 5): A user constantly receives emails. For the user, it is unknown whether the emails are legitimate or spam messages. Looking at them manually is *expensive* in that a user has to spend time on it and also involves some risk as an inexperienced user might open dangerous attachments or fall for phish-

ing attacks. Therefore, SPAMATO separates spam messages from legitimate messages before they are delivered to the user. For this purpose, SPAMATO relies on TROOTH which calculates an evaluation based on reports for spam messages and revokes for legitimate messages that have been sent by users previously.

The remainder of this paper is organized as follows. In the next section, we discuss related work. In Section 3, we describe some basic notions which we use to present the TROOTH recommendation system in Section 4. Finally, in Section 5, we show how to embed TROOTH in our SPAMATO spam filter system.

2 Related Work

All work on collaborative filtering shares the concept of predicting future user behavior or recommending suitable choices based on historical data. One of the earliest work on collaborative filtering systems is Tapestry [6] which uses a SQL-like language to filter manually annotated (electronic) messages. The focus of current research lies in different areas; [9, 7, 2] study several approaches and give a good overview.

We share some ideas of collaborative filtering but focus on specific practical aspects. For this purpose, we only provide a heuristic to evaluate an item for a user. We do not provide mathematical analysis as in [8, 2].

In contrast to *offline* algorithms, such as [8, 11, 9], that usually recommend a single item to a user based on a fixed set of votes, we consider a continuous stream of items which a user has to assess. These items can arrive at anytime, making predictions time-dependent. Furthermore, users cannot be asked to *train* a server-based system in order to increase the rate of correct predictions; items are always selected client-side. We also emphasize the role of clients in that we store only a minimum of data on the server and evaluate items on clients. We differ from other *online* approaches, such as [5, 3], in that we do not consider a round-based synchronous model. Again, in TROOTH, users can vote for items at any time.

Systems, such as employed by Amazon [10], which recommend articles to clients differ from ours in two aspects. First, Amazon chooses a fast, server-based approach while we explicitly integrate clients to evaluate items. And second, we do not recommend items to users but evaluate them when they arrive.

Other collaborative spam filtering approaches also rely on trust systems to distinguish between trustworthy and malicious users. Razor [14] uses a server-based AIMD approach, while products such as Cloudmark's Desktop [4] or IBM's SpamGuru [13] might entail a manual, administered clustering.

3 Preliminaries

In this section, we describe some basic primitives which we use throughout this paper.

In this paper, the general aim of *recommendations* is to express the evaluation of items, such as products, people, or emails, to be either *good*, *bad*, or *unknown* by allowing users to classify the item as *good* or *bad*.[1] Given an item, a user first tries to revert to a recommendation to check whether it is *good* or *bad*. If the evaluation has been *unknown*, she has to manually assess the item. Afterwards, she casts a vote to express her assessment.

In this section, we assume that a pre-defined, globally valid evaluation for an item exists which has to be exposed for each item. In Section 4, we revise this assumption and instead calculate *individual opinions* about each item.

3.1 Evaluation Functions

A global evaluation of an item can be derived from all *user votes* in various ways. For instance, using a simple *majority* evaluation, the overall categorization of an item is *good* if a majority of all users ($> 50\%$) votes in favor of the item, *bad* if a majority votes against the item, and *unknown* if the number of *good* and *bad* votes are equal.

For the *threshold* evaluation function, we let V be the set of all user votes for an item, and, thereof, V_g be the set of all good votes. Additionally, $\rho_g := \frac{|V_g|}{|V|}$ denotes the fraction of the good votes of all votes. We then define the two thresholds $h_g \in R_0^+$ and $h_b \in R_0^+$ such that $0 \leq h_b \leq h_g \leq 1$. The threshold evaluation function can now be expressed as

$$threshEval(V) = \begin{cases} good & \text{if } h_g < \rho_g \leq 1, \\ bad & \text{if } 0 \leq \rho_g < h_b, \\ unknown & \text{if } h_b \leq \rho_g \leq h_g. \end{cases}$$

Please note that the majority evaluation is the special case where $h_g = h_b = 0.5$.

