Capturing the Moment: Lightweight Similarity Computations

Georgios Damaskinos
EPFL
georgios.damaskinos@epfl.ch

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Rhicheek Patra
EPFL
rhicheek.patra@epfl.ch

***Abstract*—Similarity computations are crucial in various web activities like advertisements, search or trust-distrust predictions. These similarities often vary with time as product perception and popularity constantly change with users' evolving inclination. The huge volume of user-generated data typically results in heavyweight computations for even a single similarity update.**

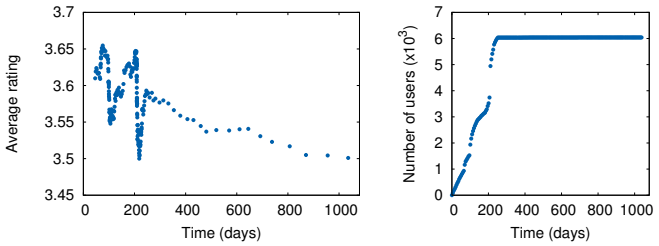
We present I-SIM, a novel similarity metric that enables lightweight similarity computations in an *incremental* and *temporal* manner. Incrementality enables updates with low latency whereas temporality captures users' evolving inclination. The main idea behind I-SIM is to disintegrate the similarity metric into mutually independent time-aware factors which can be updated incrementally. We illustrate the efficacy of I-SIM through a novel recommender (SWIFT) as well as through a trust-distrust predictor in Online Social Networks (I-TRUST). We experimentally show that I-SIM enables fast and accurate predictions in an energy-efficient manner.

I. INTRODUCTION

The growth in the market for intelligent terminals like smart phones or tablets is enabling more and more users to access online services like e-commerce and social networks. The users of these online services generate a huge volume of data: we talk about a few quintillion (10^{18}) bytes per day. There is an evident need for online services like *personalization schemes*, namely *recommenders* as well as *trust-distrust* predictors. Recommenders assist users in various web activities [1]. Trust predictors enable users to select whom and what to trust while navigating through the web [2]. Recommenders typically rely on Collaborative Filtering (CF) techniques [1], [3] to suggest *relevant* items to users such as the recommendation of photo groups on Flickr, books on Amazon, and videos on Youtube. At the heart of many practical CF techniques [4] lies the computation of *similarities* between users, also known as *like-mindedness*. Similarly, nearest neighbor graphs, used for trust-distrust prediction in Online Social Networks (OSNs), also leverage similarities between the nodes [2].

A. Motivation

The starting point of this paper is the observation that existing similarity metrics were not designed to handle a very large number of users with rapidly changing behavior. The number of recommendation requests issued by users today, is in the order of millions per day [4], which poses a major scalability challenge. State-of-the-art scalable recommenders [5]–[7] employ batch processing and update their recommenders at intervals of weeks. They indeed achieve low latency recommendations, but ignore the temporal behavior of users (*temporal relevance* [8], [9]), thereby leading to relatively lower recommendation accuracy. For example, the number of views of news articles saturates within a few hours [10]: these articles should be recommended within this time span to be relevant. On the other hand, the very few recommenders that account for temporal relevance [4], [8] do not scale as they require heavyweight computations, inducing high energy consumption which is becoming a key issue in cloud computing [11].



(a) Moving global average rating where each point averages the 100,000 previous ratings.

(b) Total number of users.

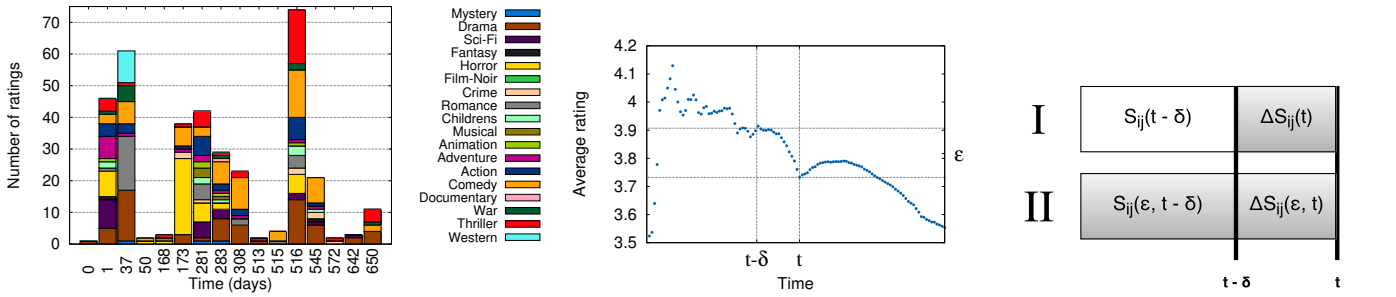
Fig. 1: Temporal effects in ML-1M dataset.

An interesting temporal effect that emerges from the MovieLens (ML) dataset [12] is depicted in Figure 1. Users typically provide their preferences for items in terms of *feedback* like *ratings*. Figure 1(a) conveys the fact that the moving global average rating fluctuates within the first 200 days. This fluctuation can be attributed to the initial user churn (as shown in Figure 1(b)). However, when the number of users is stable, we observe a downward trend in the average rating which saturates at around 3.5. The primary reasons behind this temporal behavior can be attributed to the users' *preference* and *behavioral drifts*.

Preference drift. Users' *preferences* typically fluctuate over time. For example, a change in the family structure can drastically change shopping patterns. Figure 2(a) depicts the preference distribution of an individual user over time. The top genre preferences for this user on Day-1 were Adventure, Horror and Sci-Fi whereas on Day-37 her preferences were mostly Western, Romance and Drama. We also observe other genre preferences that vary over the following days (e.g. Thriller).

Behavioral drift. At another personalization level, a user's feedback (e.g. scores, ratings, votes) also fluctuates over time possibly due to her varying behavior (e.g. mood). This feedback fluctuation results in a *user bias*. Given that a user u provides a feedback s_{ui} for an item i at a time $t - \delta$ when her average feedback was $\bar{s_u}(t - \delta)$, then the user is biased towards this item by $b_{ui}(t - \delta) = s_{ui} - \bar{s_u}(t - \delta)$. Sarwar et al. empirically showed that including such a user bias in the similarity computations, however in a static (non-temporal) manner, leads to better recommendation quality [13]. The change in this user bias ($b_{ui}(t - \delta) - b_{ui}(t)$) over time is the change in the average feedback ($\bar{s_u}(t) - \bar{s_u}(t - \delta)$).

Figure 2(b) captures the change in the user bias (behavioral



(a) Preference drift of a user in *ML-1M*.

(b) Behavioral drift of a user in *ML-1M*.

(c) Dependence on temporal parameter ϵ.

Fig. 2: Limitations of state-of-the-art similarity metrics with respect to temporal relevance and incremental updates. The gray areas in the right subfigure indicate the similarities (S_{ij}) that need to be updated within a time interval $[t-\delta, t]$.

drift) which we quantify using a key user attribute (ϵ) defined as follows: *the average feedback of a user varies over time in steps of a temporal parameter ϵ between a time interval* $[t-\delta, t]$. State-of-the-art incremental similarity metrics [4], [9] do not take into account this attribute (Figure 2(c)I). Performing incremental updates based on the temporal parameter ϵ is non-trivial. Similarities until time $t-\delta$ are also a function of ϵ and thus also need to be adjusted at time t (Figure 2(c)II).

Based on these observations, one can easily infer that users' temporal behavior can impact the prediction accuracy significantly. However, designing an incremental similarity metric that captures this temporal behavior is non-trivial.

B. Contributions

The main contribution of this paper is a novel similarity metric, we call I-SIM, which enables lightweight similarity computations incorporating the preference and behavioral drifts. I-SIM can be considered as a "temporalization" of the adjusted cosine similarity [13] and hence of the cosine similarity. Therefore, I-SIM can be easily integrated with time-aware applications in OSNs. In this paper, we primarily focus on collaborative filtering due to space limitations but nonetheless we also explore trust predictions in OSNs.

I-SIM is *lightweight* in the sense that it can be updated incrementally to achieve low latency and limited energy consumption. In particular, I-SIM accounts for temporal relevance through an exponential decrease in the weight of previous feedback over time. We formally prove that the time complexity[1] of I-SIM is $\mathcal{O}(|\Delta U|)$ where ΔU is the set of active users within a given time interval (unlike the time complexity of non-incremental metrics [13] which is $\mathcal{O}(|U|)$ where U is the set of total users in the system).

First, we illustrate the power of I-SIM in personalization applications by implementing a novel recommender leveraging I-SIM, which we call SWIFT (**S**calable **I**ncremental **F**lexible **T**emporal recommender). SWIFT is interesting in its own right, as it enables flexible switching between *stream processing* and *batch processing* [14]. We demonstrate the efficiency of I-SIM through an in-depth experimental evaluation of SWIFT. More precisely, we compare SWIFT with recommenders using incremental similarity computations (TENCENTREC [4]), matrix factorization techniques using temporal relevance (TIMESVD [8]), Alternating Least Squares (ALS [15]) and factored similarity models (FISM [16]), on real-world traces in terms of latency, energy consumption, and accuracy.