It is easy to extend this *simple* voting scheme from the set $\{good, bad, unknown\}$ to a more general range where votes and evaluations can be in the interval $[0, 1]$. In this case, we define $\rho = \frac{\sum_{v \in V} v}{|V|}$ to denote the average of all vote values. Then, we can apply the threshold evaluation function with ρ instead of ρ_g or directly use ρ as the result of the evaluation. It is obvious that the simple voting scheme can be derived by using votes with values of 0 (*bad*) and 1 (*good*) only.

[1] We assume that a user who does not know how to classify an item does not vote at all.

3.2 Weighting Votes With Trust Values

So far, votes or rather users have been considered to be equally important. In real life, however, it can be beneficial to apply different weights to votes or, in other words, to consider some users to be "more equal than others." Since we assume the existence of a single, pre-defined evaluation for each item, users who often agree with the majority of user should be trusted more than those who regularly dissent.

Ideally, this means trying to separate users into two groups: One group contains those users who are *trustworthy* and the other group those who are *malicious*. Practically, it is possible to approximate these groups by introducing *trust values* for each user that are adjusted whenever new information is available. Then, instead of simply summing up equal *good* (and *bad*) values as before, each vote is previously weighted with the trust value of the associated user. For this approach to make sense, we generally assume that the group of trustworthy users are a majority, or more precisely: that those users who agree with the majority are trustworthy.

The *Additive Increase, Multiple Decrease* (AIMD) approach takes user specific and automatically adjusted trust values into account. When all users have cast their votes, the trust values are modified. Using AIMD, users who voted *correctly*, that means in accordance with the majority, are awarded by slightly increasing their trust values. On the other hand, users whose votes do not comply with the majority are punished by harshly decreasing their trust values.

Formally, let e^i denote the evaluation of an item i, t_u the trust value of a user u, and v_u^i the vote of a user u for item i, where $u \in U^i$ and U^i the set of all users who have cast a vote for item i. The new trust value t'_u is calculated as follows ($inc \in N^+$, $dec \in R_0^+$, $dec < 1$):

$$\forall u \in U^i : \quad t'_u := \begin{cases} t_u + inc & \text{if } v_u^i = e^i, \\ t_u \cdot dec & \text{if } v_u^i \neq e^i. \end{cases}$$

Initially, we set t_u to $t_{default}$, for instance 1, and demand t_u to be smaller than a maximum trust value t_{max}.

Please note that we are rating now in two different domains: On the one hand, we want to evaluate items by having unknown users vote *good* or *bad* for it. On the other hand, we want to calculate trust values for unknown users to make votes more reliable. The voting (and thus also the evaluation) is actively performed; trust values are implicitly generated. While in principle it is possible to let users choose whom they want to trust, in reality this is considered too involved.

Implementation Issues

For applications like the collaborative spam filter system SPAMATO (see Section 5), the voting for an item (in this case, a message) and the categorization of it (spam/not spam) will not take place at a single point in time. Instead, users can always vote for an arbitrary message, and SPAMATO classifies a message whenever it arrives in a user's inbox. Additionally, not all users vote for all items, since not all users receive the same messages. Therefore, user votes have to be stored for later usage and the evaluation of an item has to be recalculated whenever a new vote has been cast. In other words, evaluations are time-dependent. Furthermore, users must not be able to vote more than once for the same item or a scheme must exist to handle multiple votes in a reasonable way. Finally, users casting a vote have to be authenticated to prevent manipulation.

Implementing the AIMD approach or weighting algorithms in general entails some difficulties. As users can vote at any time, the update of trust values either has to be conducted at a specific point in time, or needs to consider earlier changes associated with the same item. While an algorithm for the former solution can calculate the new values by knowing few variables only, such as the time of the first vote, the current time, and the number of votes so far, it obviously ignores later votes and thus important information to provide fair trust values. On the other hand, adjusting the trust values only once reduces the complexity of the system and saves server resources. The latter solution, however, means that we need to manage extensive historic information about the voting process. Additionally, trust values have to be updated every time a new vote is cast, thus, increasing the demand for server resources. In both cases, the server has to store the trust values for each user and the overall evaluation of each item—to avoid instant time- and resource-consuming calculations whenever these values become necessary.

4 The Trooth System

In this section, we introduce TROOTH as a robust, partially decentralized, collaborative, and personalized voting and trust system.

4.1 Motivation

In Section 3, we have assumed that it is possible to globally evaluate an item—that an overall evaluation exists which coincides with the votes of all trustworthy users. But the separation of users into groups of trustworthy and malicious users, as described in Section 3.2, often is too harsh.

Instead, we believe it is more reasonable to individually evaluate an item for each user separately. A user does not distinguish between trustworthy and malicious users anymore, but between users who generally vote in accordance and those who do not. Thus globally seen, users are implicitly separated into several *special interest groups* (SIGs)

who share a "similar opinion" rather than to discriminate them with the "black & white" scheme described before.

Please notice that we still believe that trustworthy and malicious users exist. While the former describe users who really try to express their opinion, the latter usually vote against the common sense and try to deceive the system, probably for personal benefits. We also consider users who make failures and others who just do not understand how to operate a voting system. So generally, from a user's point of view, all these categories can be reduced to assenting and dissenting users only. For simplicity, in this section we use the term *malicious* for byzantine as well as incautious and unaware users.

Depending on the voting domain, the number of groups might vary significantly between only a few and tens or more. Although we expect groups to be rather large and overlapping, in the extreme, each user might trust only herself so that the number of groups equals the number of users. But this especially expresses the strength of the system: Even if all except one user are malicious, this one will (eventually) figure out not to trust anybody except herself. Thus, the system can even serve different minorities with satisfying results while approaches that assume the existence of an objective evaluation cannot.

Since TROOTH does not compute a global evaluation of an item for all users, it is possible to reduce the consumption of server-side resources to a minimum. Therefore, in TROOTH we store only (item,user,vote)-tuples server-side and calculate user specific trust values client-side as we show in the next two sections.

4.2 Managing Votes and Trust

As in structured peer-to-peer systems, we assign each item and each user a unique identifier from an interval $[0, ..., N]$, organized as a "ring." Thus, we can use the notions of *clockwise* and *anti-clockwise* to denote neighbors on the ring. In Section 5.2, we show how user IDs (and signatures to authenticate users) are generated in SPAMATO.

4.2.1 The Voting Process

When a user votes for an item, she sends her opinion (*good* or *bad*) to the TROOTH server and locally adapts the trust values for other users who voted for the same item. In more detail, the voting process takes the following steps:

- User u_0 sends a vote v_0 for an item i to the server where the (i, u_0, v_0)-tuple is stored.
- The server assembles two lists which are populated with the identifiers of other users who voted *good* (list G) and *bad* (list B) for item i before. Each list contains a maximum of k user IDs that are numerically nearest to u_0 in respect to the ring formation. The lists G and B are sent to the client.
- User u_0 locally adapts the trust values of the users sent to her by increasing the trust values of those users who agreed with her own vote v_0 and decreasing the trust values of those users who voted against it (using the AIMD approach described in Section 3.2 or any other weighting scheme).

4.2.2 The Evaluation Process

To classify an item, *good* and *bad* votes from the server are weighted with the client-side stored trust values. In detail, the following steps are performed for the evaluation process:

- User u_0 sends a query for item i to the server.
- The server returns two lists containing identifiers of users who voted *good* (list G) and *bad* (list B) as in the second step of the voting process described above.
- User u_0 extracts the $l \leq k$ most trustworthy users of each list, resulting in the lists $G' \subseteq G$ and $B' \subseteq B$.
- Finally, the classification can be calculated using the threshold evaluation function described in Section 3.2 with $V' = G' \cup B'$ (or rather all weighted votes of the selected users).

4.2.3 User Specific Parameters

The size k of the *good* and *bad* lists returned by the server, l that denotes the number of the most trusted users to select, inc and dec as parameters of AIMD, and the values of h_g and h_b for the threshold evaluation function are user specific values. Thus, the user is able to further configure the processes to some extent on the client-side.

4.2.4 Discussion

In the second step of both algorithms, the server returns about $k/2$ clockwise and anti-clockwise neighbors of user u_0 for each list. If there are less then k other users who voted *good* (*bad*) for item i, we return only that many without any loss in quality. Assuming that users usually vote for the same *type* of items[2], it is reasonable to believe that the total number of trust values that have to be handled client-side is bounded. This means that each user generally stores

[2]Regarding e-mails, for instance, users who are collected on the same e-mail address list often get the same spam messages. Regarding products, Amazon-like "Users who bought this product also bought..." statements also underline our assumption that users will often vote for the same type of items.

only a small subset of all users who share the same opinion. It is also an advantage that a user's vote can only affect those users in the implicit neighborhood. Thus, the impact of a possibly malicious user trying to cheat the system is limited. On the other hand, though, the rather high consumption of bandwidth for each voting and evaluation operation can be regarded as a drawback.