[1]If not stated otherwise, we refer to the worst-case complexity.

Second, after demonstrating that trust relations in OSNs exhibit temporal behavior, we illustrate the power of I-SIM for trust predictions by implementing I-TRUST. We empirically show that I-TRUST significantly outperforms the non-incremental alternative, both in terms of runtime and accuracy.

C. Roadmap

The rest of the paper is structured as follows. We recall some preliminary concepts in § II. We introduce and analyze I-SIM in § III. We present two applications of I-SIM (SWIFT and I-TRUST) in § IV. We evaluate and demonstrate the effectiveness of I-SIM on real-world traces in § V and then review the related work in § VI. Finally, we conclude our paper in § VII. We present the detailed proofs of the theorems in our companion technical report [17]. Our code is available[2].

II. PRELIMINARIES

A. Temporal Relevance

Temporal relevance [8], [9] is a popular notion in data mining, commonly known as *concept drift*, a dynamic learning problem over time. A typical example is the change in user's interests when following an online news stream. In such domains (e.g. news, deals), the target concept (user's interests) depends on some temporal context (e.g. mood, financial state). This constantly changing context can induce changes in the target concepts, producing a concept drift. We now provide the definition of temporal relevance at any given timestep as follows where *timestep* is a logical time corresponding to the current number of incremental updates.

Definition 1 (Temporal Relevance): Temporal relevance measures the relevance of a feedback s_{ui} for making predictions at a timestep t based on a time-decaying parameter α. In the following, we denote the temporal relevance of s_{ui} at a timestep t by $f_{ui}^{\alpha}(t)$ and assign a weight to s_{ui} depending on the interval since the timestep (t_{ui}) when the actual feedback was provided.

$$f_{ui}^{\alpha}(t) = e^{-\alpha(t-t_{ui})} \tag{1}$$

Temporal relevance can be incrementally updated as follows: $f_{ui}^{\alpha}(t+1) = e^{-\alpha} f_{ui}^{\alpha}(t)$. This update relation is crucial for designing our novel similarity metric (I-SIM) as we demonstrate in § III.

We consider one decay factor (Equation 1). However, multiple weighting factors like temporal regression [18] based ones could also be considered in which case the corresponding update relations should be re-formulated accordingly.

[2]https://github.com/gdamaskinos/isim

B. Collaborative Filtering

The goal of CF [1] is to suggest new items to users by predicting scores on a set of items based on their rating feedback, as well as the rating feedback of other users. Each rating reflects the user's *explicit* feedback for the corresponding item. Alternative user behavior such as clicking, tagging or liking, introduces a form of *implicit* feedback. In this paper, we focus on explicit feedback, however, mapping techniques like *pseudoratings* [19] could be leveraged to convert implicit feedback to explicit ratings.

CF recommenders compare items and users for generating predictions. One of the approaches to achieve such comparisons is the *neighborhood* method [20]. The goal is to find similar objects (users or items) by exploring the relationships between them. The techniques employed by recommenders to explore these relationships can be divided into two categories: *user-based* and *item-based*. A user-based technique predicts a target user's preference for an item by leveraging the rating information aggregated from similar users. An item-based technique applies the same approach, but utilizes similarities between items instead of users. In this paper, we focus on item-based CF as it is shown to perform better than the user-based one [13]. Nevertheless, I-SIM can also be adapted to user-based CF.

We now present the recommendation setting before introducing our novel metric in § III. We consider a database consisting of $\mathcal{N}$ ratings on a set of m items $\mathcal{I} = \{i_1, i_2, .., i_m\}$ by a set of n users $\mathcal{U} = \{u_1, u_2, ..., u_n\}$ over time. The ratings are sorted based on the time of the event. Each rating event is in the form of a tuple: $\langle u, i, r_{ui}, \tau_{ui} \rangle$ which reflects the fact that user u provided a rating r_{ui} for an item i at a timestamp τ_{ui}. Furthermore, $\mathcal{U}_i^t$ denotes the set of users who have rated i until timestep t.

A standard item-based CF scheme typically includes three phases as shown in Algorithm 1. We briefly describe each of these phases as follows.

Similarity computation phase. This phase concerns the similarity computation based on the observed ratings. We use the adjusted cosine similarity as it was empirically demonstrated to be superior to other metrics for item-based CF [13]. The deviation from the average rating effectively captures the user's rating behavior. Moreover, the ratings provided by users that generally give low (strict) or high (generous) ratings, have a uniform effect on the similarities. The time complexity of this step is $\mathcal{O}(m^2 n)$ as there are a total of m^2 possible item-item similarities and each similarity computation has a complexity of $\mathcal{O}(n)$.

Neighborhood computation phase. This phase deals with computing the most similar items corresponding to a given item, based on the computed similarities, and creating the item-item network. For each item i, the top-K items, i.e. with the K highest similarities, are selected as the neighbors. The parameter K denotes the *model size*. The time complexity of this step is $\mathcal{O}(m)$ as for each item the complexity for the neighborhood computation is $\mathcal{O}(1)$ with the use of a max-heap.

Prediction phase. In this phase, the prediction scores are computed for each item according to Equation 3. The time complexity of this step is $\mathcal{O}(m)$ with the use of the max-heap containing the predictions for each of the items. Note that subtracting a user's average rating $\bar{r_u}$ compensates for differences in her rating scale thus making predictions more accurate.

Algorithm 1 Standard Item-based CF

Require: $\mathcal{I}$: Item set; $\mathcal{U}$: User set; $\mathcal{U}_j$: Set of users who rated an item with item-id j; $\bar{r}_u$: Average rating for user u.
Ensure: R_A: Top-N recommendations for a user Alice (A)

Phase 1 - Similarity computation: GetSimilars(j,$\mathcal{I}$)

Ensure: s_j: Max-heap for item j with item-ids as keys and similarities as values.
1: **for** i in $\mathcal{I}$ **do**
2:
$$s_j[i] = \frac{\sum_{u \in \mathcal{U}_i \cap \mathcal{U}_j} (r_{ui} - \bar{r_u})(r_{uj} - \bar{r_u})}{\sqrt{\sum_{u \in \mathcal{U}_i} (r_{ui} - \bar{r_u})^2} \sqrt{\sum_{u \in \mathcal{U}_j} (r_{uj} - \bar{r_u})^2}} \quad (2)$$
3: **end for**
4: **return:** s_j

Phase 2 - Neighborhood computation: KNN(j,$\mathcal{I}$)

Ensure: N_j: K most similar items to item j.
5: $N_j = nlargest(K, GetSimilars(j, \mathcal{I}))$
6: **return:** N_j

Phase 3 - Prediction: Top-N($\mathcal{I}$)

Require: S_{ij}: similarity between two items i, j.
Ensure: R_A: Top-N recommendations for Alice.
7: var Pred ▷ Max-heap with predictions for Alice
8: **for** $i : item$ in $\mathcal{I}$ **do**
9:
$$Pred[i] = \bar{r_u} + \frac{\sum_{j \in KNN(i,\mathcal{I})} S_{ij}(r_{uj} - \bar{r_u})}{\sum_{j \in KNN(i,\mathcal{I})} |S_{ij}|} \quad (3)$$
10: **end for**
11: $R_A = nlargest(N, Pred)$
12: **return:** R_A

These three steps do not consider temporal information. Designing an incremental update for the adjusted cosine metric is not trivial due to the time-varying user bias (Figure 2(b)). Including temporal relevance in this metric makes the problem even more challenging.

C. Trust-distrust Relationship in Online Social Networks.

Trust-distrust relations between users play a vital role in making decisions in OSNs like voting for administrators. In practice, the available explicit trust relations are often extremely sparse, therefore making the prediction task more challenging. Weighted nearest neighbor algorithms are widely used for predicting trust relations [2], [21]. Algorithm 2 demonstrates one such algorithm leveraging K-nearest neighbors (KNN) to predict trust relations.

We denote the trust level of user w for a user v as R_{wv}. Given n classes with labels C_0, C_1,..., C_n which reflect the different levels of trust/distrust [22] between two users, we define a mapping function ϕ such that $\phi(R_{wv}) = C_i$ and $0 \leq i \leq n$. We then define $Score(w, v, C_i)$ as follows.

$$Score(w, v, C_i) = \begin{cases} 1 & \phi(R_{wv}) = C_i \\ 0 & \phi(R_{wv}) \neq C_i \end{cases}$$

Since trust relation between users is asymmetric, it is possible to have $Score(w, v, C_i) \neq Score(v, w, C_i)$ when $R_{wv} \neq R_{vw}$.

Algorithm 2 Trust Prediction

Require: $\mathcal{U}$: User set; $\mathcal{U}_w$: Set of users who trusted/distrusted another user with user-id w.
Ensure: R_{wv}: Trust level of user w for a user v.

Phase 1 - Similarity computation: GetSimilars(v, $\mathcal{U}$)

Ensure: s_v: Max-heap for user v with user-ids as keys and similarities as values.
1: **for** w in $\mathcal{U}$ **do**
2:
$$s_v[w] = \frac{\sum_{u \in \mathcal{U}_w \cap \mathcal{U}_v} R_{wu} R_{vu}}{\sqrt{\sum_{u \in \mathcal{U}_w} R_{wu}^2} \sqrt{\sum_{u \in \mathcal{U}_v} R_{vu}^2}} \quad (4)$$
3: **end for**
4: **return:** s_v

Phase 2 - Neighborhood computation: KNN(v, $\mathcal{U}$)

Ensure: N_v: K most similar users to user v.
5: $N_v = nlargest(K, GetSimilars(v, \mathcal{U}))$
6: **return:** N_v

Phase 3 - Prediction: PredictTrust(w, v)

Ensure: Trust prediction of user w for a user v.
7: **return:** $\underset{C \in \{C_0, \ldots, C_n\}}{\text{argmax}} \sum_{l \in KNN(w, \mathcal{U})} Score(l, v, C)$

These three phases resemble the ones in Algorithm 1. The first phase (similarity computation) employs the standard cosine similarity between users. The second phase is similar to the one in Algorithm 1 and derives the KNN set for a given user. Finally, the last phase predicts the trust relation between two users based on the KNN graph constructed in the previous two phases.

III. I-SIM: A NOVEL SIMILARITY

In this section, we first pose the similarity computation problem more formally and then present our I-SIM similarity metric before analyzing it. We then show how I-SIM enables incremental updates (for item-item similarities) over time.

A. Problem Definition

Let $\mathcal{U}$ be a set of users, $\mathcal{I}$ be a set of items, and $S_{ij}(t)$ be the similarity between items $i, j \in \mathcal{I}$ till timestep t. We define the similarity function as follows.

$$S_{ij}(t) = \frac{P_{ij}(t)}{\sqrt[n]{Q_i(t)} \cdot \sqrt[n]{Q_j(t)}} \quad (5)$$

where n is a positive integer, P is a function of the item vectors i, j, and Q is a function of each individual item vector. For example, if we take the standard cosine similarity (Equation 4), then n is 2, P is the dot product of item vectors i and j whereas Q is the squared L^2-norm of each individual item vector. Note that the similarity function definition is formulated for the similarity metrics designed for sparse data (e.g. cosine, jaccard, pearson correlation). For sparse data, which often contains asymmetric data, similarity depends more on attributes that are shared, rather than attributes that are lacking.

For an incremental similarity computation, each of these terms (P, Q) could be incrementally updated as follows.

$$P_{ij}(t) = \Delta P_{ij}(t) + P_{ij}(t-1)$$
$$Q_i(t) = \Delta Q_i(t) + Q_i(t-1)$$

This incremental update seems straightforward when each of the P and Q functions could be expressed as a summation term independent of any time-varying parameter (Figure 2(c)I). Nevertheless, for more precise similarity metrics, like adjusted cosine similarity, each timestep depends on some time-varying parameter like the average rating of users. Therefore, the P and Q values, computed in all previous $t-1$ timesteps, need to be updated (Figure 2(c)II).

In this paper, we solve this non-trivial problem by essentially caching some additional terms. We break the update computation into two components: *standard* (P^s, Q^s) and *adjustment* (P^a, Q^a) components as follows.

$$P_{ij}(t) = \underbrace{P^s_{ij}(t)}_{\text{standard component}} + \underbrace{P^a_{ij}(t)}_{\text{adjustment component}}$$
$$Q_i(t) = \underbrace{Q^s_i(t)}_{\text{standard component}} + \underbrace{Q^a_i(t)}_{\text{adjustment component}}$$

More precisely, the standard component incorporates the preference drift (Figure 2(a)) whereas the adjustment component incorporates the behavioral drift (Figure 2(b)).

B. I-SIM

We now describe our I-SIM metric which temporalizes adjusted cosine similarity (Equation 2). Given m items and n users, the overall time complexity of the similarity update for standard techniques (Algorithm 1) is $\mathcal{O}(m^2 n)$ per timestep. Naively augmenting the standard adjusted cosine with temporal relevance would require computing item-item similarities at each batch update leveraging all the ratings (Figure 2(c)II). The resulting time complexity ($\mathcal{O}(m^2 n)$ per batch update) would be prohibitive for an online recommender.

We first rewrite the adjusted cosine similarity (Equation 2), incorporating temporal relevance (Equation 1), in terms of *pre-normalized correlation* (P_{ij}) and *normalization factors* (Q_i, Q_j) following the pattern presented in Equation 5.

$$S_{ij}(t) = \frac{P_{ij}(t)}{\sqrt{Q_i(t)} \sqrt{Q_j(t)}} \quad (6)$$

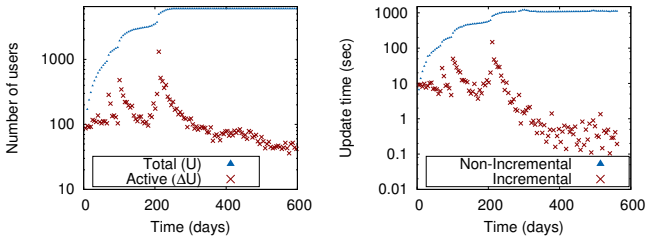
where

$$P_{ij}(t) = \sum_{u \in U^t_i \cap U^t_j} f^{\alpha}_{ui}(t)(r_{ui} - \bar{r_u}(t)) f^{\alpha}_{uj}(t)(r_{uj} - \bar{r_u}(t)) \quad (7)$$

$$Q_i(t) = \sum_{u \in U^t_i} (f^{\alpha}_{ui}(t)(r_{ui} - \bar{r_u}(t)))^2 \quad (8)$$

Next, we show that the functions $P_{ij}(t)$ and $Q_i(t)$ can be incrementally updated with a time complexity $\mathcal{O}(|\Delta \mathcal{U}|)$. Thus $S_{ij}(t)$ can also be incrementally computed on-the-fly. Additionally, this incremental feature reduces the time complexity drastically, enabling lightweight model updates with incoming streams of data. The *active users* at any given time interval are the users who provide ratings in that interval. Figure 3(a) compares the total number of users ($|U|$) at any given time with the number of active users ($|\Delta U|$) during the last 5 days. Figure 3(b) indicates that the computation time required for the similarity update of our incremental approach on a single machine is a few orders of magnitude lower than a non-incremental one. We also observe that the computation time for the incremental approach (Figure 3(b)) corresponds to the number of active users (Figure 3(a)) at any given time.

Before providing the incremental update relations, we introduce two adjustment terms (L, M). These adjustment terms incorporate the behavioral drift captured by $\epsilon(t)$.



(a) Total users vs active users. (b) Similarity computation time.

Fig. 3: Comparison between incremental (I-SIM) and non-incremental similarity computations [13], [23] for ML-1M dataset. The time interval for the active users is 5 days.

$$L_{ij}(t) = \sum_{u \in U_{ij}^t} \epsilon(t) f_{ui}^{\alpha}(t) f_{uj}^{\alpha}(t)[(r_{ui} - \bar{r_u}(t)) + (r_{uj} - \bar{r_u}(t))],$$

$$L_i(t) = 2 \sum_{u \in U_i^t} \epsilon(t) f_{ui}^{2\alpha}(t)(r_{ui} - \bar{r_u}(t)) \tag{9}$$

$$M_{ij}(t) = \sum_{u \in U_{ij}^t} \epsilon(t)^2 \cdot f_{ui}^{\alpha}(t) f_{uj}^{\alpha}(t),\ M_i(t) = \sum_{u \in U_i^t} \epsilon(t)^2 \cdot f_{ui}^{2\alpha}(t) \tag{10}$$

where $\epsilon(t) \triangleq \bar{r_u}(t) - \bar{r_u}(t-1)$.

Theorem 1 (P_{ij} incremental update): Let $\Delta\mathcal{U}_i^t$ denote the set of users who newly rated i at timestep t, i.e. $\Delta\mathcal{U}_i^t = \mathcal{U}_i^t \setminus \mathcal{U}_i^{t-1}$, then the time complexity for updating $P_{ij}(t)$ is $\mathcal{O}(|\Delta\mathcal{U}_i^t| + |\Delta\mathcal{U}_j^t|)$.

Sketch: The incremental update relation of P_{ij} is:

$$P_{ij}(t) = \Delta P_{ij}(t) + e^{-2\alpha}[P_{ij}(t-1) - L_{ij}(t-1) + M_{ij}(t-1)]$$

where $\Delta P_{ij}(t)$ is defined as follows.

$$\begin{aligned}\Delta P_{ij}(t) = &\sum_{u \in \Delta\mathcal{U}_i^t \cap \mathcal{U}_j^{t-1}} (r_{ui} - \bar{r_u}(t)) f_{uj}^{\alpha}(t)(r_{uj} - \bar{r_u}(t)) \\ &+ \sum_{u \in \mathcal{U}_i^{t-1} \cap \Delta\mathcal{U}_j^t} f_{ui}^{\alpha}(t)(r_{ui} - \bar{r_u}(t))(r_{uj} - \bar{r_u}(t)) \\ &+ \sum_{u \in \Delta\mathcal{U}_i^t \cap \Delta\mathcal{U}_j^t} (r_{ui} - \bar{r_u}(t))(r_{uj} - \bar{r_u}(t))\end{aligned}$$

The summation terms in $\Delta P_{ij}(t)$ have a time complexity of $\mathcal{O}(|\Delta\mathcal{U}_i^t| + |\Delta\mathcal{U}_j^t|)$.