By choosing only the most trustworthy users in the third step of the evaluation process, we decrease the influence of unwanted users to a minimum. As an additional optional step, trust values can be adjusted after the evaluation process similar to the last step in the voting process. Doing so would amplify the influence of trust values even more. Furthermore, the calculated evaluations could automatically be sent as a personal vote to the server. If the user does not agree with the evaluation, she would (immediately or after some time) send her correct opinion rejecting the old one.

Please note that a malicious user who tries to gain a high trust value in order to manipulate the evaluation process, previously would have to "play by the rules" for a long time and, thus, helping other users more than harming the system. Furthermore, to manipulate a particular user (or a group of users with assenting opinions), it is necessary to, first, get an ID that is near that of the user, and second, to know which items the user is "interested" in. While attacking one particular user will be hard, it is almost impossible to oppose against many groups or even all users at once. Thus, TROOTH significantly reduces the impact of malicious users in the evaluation process.

4.3 The Majority Heuristic

We introduce the *majority heuristic* which can be applied as a special case when many users have almost unanimously decided about an item. To use it, one should have ruled out the chance of malicious users being a majority.

More formally, let $votes_{min}$ denote the minimal number of users who have to cast a vote for an item i and let ρ_{dv} be the maximal allowed fraction of *dissenting* votes. Additionally, V denotes the set of all votes for the item, and V_g and V_b are the sets of *good* and *bad* votes respectively. Then, the majority heuristic can be applied instead of the normal case described in Section 4.2 if $|V| \geq votes_{min}$ and either $\frac{|V_g|}{|V|} \leq \rho_{dv}$ or $\frac{|V_b|}{|V|} \leq \rho_{dv}$.

In the voting process, the server still stores the vote of a user for an item. But the server does not return any data and the client, therefore, cannot adjust any trust values.

In the evaluation process, the server sends only the number of *good* and *bad* votes to the client. Therefore, the client is not able to select her most trusted users anymore; all votes count the same. The evaluation for the item is calculated using the majority or threshold evaluation function.

As in Section 4.2.3, the user can completely configure the processes: The total number of voters and the fraction of dissenting voters have to be sent to the server; the parameters h_g and h_b can directly be adjusted in the client.

The majority heuristic clearly simplifies the voting and evaluation processes by sending less information between client and server and, thus, also saving bandwidth. By doing so, it slightly reduces the reliability of the classification since trust values are not considered anymore. But this can be neglected since there are almost none dissenting votes, and malicious users cannot be a majority.

5 Applications

In this section, we introduce the spam filter system SPAMATO which utilizes TROOTH in the context of collaborative spam filtering. First, we give a short overview of SPAMATO. After that, we describe the *Spamato Authentication and Authorization System* which we use to generate unique user identifiers for TROOTH. And finally, we show how TROOTH is integrated into our collaborative spam filter to distinguish between spam and legitimate messages.

5.1 The Spamato Spam Filter System

SPAMATO [1] is a client-side spam filter system that can be extended using the provided plug-in mechanism. As an add-on, it can be embedded in common email clients such as Outlook or Thunderbird. Emails arriving in a user's inbox are automatically checked by several spam filters, and detected spam messages are moved to a special folder. The user can interact with SPAMATO by manually reporting messages which have not been identified as spam and revoke messages which have falsely been identified as such. This way, a user collaborates with the system, sending feedback (votes) to local and central components. SPAMATO is available for download at: `http://www.spamato.net`.

5.2 The Spamato Authentication and Authorization System

The *Spamato Authentication and Authorization System* (SAAS) is used to create unique identifiers for users who want to interact with the TROOTH system. Additionally, SAAS generates a public/private key pair with which users (automatically) sign their votes to prevent any cheating and manipulation attempts.