■

Note that if $P_{ij}(t)$ was updated non-incrementally then the time complexity would be $\mathcal{O}(|\mathcal{U}_i^t \cap \mathcal{U}_j^t|)$. With each time step, the number of new ratings for i ($|\Delta\mathcal{U}_i^t|$) tends to be significantly smaller than the total number of ratings for i ($|\mathcal{U}_i^t|$). The difference is huge even for the average case as $|\mathcal{U}_i^t|$ can be of the order of all users in the system (Figure 3). For example, following the long tail distribution (Figure 13(a)) the popular items (20% of all the items) would be rated by nearly 80% of the users in the system.

Theorem 2 (Q_i incremental update): Given that $\Delta\mathcal{U}_i^t$ denotes the set of users who newly rated i at timestep t, i.e. $\Delta\mathcal{U}_i^t = \mathcal{U}_i^t \setminus \mathcal{U}_i^{t-1}$, then the time complexity for updating $Q_i(t)$ is $\mathcal{O}(|\Delta\mathcal{U}_i^t|)$.

Sketch: The incremental update relation of Q_i is:

$$Q_i(t) = \Delta Q_i(t) + e^{-2\alpha}[Q_i(t-1) - L_i(t-1) + M_i(t-1)]$$

where $\Delta Q_i(t)$ is defined as follows.

$$\Delta Q_i(t) = \sum_{u \in \Delta U_i^t} (r_{ui} - \bar{r_u}(t))^2$$

The incremental term ($\Delta Q_i(t)$) has a time complexity of $\mathcal{O}(|\Delta\mathcal{U}_i^t|)$. Note that the complexity for the non-incremental update is again $\mathcal{O}(|\mathcal{U}_i^t|)$. ■

Hence, the final incremental relations for the adjusted cosine similarity are as follows.

$$P_{ij}(t) = \underbrace{\Delta P_{ij}(t) + e^{-2\alpha} P_{ij}(t-1)}_{\text{standard component}} - \underbrace{e^{-2\alpha}[L_{ij}(t-1) - M_{ij}(t-1)]}_{\text{adjustment component}} \tag{11}$$

$$Q_i(t) = \underbrace{\Delta Q_i(t) + e^{-2\alpha} Q_i(t-1)}_{\text{standard component}} - \underbrace{e^{-2\alpha}[L_i(t-1) - M_i(t-1)]}_{\text{adjustment component}} \tag{12}$$

$$L_{ij}(t) = \Delta L_{ij}(t) + e^{-2\alpha}[L_{ij}(t-1) - 2M_{ij}(t-1)] \tag{13}$$

$$M_{ij}(t) = \Delta M_{ij}(t) + e^{-2\alpha} M_{ij}(t-1) \tag{14}$$

The I-SIM values (S_{ij}) can thus be computed on-the-fly, leveraging the incrementally updated $P_{ij}(t)$ and $Q_i(t)$ values. We only need to store the P, L, M and Q values which requires $\mathcal{O}(m^2)$ space. Unlike classical non-incremental algorithms [13], we require extra storage for the adjustment terms (L, M). The non-incremental algorithms [13], [23] also require $\mathcal{O}(m^2)$ space for storing the item-item similarities. Nonetheless, incremental as well as non-incremental algorithms could benefit from sparse data structures as well as *count sketches* [24] for significantly reducing the storage requirements.

We now provide a variant of I-SIM we call I-SIM$_{\epsilon=0}$ which temporalizes pure cosine similarity. Adjusted cosine similarity leads to a pure cosine one if the average rating ($\bar{r_u}$) is set to 0 in Equation 2. More precisely, a lack of behavioral drift leads to L_{ij} and M_{ij} being 0 in equations 11 and 12 due to $\epsilon(t)$ being 0. The final incremental relations for pure cosine similarity are as follows and do not require any additional storage due to the absence of adjustment terms.

$$P_{ij}(t) = \Delta P_{ij}(t) + e^{-2\alpha} P_{ij}(t-1) \tag{15}$$

$$Q_i(t) = \Delta Q_i(t) + e^{-2\alpha} Q_i(t-1) \tag{16}$$

I-SIM also applies to the case of static neighborhood based algorithms (i.e. without using temporal relevance by setting α to 0 in the update equations). Such algorithms are often utilized during the cold-start phase of a system.

IV. I-SIM APPLICATIONS

A. SWIFT: A Novel Recommender

To illustrate the efficiency of I-SIM, we plug it in a novel recommender we design and implement, called SWIFT (**S**calable **I**ncremental **F**lexible **T**emporal recommender). In the following, we present SWIFT and highlight some optimization techniques that speed up its computations, as we later demonstrate through our evaluations.

Framework. We choose Apache Spark[3] as our cloud computing framework. As we pointed out, practical recommenders today need to deal with millions of recommendation requests per day, leading to billions of computations. Additionally, the recommendation service must be fault-tolerant. Spark performs fast in-memory computations by efficiently distributing tasks

[3] http://spark.apache.org/

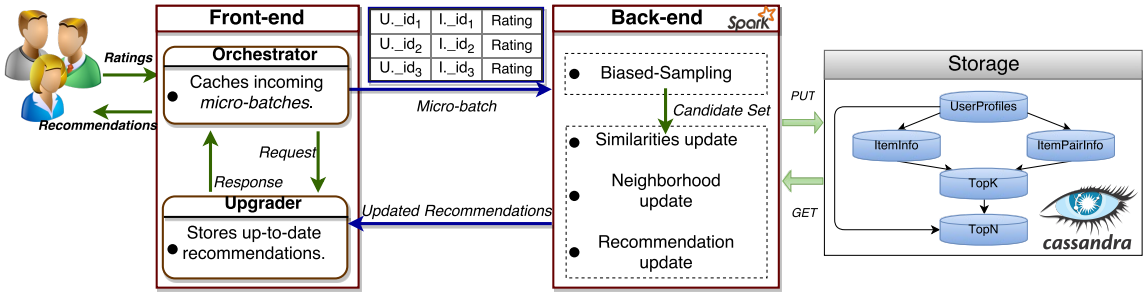


Fig. 4: The architecture overview of SWIFT. (The arrows in the back-end storage denote the dependencies for updating the information stored in Cassandra tables).

to a set of computation units commonly known as *Executors*. Data is stored in *Resilient Distributed Datasets* (RDDs) which provide the required level of fault-tolerance for our recommender. Entertainment services like Netflix use Spark for real-time stream processing in the context of online recommendations and data monitoring. We choose Apache Cassandra[4] for our permanent storage requirements as it enables fast interactions on big chunks of data. The architecture of SWIFT consists of a front-end and back-end as illustrated in Figure 4.

Front-end. The front-end accumulates new ratings in micro-batches and then leverages the available information to provide recommendations to users with low latency. The front-end consists of two sub-components implementing the recommendation functionality efficiently. These sub-components perform their tasks independently and hence can execute in parallel. In detail, these sub-components' tasks are as follows.

- *Orchestrator.* This sub-component receives the new micro-batch of rating events along with the recommendation requests. The new ratings are cached temporarily on the front-end server. Next, for each recommendation request, the orchestrator sends a response containing the recommendations to the corresponding client machine. The orchestrator then transmits the set of cached new ratings to the back-end server.
- *Upgrader.* This sub-component is responsible for receiving the set of updated recommendations from the back-end server and updating the set of locally-stored recommendations, to be used by the *orchestrator*.

The front-end provides fast recommendations, with lower time complexity, based on possibly stale similarity values (i.e. from previous updates). Nevertheless, the difference from the actual updated similarity values is negligible, as item similarities tend not to vary significantly in short time intervals [23], thus converging to the K-nearest neighbors. The recommendations are thus still accurate despite using moderately stale similarity values.

Back-end. SWIFT's back-end is built on Apache Spark and consists of a driver process deployed on a master node along with a set of executor processes deployed on worker nodes. The back-end computes the similarity updates for each micro-batch.

The back-end also hosts a Cassandra cluster for storing the aggregated information (Table I) as key-value tuples, utilized to compute the updates and the recommendations. A Spark-Cassandra connector API provides a mapping between RDDs and Cassandra tables thus allowing Spark to perform cassandra query operations on RDDs.

[4]http://cassandra.apache.org/

Table	Key	Value
UserProfiles	User_id	Rating feedback
ItemInfo	Item_id$_i$	L_i, M_i, Q_i
ItemPairInfo	(Item_id$_i$, Item_id$_j$)	L_{ij}, M_{ij}, P_{ij}
TopK	Item_id	K-nearest neighbors
TopN	User_id	Top-N recommendations

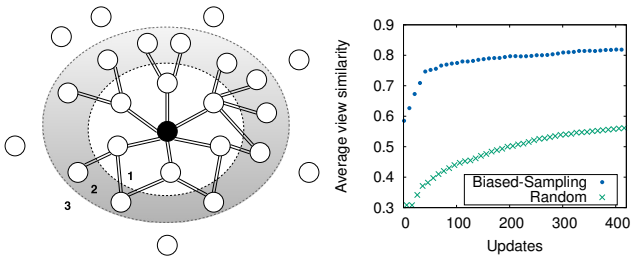
TABLE I: Back-end storage on Cassandra.

The back-end performs two majors tasks: *sampling* and *update* as shown in Figure 4. The information that SWIFT utilizes to perform the recommendations is distributed in five tables, represented as blue cylinders in Figure 4. The stored information is incrementally updated in five phases corresponding to each of the described tables. Finally, the back-end sends the updated recommendations derived from the *TopN* table to the front-end upgrader.

A key advantage of this front-end, back-end design is parallelism, separating the two different functionalities of SWIFT, namely *recommendation request handling* (front-end) and *incremental update* (back-end). The information between the front-end and back-end is transferred via the network in a compressed gzip format in order to avoid an additional energy overhead.

This design also provides *flexibility* to our system as the size of the micro-batch can be tuned. The service provider that hosts SWIFT can choose the frequency of the updates depending on the available resources. A small start-up company using SWIFT can aim for a medium-sized micro-batch (say around 100 events per micro-batch) to trade the additional costly updates for relatively less accurate similarity values. By setting a micro-batch size value of 1, SWIFT performs stream processing (similar to TENCENTREC [4]). The micro-batch size can also be automatically set by the front-end based on the rate of incoming events as well as the estimated latency of the back-end such that bigger mini-batches can be used at peak usage times. Additionally, the front-end can temporarily increase the mini-batch size to allow for some back-end maintenance. The ability to trade between stream and micro-batch processing of new ratings, depending on the users' demands, highlights the flexibility of our approach.

Biased sampling. Calculating all the similarity pairs for every new update would lead to a prohibitive $\mathcal{O}(|\mathcal{I}|^2 * |\Delta\mathcal{U}|)$ time complexity for each update where $\mathcal{I}$ denotes the set of all items and $\Delta\mathcal{U}$ denotes the set of users who provided new ratings. In the average case, a small fraction of the total similarity pairs is significantly affected after an update. Therefore, updating the similarities only for the aforementioned small fraction of item pairs and using stale values for the rest would notably reduce time complexity without compromising the recommendation accuracy. A sampling method is required for carefully selecting the item pairs to be updated, balancing the trade-off between



(a) Candidate set for an item (in black). (b) Convergence for *ML-1M*.

Fig. 5: The biased sampling technique of SWIFT.

the number of updates and the recommendation accuracy.

We apply an incremental biased sampling technique to address this issue. Our sampling technique is applied in an item-based manner as item-item similarities are more stable than user-user similarities [25]. This biased sampling technique is illustrated in Figure 5(a). The black item i is the most recently rated item. Region 1 contains the K-nearest neighbors of i which we will reference to as one-hop neighbors ($knn_i^{(1)}$). Region 2 contains K^2 two-hop neighbors of i ($knn_i^{(2)}$). Finally, region 3 contains K random items ($Rand(K)$), thus creating the *candidate set*[5] of maximum size: $1+K+K*K+K = (K+1)^2$ items. The random neighbors are required in order to update the similarities for some items that are not in the two-hop neighborhood. Therefore, the function for selecting the K-nearest neighbors is not stuck at a local minimum. This technique results in a convergence to neighbors of *good* quality[6] within a few updates and eventually converges to the optimal top-K (Figure 5(b)).

Theorem 3 (Biased sampling): The incremental biased sampling eventually converges to the optimal top-K neighbors.

Sketch: First, we mathematically denote the candidate set at timestep t: $cand_i(t) = \{knn_i^{(1)}(t-1) \cup knn_i^{(2)}(t-1) \cup Rand(k)\}$. Our biased sampling technique results in a directed graph $G_{KNN}(t)$ that connects each item with a set of items $knn_i^{(1)}(t)$ that maximizes the similarity function $S_{ij}(t)$:

$$knn_i^{(1)}(t) = \max_{j \in cand_i(t)} \sum_{j=1}^{K} S_{ij}(t)$$

After T iterations, the scanned items consist of $\bigcup_{t=1}^{T} cand_i(t)$. Moreover, we have $\bigcup_{t=1}^{T} cand_i(t) \xrightarrow{T \to \infty} \mathcal{I}$ where $\mathcal{I}$ is the set of all items. Hence, our biased sampling technique eventually converges to the optimal top-K neighbors. ■

Figure 5(b) depicts the fast convergence of our biased sampling as compared to a random sampling technique where the candidate set does not include the two-hop neighbors ($cand_i(t) = \{knn_i^{(1)}(t-1) \cup Rand(k)\}$). The *view similarity* denotes the average similarity of the top-K neighbors at any given update step.

SWIFT's sampling technique improves the incremental update time complexity to $\mathcal{O}((K+1)^2 * |\Delta\mathcal{U}|) = \mathcal{O}(|\Delta\mathcal{U}|)$. Note that there are other sampling techniques used to speedup K-nearest neighbor computation like the one in TENCENTREC with $\mathcal{O}(|\mathcal{I}| * |\Delta\mathcal{U}|)$ time complexity for each incremental update which makes our sampling technique significantly faster.

Recommendation. We implement item-based CF (Algorithm 1), introduced in § II-B, by executing the following phases in SWIFT.

- We substitute the *similarity computation phase* by leveraging our novel I-SIM metric.
- The *neighborhood computation phase* leverages the candidate set selected using our *item-based biased sampling* technique to reduce the time complexity of the K-nearest neighbor search. More precisely, we replace the item set I with the candidate set in the *GetSimilars* function within Phase 2 of Algorithm 1.
- For the *prediction phase*, we apply the prediction score function, shown in Equation 3, to generate the final predictions. We reduce the computations by predicting only for the top 10% of the items sorted by popularity. We then compute the top-N recommendations by sorting the prediction scores.

One general problem for a recommender is the cold-start, when recommendations are required for *new items* (i.e. items with no previous ratings in the database). In SWIFT, we initially assign the K most popular items as neighbors for the new item. Neighbors converge to the K-nearest ones after a few iterations for this item as we demonstrate in Figure 5(b).

B. I-TRUST: *Trust-distrust Predictor in OSNs*

To demonstrate the efficiency of I-SIM in trust-distrust predictions, we plug I-SIM$_{\epsilon=0}$ in a trust-distrust prediction application which we call I-TRUST.

Temporal behavior also exists in trust-distrust relationship in OSNs. For example, the trust between an elector and voters might change over time. One such behavior is demonstrated in the *Wiki-Elections* trace [26]. We observe a decreasing trend in the number of votes on Wiki-Elections as shown in Figure 6. More intuitively, this shows that during the first election, the voters' trust for this wikipedia administrator decreases with time due to more negative votes (distrust).

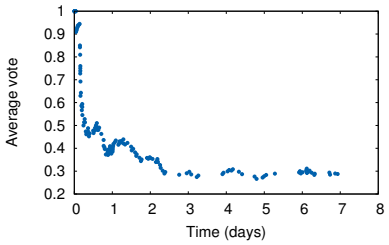


Fig. 6: Voters' trust in an administrator during a Wiki-Election

We design a trust predictor which captures these temporal effects. We employ Algorithm 2 for two classes (C_0: Trust, C_1: Distrust) to predict the trust relationships. We plug I-SIM$_{\epsilon=0}$ in the similarity computation phase. Based on equations 15 and 16, we update the similarity computations incrementally after some given number of events during which $\mathcal{O}(|\Delta U|)$ users were active. The time complexity of each update step then decreases from $\mathcal{O}(|U|)$ to $\mathcal{O}(|\Delta U|)$ as shown in § III. As we demonstrate later in our experimental evaluation, I-TRUST's incrementality improves the latency significantly whereas its temporality improves the prediction accuracy.

[5]The candidate set consists of all the items for which the information (i.e. P, Q, L, M) is incrementally updated by SWIFT's back-end.

[6]Good quality neighbors are the neighbors with relatively high similarity.

V. Experimental evaluation

In this section we report on the performance of our two applications (SWIFT and I-TRUST) in terms of accuracy, latency and energy consumption. Then, we compare them with state-of-the-art alternatives on real-world traces.

A. Experimental Setup

We first describe our experimental environment along with our methodology for obtaining the results.

Platform. We select the *Grid5000* testbed[7] as our experimental platform. Each cluster on Grid5000 has a set of nodes with specific resources. We measure the energy consumption of our implementations using Grid5000's customized Wattmeter which monitors the power consumption.

Unless stated otherwise, we deploy our implementations on a Spark cluster consisting of four nodes. Each node consists of two six-core Intel Xeon E5-2630 v3 CPUs, 128 GB of memory along with 600 GB disk storage. We tune our Spark cluster optimally in order to achieve the best possible performance in terms of the number of partitions and executors per node. We empirically found that the optimal performance, in terms of latency, is obtained by using one executor per machine and setting the number of partitions for all RDDs approximately equal to the total number of physical cores in the Spark cluster.