Since SPAMATO (and therefore SAAS) is embedded into an email client, SAAS can make use of the authentication process between the email client and the server. In other words, if a user is able to receive an email that has been sent to him via SAAS, the user is "authenticated" also for TROOTH.

In more detail, the first time SPAMATO is started, the SAAS client locally generates a public/private key pair. The public part of this pair and the user's email address is sent to an SAAS server (using a TCP connection) which in turn sends a random *challenge email* to the stated address. On receiving this challenge email, the client signs the message with its private key and sends it back to the server (again using a TCP connection). After that, the user is fully registered with the SAAS server which stores the user's public key and the (hashed) email address.

Please notice that the actual implementation is slightly more complicated to allow for a reregistration of users who want to use the same SAAS user account, for instance, on several machines. Additionally, the TROOTH and SAAS servers need to exchange data so that the TROOTH server can validate a user's signature. See [12] for a detailed description.

5.3 Collaborative Spam Filtering

The *Earl Grey* filter is a collaborative URL filter. When receiving a new message, it collects all URLs in the message, extracts the domains, and calculates a hash value of them. This hash value is sent to the Earl Grey server which queries a database to find out whether the message is spam or legitimate. Entries in the database are collaboratively inserted by users who report "spam" or revoke "legitimate" emails (or rather the calculated hashes). Thus, users help each other to filter spam messages.

Since not all users define the term "spam" equally—some also declare unwanted newsletters to be spam while others like to read about online drug stores—clearly, a system like TROOTH is necessary to handle these different opinions.

In the context of TROOTH, the hash values are the identifiers for items, users are identified with their email addresses (or their SAAS public key), and reports and revokes correspond to *bad* and *good* votes. As said before, to prevent malicious users from harming the system, votes are signed with a user specific private key. Additionally, the Earl Grey server ignores multiple reports/revokes and removes contrary votes for the same message and user.

The Earl Grey filter is fully implemented and runs for more than one year with several users without failures.

Acknowledgment

The authors would like to thank Simon Schlachter for implementing TROOTH [12], Nicolas Burri for the initial idea and fruitful discussions, and all users of SPAMATO for using it.

References

[1] K. Albrecht, N. Burri, and R. Wattenhofer. Spamato – An Extendable Spam Filter System. In *Proceedings of 2nd Conference on Email and Anti-Spam (CEAS)*, 2005.

[2] B. Awerbuch, Y. Azar, Z. Lotker, B. Patt-Shamir, and M. R. Tuttle. Collaborate With Strangers To Find Own Preferences. In *Proceedings of 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2005.

[3] B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. Tuttle. Improved Recommendation Systems. In *Proceedings of 16th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2005.

[4] Cloudmark Desktop. www.cloudmark.com.

[5] P. Drineas, I. Kerenidis, and P. Raghavan. Competitive Recommendation Systems. In *Proceedings of 34th ACM Symposium on Theory of Computing (STOC)*, 2002.

[6] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using Collaborative Filtering to Wave an Information Tapestry. *Communications of the ACM*, 35(12):51–60, 1992.

[7] J. L. Herlocker, J. A. Konstan, A. Borchers, and J. Riedl. An Algorithmic Framework for Performing Collaborative Filtering. In *Proceedings of the 1999 Conference of the American Association of Artificial Intelligence (AAAI)*, 1999.

[8] J. Kleinberg and M. Sandler. Convergent Algorithms for Collaborative Filtering. In *Proceedings of ACM Conference on Electronic Commerce (EC)*, 2003.

[9] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Recommendation systems: a probabilistic analysis. In *Proceedings of 39th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1998.

[10] G. Linden, B. Smith, and J. York. Amazon.com Recommendations, Item-to-Item Collaborative Filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

[11] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Analysis of Recommendation Algorithms for E-Commerce. In *Proceedings of ACM Conference on Electronic Commerce (EC)*, 2000.

[12] S. Schlachter. Spamato reloaded. Master thesis, Federal Institute of Technology Zurich (ETHZ), 2004.

[13] R. Segal, J. Crawford, J. Kephart, and B. Leiba. SpamGuru: An Enterprise Anti-Spam Filtering System. In *Proceedings of the First Conference on E-mail and Anti-Spam*, 2004.

[14] Vipul's Razor. http://razor.sourceforge.net.