Datasets. We use publicly available real-world datasets. More specifically, we use *MovieLens* datasets [12]: ML-1M and ML-20M. The ML-1M dataset consists of 1,000,209 ratings from 6040 users on 4000 movies. The ML-20M dataset consists of 20,000,263 ratings from 138,493 users on 27,278 movies. *Rating density* denotes the fraction of actual ratings collected among all possible ratings. To evaluate the effect of increasing the rating density, we use a densified[8] Flixster dataset by employing the method introduced in [9] which leads to 5,105,850 ratings from 10,000 most active users on 4000 most popular movies. Finally, for evaluating I-TRUST we employ the *Wiki-Elections* dataset [26] containing 114,029 votes from 6210 users on 2391 editors.

Metrics. We evaluate both our applications from various aspects. We describe below the metrics used in our evaluation.

Click-Through-Rate (CTR). We adopt this metric to test the accuracy of the recommendations. Given that $\mathcal{H}_u$ is the set of recommended items that were clicked by a user u (hits), and $\mathcal{R}_u$ is the set of items recommended to u, we denote the CTR for u by CTR_u and define it as follows: $CTR_u = |\mathcal{H}_u|/|\mathcal{R}_u|$

The overall CTR over the whole test set is the average over the CTR values for all users in the test set. Note that a recommended item is considered as a *hit*, if the user rates that item anytime later than the time of the recommendation. Ideally, CTR for e-commerce services varies between 1%-5% depending on the type of service [27].

Recall. We use this metric to capture the sensitivity of a recommender to the frequency of updates. Given that $\mathcal{C}_u$ is the set of items clicked by a user u, we denote the recall for u by $Recall_u$ and define it as follows: $Recall_u = |\mathcal{H}_u|/|\mathcal{C}_u|$. The overall recall is the average over the recall values for all the users in the test set.

Classification accuracy. We use this metric to test the accuracy of trust-distrust predictions in OSNs. More precisely, the classification accuracy is the fraction of correct predictions among all the predictions.

Mean Absolute Error (MAE). We employ this metric to ensure a fair comparison with model-based alternatives which optimize for low prediction error. The MAE is defined as follows: $MAE = \sum_{u,i \in S} |\hat{R}_{ui} - R_{ui}|/|S|$, where $\hat{R}_{ui}$ denotes the rating prediction for user u and item i, R_{ui} denotes the actual rating and S denotes the set of test rating events. Since MAE captures how close the predictions are to the actual ratings, the lower the error, the higher the model prediction accuracy.

Latency. This metric quantifies the delay observed to complete a single task. This delay consists of three main parts: CPU time, I/O time, and communication delay (e.g. if data is scattered on multiple nodes). For a set of tasks, we show the minimum, median and 99^{th}-*percentile* latency[9].

Energy-per-click. This metric quantifies the amount of energy required for performing computations for a single user click. This metric intuitively evaluates the *impact of a single click* on the consumed energy. More precisely, we measure the aggregated energy consumption of the entire cluster, on which we deploy our experiments, for the operations that a single recommendation task (click) triggers. Given that $\bar{P}$ denotes the average cluster power consumption throughout the computation time of a click (denoted as t), the energy consumption is computed as follows: $E = \bar{P} * t$. We measure the energy-per-click in terms of watt-hour (Wh).

Evaluation scheme. The datasets include the timestamp for each event. We replay the dataset, ordered by the timestamp, to capture the same temporal behavior as the original one. Furthermore, we split the dataset into *training*, *validation* and *test* sets. Based on the benchmark for evaluating stream-based recommenders [28], our test set consists of the most recent 1000 ratings. The validation set consists of the last 1000 ratings from the training set and is used for parameter tuning. For the non-incremental competitors we train the model on the training set until it converges and then we evaluate the trained model on the test set.

B. SWIFT Evaluation

SWIFT is designed to provide accurate recommendations with low latency in an energy-efficient manner. In this section, we evaluate SWIFT's performance for varying parameter settings and then compare it with state-of-the-art incremental and non-incremental competitors.

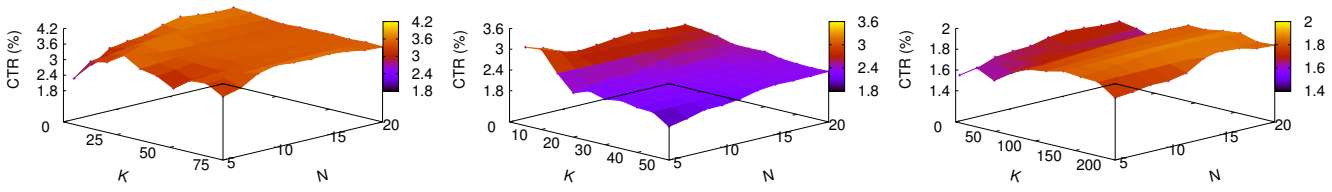
To compare with incremental recommenders, we consider TENCENTREC's practical item-based CF (which we refer to as TENCENTREC). Compared to SWIFT, TENCENTREC's practical algorithm employs incremental approximate cosine similarity (instead of I-SIM) with *real-time pruning* (instead of biased sampling) and *real-time personalized filtering* while predicting only for the top 10% of the items sorted by popularity similar to SWIFT (Phase 3 in Algorithm 1).

For the non-incremental alternatives, we compare with a standard matrix factorization based recommender using temporal relevance (TIMESVD [8]) as well as with the factored similarity models (FISM [16]), both of which are publicly

[7] https://www.grid5000.fr/

[8] The density for ML20M is 0.0053, for ML1M 0.045, and for Flixster 0.128.

[9] The latency observed by 99% of the tasks is below this value.



(a) ML-1M (b) ML-20M (c) Flixster

Fig. 7: Impact of model size (K) and recommendations-per-click (N) on accuracy.

available in the LIBREC[10] library for recommenders. Additionally, we compare with the distributed alternating least squares (ALS) algorithm available in Spark's MLlib.

We train SWIFT using the training set and then provide recommendations for each rating event in the test set. More precisely, for the training set, SWIFT computes the required information (P, Q, L, M) based on the equations 7 to 10 of the adjusted-cosine similarity (Equation 6). For the test set, SWIFT updates this information using equations 11 to 14 and then provides recommendations using the updated information. Depending on the flexibility mode, the back-end is invoked for the update operations either per click (stream processing) or per micro-batch (batch processing). In the stream processing mode, the front-end responds to the clients' requests only after receiving the updated recommendations from the back-end.

Accuracy. The following experiments demonstrate the effect of SWIFT's parameters on the recommendation accuracy, namely: *model size* (K), *recommendations-per-click* (N), *micro-batch size* (L) and *temporal relevance* (α).

Model size. We measure the CTR while varying the model size (K) which is the number of neighbors in the item-item network. We observe in Figure 7 that after a certain model size any further increase in the model size reduces the CTR. This decrease in CTR is due to the inclusion of less similar neighbors in the neighborhood of an item. These less similar neighbors add noise to the predictions.

Recommendations-per-click. The number of recommendations provided per click, is another important parameter that affects the CTR as too few will be insufficient whereas too many will reduce the interest of users in the recommendations. Hence, it is important to highlight that in practical recommenders, the recommendations-per-click (N) should not exceed 20. For example, IMDB uses Top-12 list to suggest movies and Last.fm uses Top-5 list to suggest songs. We observe a steady behavior in CTR with increasing N as shown in Figure 7. This behavior can be attributed to the fact that the size of the recommendation hits grows proportionally to the size of the recommended items.

Micro-batch size. Recall that SWIFT provides a flexible back-end as mentioned in § IV-A. More precisely, SWIFT provides recommendations treating each stream of rating events as a micro-batch. Hence, SWIFT can provide stream processing with the micro-batch size set to 1 whereas the micro-batch size can be set to few hundreds of rating events for batch processing. Note that this flexibility is an important feature for practical recommenders, as depending on the available resources (due to limited operational costs) or the network traffic (due to multiple recommendation requests), the micro-batch size can be adjusted by the service provider hosting SWIFT.

We now evaluate the impact of the flexibility mode on accuracy. Practically, many recommenders like Amazon or eBay repeat certain recommendations similar to SWIFT. Such repeated recommendations are less frequent in the stream processing mode (more frequent updates in top-N recommendations) but occur more often as the micro-batch size increases. Therefore, the denominator of the CTR (number of recommended items) decreases as the micro-batch size increases. On the contrary, the denominator of the recall (number of clicked items) is independent of the micro-batch size. More updated recommendations (smaller micro-batch size) lead to more hits and thus result in an increase in the numerator. Hence, we employ the recall to capture the difference in accuracy for varying micro-batch sizes.[11]

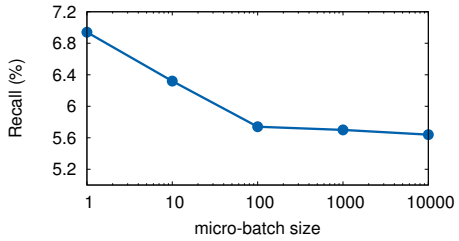


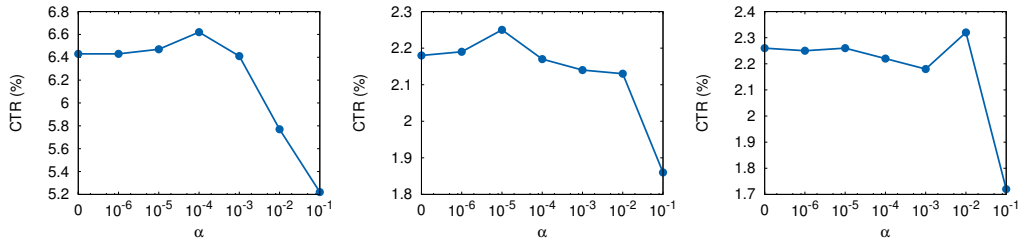
Fig. 8: Impact of flexibility mode on accuracy for ML-1M.

More precisely, Figure 8 illustrates this trade-off between accuracy and micro-batch size. Compared to the stream processing mode (micro-batch size set to 1), there is an impact on the recommendation accuracy, in terms of recall, for the batch processing mode. Furthermore, there is a steep decrease in the recall with increasing micro-batch size. This behavior is due to less frequent updates leading to more temporally stale similarities.

Temporal relevance. We analyze the effect of temporal relevance on the quality of recommendations in terms of CTR. For these experiments, we increase the test set to the last 10,000 events as the drift in the users' interests is more evident over longer test periods. We set the micro-batch size to 100 and tune the degree of temporal relevance by regulating the temporal weight parameter α. We observe an improvement in the CTR while increasing the value of α as shown in Figure 9. Moreover, we also observe that the CTR starts decreasing at some point. This outcome occurs due to the fact that many of the users rated very few items and our item-based approach leverages the items in the profile of the user. Hence, an increased value of α results in degrading the already few ratings in the user profile leading to a cold-start scenario for the given user. Note that we can also vary α specifically for each user profile; this is left for future work.

[10] http://www.librec.net/

[11] Note that all the experiments leveraging the CTR metric have a fixed micro-batch size.



(a) ML-1M (b) ML-20M (c) Flixster

Fig. 9: Impact of temporal relevance (α) on accuracy. Setting α to 0 deactivates SWIFT's temporal feature.

Table II compares SWIFT with incremental recommenders (TENCENTREC) as well as with non-incremental ones (TIMESVD, ALS, FISM) in terms of mean absolute error in predictions. We observe that SWIFT outperforms the others on the more sparse datasets (ML-1M, ML-20M) whereas ALS performs best on a relatively dense dataset (Flixster).

Approach \ Dataset	*ML-1M*	*ML-20M*	*Flixster*
FISM	0.731	0.873	0.713
TIMESVD	0.806	0.892	0.73
ALS	0.707	0.746	**0.629**
SWIFT	**0.686**	**0.662**	0.669
TENCENTREC	0.784	0.721	0.684

TABLE II: Model comparison (MAE) between incremental and non-incremental alternatives.

Latency. SWIFT's latency is primarily affected by the *model size* (K), *micro-batch size* (L) and *cluster size* parameters. We now provide the results concerning SWIFT's latency for different settings for these parameters.

Model size. SWIFT's biased sampling depends on the model size (K). An increase in the model size generates larger candidate sets ($\mathcal{O}(K^2)$ size) thereby leading to more computations. Figure 10 depicts that the increase in the computations is more evident for large and sparse datasets like ML-20M. This behavior is due to the fact that the larger amount of items in the database combined with the sparsity leads to more diverse items in a candidate set. Hence, the amortized complexity of our biased sampling increases. In this specific case, the biased sampling does not reduce the computations with large values of K, thereby having a significant impact on latency (as shown in Figure 10 for ML-20M and $K = 200$).

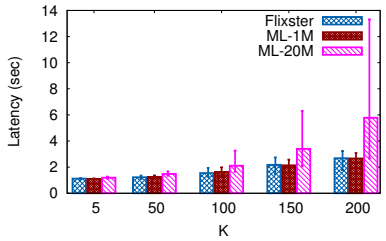


Fig. 10: Impact of model size (K) on latency (stream processing).

Micro-batch size. We evaluate the flexibility of SWIFT by varying the micro-batch size. Figure 11 shows the recommendation and update latency of SWIFT's front-end and back-end respectively for $K = 50$. The update latency is increasing with the micro-batch size as the information for more items' candidate sets needs to be updated. Nevertheless the recommendation time is nearly the same for varying micro-batch size. The latency observed between a click and the generation of the recommendations is a few milliseconds. Note that in the batch processing mode, the similarities are updated only after the system receives a micro-batch of L fresh ratings.

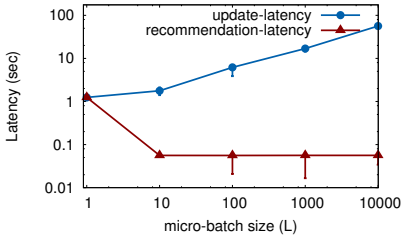


Fig. 11: Impact of batch processing on latency for ML-1M.

Cluster size. We deploy SWIFT and ALS on the same cluster while increasing the cluster size (number of nodes in the cluster) and compare the improvement in terms of median latency (which we quantify as speedup). Figure 12 demonstrates that SWIFT (stream processing mode with the model size set to 200) achieves a better speedup than ALS. Furthermore, an increase in the micro-batch size leads to an increase in the speedup for SWIFT. Therefore, the increase in the update latency, shown in Figure 11, can be mitigated by employing more nodes due to SWIFT's scalability.

The scalability saturates after a certain cluster size (5 nodes) due to the communication time with Cassandra as well as the sequential dependencies among SWIFT's tasks. The communication overhead with Cassandra could be possibly mitigated by using a distributed Cassandra cluster and tuning it to maximize the benefits from locality whereas the sequential dependencies could be reduced by pipelining the tasks to exploit more parallelism. It is important to note that the observed bottleneck is implementation specific and not a limitation of I-SIM.

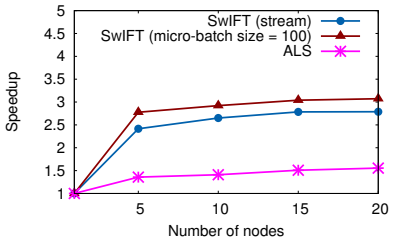
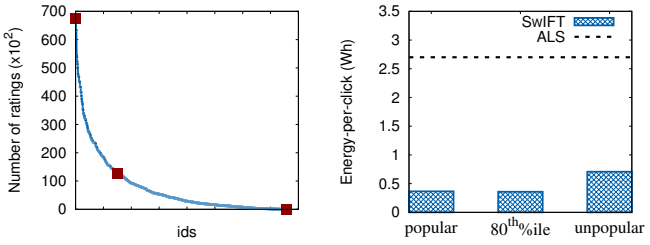


Fig. 12: Scalability comparison for ML-20M.

Energy Consumption. We evaluate the energy consumed by the computations induced due to a user click. In other words, we estimate the impact of a single click on energy consumption. Recall that our goal is to reduce the energy consumption by reducing the time complexity. We analyze the energy consumption corresponding to the clicks for three representative items: *most popular*, *least popular* and 80^{th}

percentile[12]. The ratings provided by users follow a long tail distribution (Figure 13(a)) where 80% of the users rate only 20% of the items. Hence, we choose our 80^{th} percentile item along with the most popular and unpopular items as shown in Figure 13(a).

Figure 13(b) depicts the energy consumption of SWIFT ($K = 100$) for clicks corresponding to these three items. The unpopular items are not strongly correlated to their neighbors due to the relatively small number of ratings provided for each of them. Therefore, the items in their candidate sets have less overlap compared to those in the candidate sets of the more popular items. Thus, there is an increase in the computation time for the unpopular items leading to an increase in the energy-per-click. We deploy ALS on the same Spark cluster for benchmarking the energy consumption of a single update on this cluster (Figure 13(b)). Note that ALS is non-incremental and therefore requires significantly more time for one update than SWIFT, thus leading to higher energy consumption.



(a) Item Popularity (b) Energy-per-click

Fig. 13: Impact of item popularity on energy consumption for ML-20M.

C. I-TRUST *Evaluation*

We now evaluate the effectiveness of I-TRUST in providing accurate predictions with low latency. We denote the classical predictor implementing Algorithm 2 as C-TRUST. For the experiments, we set the model size (K) to 150 for C-TRUST to achieve the optimal quality. We have the same model size with the temporal parameter (α) as 0.3 for I-TRUST. We deploy these experiments on a single node. While training I-TRUST, we update the similarities incrementally after a fixed micro-batch of training events whereas for C-TRUST the similarities are computed using all the training events in a non-incremental manner.

Runtime. We measure the total runtime for updating the similarities needed for constructing the K-nearest neighbor graph using all the training events. This graph is then used to predict the trust relations as shown in Algorithm 2 (Phase 3). For I-TRUST, we set the micro-batch update for similarity computations to 1000 voting events. From Table III, we observe that the runtime improves by 36 times.

Accuracy. Table III confirms I-TRUST's superiority in terms of accuracy. I-SIM$_{\epsilon=0}$ incorporates the time-varying trust relations between an administrator and the voters, in the similarity values. Therefore, the k-nearest neighbor graph is temporally more accurate and leads to better predictions. The improvement is reflected in the difference with C-TRUST for the voting classification task.

[12]The 80^{th} percentile popular item is the one with popularity higher than 80% of the items.

Approach	Runtime	Classification Accuracy
C-TRUST	421.2 s	79.21%
I-TRUST	**11.66 s**	**80.75%**

TABLE III: Runtime and accuracy comparisons for I-TRUST and C-TRUST.

VI. RELATED WORK

Collaborative filtering. CF algorithms can be generally divided into two categories: *memory-based* and *model-based*. Memory-based algorithms employ user-item ratings to compute the predictions and then generate relevant recommendations. These algorithms can be either *user-based* [20] or *item-based* [13]. Our work focuses on the item-based CF technique which has been shown to provide more accurate recommendations compared to the user-based one [13]. In contrast to memory-based techniques, model-based ones build parametric models by learning iteratively on the training datasets and then leverage the learned model to generate predictions. Different types of models are typically used, including matrix factorization [8] and factored item similarity models [16]. Standard model-based techniques require to update their learned models by employing all the ratings, including the new ones, and hence are not incremental in nature.

Real-time recommenders. These have recently attracted a lot of attention. Huang et al. presented TENCENTREC, a real-time stream recommender [4] which uses an incremental version of approximate cosine similarity. We demonstrate in § V that by trading storage (to store the L and M information), I-SIM performs better in terms of accuracy compared to the similarity metric leveraged by TENCENTREC. Furthermore, SWIFT's biased sampling is significantly faster than TENCENTREC's real-time pruning as we explained in Section IV. Whilst Yang et al. [29] presented a scalable item-based CF method by using incremental update, they did not however address the problem of temporal relevance.

Temporal relevance. Few approaches have addressed the problem of temporal relevance in the context of CF. One simple heuristic to capture the temporal behavior of a user, applicable to any recommender, is to consider only the most recent ratings in her profile for generating the recommendations [4], [30], [31]. In our work, we focus on the temporal relevance in the context of similarity computations. Ding et al. [32] exploited the timestamps of ratings to adapt the item-based CF technique. They incorporated time-based weights in the score prediction stage but did not adapt the similarity computations, hence leading to higher time complexity. Lathia et al. [33] analyzed the effect of temporal relevance by varying the neighborhood size over time. Koren et al. [8] designed a matrix factorization model that considers the temporal behavior of users. However, their model has a higher time complexity as they employ multiple time dependent parameters. Liu et al. [9] introduced an incremental version of cosine similarity that provides temporal relevance. However, Sarwar et al. [13] empirically showed that an item-based CF technique provides more accurate recommendations by leveraging the adjusted cosine metric (compared to the classical cosine one). I-SIM provides incremental updates for the adjusted cosine similarity while incorporating the temporal relevance feature.

Energy-efficiency. Despite a large amount of work on large-scale CF [13], [34], [35], none of the existing approaches focuses on reducing the time complexity. The main focus

has been so far to design distributed algorithms which can decentralize the computations over multiple nodes leading to better scalability. This strategy leads to more resource utilization and thereby higher energy requirements. However, energy consumption is currently a major concern in data centers [36]. Energy costs are quickly rising in large-scale data centers and are soon projected to overtake the cost of hardware. Energy-efficiency is the new holy grail of data management systems research [37]. We address this energy-efficiency issue by designing incremental computations with lower time complexity.

Trust-distrust in OSNs. Trust inference algorithms rely on users' feedback to predict future trust relations. However, trust relations are assumed to be static in existing literature [2], [21]. In this paper, we first demonstrate that trust relations can be time-varying and then present how to capture these dynamic trust relations by leveraging I-SIM and thus enabling lightweight incremental similarity updates.

VII. CONCLUDING REMARKS

We present I-SIM, a novel similarity metric that enables similarity computations in an incremental and temporal manner. We illustrate through two applications the effectiveness of I-SIM in practice: (a) SWIFT incorporating I-SIM for recommendation and (b) I-TRUST incorporating I-SIM$_{\epsilon=0}$ for trust prediction. We empirically show that I-SIM leads to better accuracy and lower latency along with energy efficiency compared to state-of-the-art alternatives. Moreover, I-SIM can be leveraged to incorporate time-awareness in similarity-based applications, for example, trust recommendation in mobile ad-hoc networks [21] or predictive blacklisting against malicious traffic on the Internet [38].

Acknowledgement: This work was funded by the Web-Alter-Egos Google Focused Award and the ERC grant for Adversary-Oriented Computing under grant agreement number 339539.

REFERENCES

[1] X. Su and T. M. Khoshgoftaar, "A survey of collaborative filtering techniques," *AAI*, vol. 2009, p. 4, 2009.

[2] Q. Zhao, W. Zuo, Z. Tian, X. Wang, and Y. Wang, "Predicting trust relationships in social networks based on wknn," *Journal of Software*, 2015.

[3] A. Boutet, D. Frey, R. Guerraoui, A.-M. Kermarrec, and R. Patra, "Hyrec: leveraging browsers for scalable recommenders," in *Middleware*, 2014, pp. 862–873.

[4] Y. Huang, B. Cui, W. Zhang, J. Jiang, and Y. Xu, "Tencentrec: Real-time stream recommendation in practice," in *SIGMOD*, 2015, pp. 227–238.

[5] J. J. Levandoski, M. D. Ekstrand, M. J. Ludwig, A. Eldawy, M. F. Mokbel, and J. T. Riedl, "Recbench: benchmarks for evaluating performance of recommender system architectures," *VLDB*, vol. 4, no. 11, 2011.

[6] B. M. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Recommender systems for large-scale e-commerce: Scalable neighborhood formation using clustering," in *ICCIT*, 2002.

[7] R. Bambini, P. Cremonesi, and R. Turrin, "A recommender system for an iptv service provider: a real large-scale production environment," in *Recommender systems handbook*, 2011, pp. 299–331.

[8] Y. Koren, "Collaborative filtering with temporal dynamics," *CACM*, vol. 53, no. 4, pp. 89–97, 2010.

[9] N. N. Liu, M. Zhao, E. Xiang, and Q. Yang, "Online evolutionary collaborative filtering," in *RecSys*, 2010, pp. 95–102.

[10] K. Lerman and T. Hogg, "Using a model of social dynamics to predict popularity of news," in *WWW*, 2010, pp. 621–630.

[11] U. Awada, K. Li, and Y. Shen, "Energy consumption in cloud computing data centers," *International Journal of Cloud Computing and services science*, vol. 3, no. 3, p. 145, 2014.

[12] "*MovieLens*," http://grouplens.org/datasets/movielens/.

[13] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *WWW*, 2001, pp. 285–295.

[14] S. Shahrivari, "Beyond batch processing: towards real-time and streaming big data," *Computers*, vol. 3, no. 4, pp. 117–129, 2014.

[15] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 8, no. 42, pp. 30–37, 2009.

[16] S. Kabbur, X. Ning, and G. Karypis, "Fism: factored item similarity models for top-n recommender systems," in *SIGKDD*. ACM, 2013, pp. 659–667.

[17] "*Technical Report*," https://www.dropbox.com/s/861zhoihb366l1j/ISim_TechReport.pdf?dl=0.

[18] A. Brenner, B. Pradel, N. Usunier, and P. Gallinari, "Predicting most rated items in weekly recommendation with temporal regression," in *Proceedings of the Workshop on Context-Aware Movie Recommendation*. ACM, 2010, pp. 24–27.

[19] T. Q. Lee, Y. Park, and Y.-T. Park, "A time-based approach to effective recommender systems using implicit feedback," *Expert systems with applications*, vol. 34, no. 4, pp. 3055–3062, 2008.

[20] J. Herlocker, J. A. Konstan, and J. Riedl, "An empirical analysis of design choices in neighborhood-based collaborative filtering algorithms," *Information retrieval*, vol. 5, no. 4, pp. 287–310, 2002.

[21] J. Luo, X. Liu, Y. Zhang, D. Ye, and Z. Xu, "Fuzzy trust recommendation based on collaborative filtering for mobile ad-hoc networks." in *LCN*. Citeseer, 2008, pp. 305–311.

[22] J. Golbeck, B. Parsia, and J. Hendler, "Trust networks on the semantic web," in *International Workshop on Cooperative Information Agents*. Springer, 2003, pp. 238–249.

[23] K. Ali and W. Van Stam, "Tivo: making show recommendations using a distributed collaborative filtering architecture," in *SIGKDD*, 2004, pp. 394–401.

[24] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in data streams," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1530–1541, 2008.

[25] D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich, *Recommender systems: an introduction*. Cambridge University Press, 2010.

[26] "*Wiki-Elections dataset*," https://snap.stanford.edu/data/wiki-Elec.html.

[27] B. Kille, A. Lommatzsch, R. Turrin, A. Serény, M. Larson, T. Brodt, J. Seiler, and F. Hopfgartner, "Stream-based recommendations: Online and offline evaluation as a service," in *Experimental IR Meets Multilinguality, Multimodality, and Interaction*, 2015, pp. 497–517.

[28] ——, "Overview of clef newsreel 2015: News recommendation evaluation lab," 2015.

[29] X. Yang, Z. Zhang, and K. Wang, "Scalable collaborative filtering using incremental update and local link prediction," in *CIKM*, 2012, pp. 2371–2374.

[30] P. G. Campos, A. Bellogín, F. Díez, and J. E. Chavarriaga, "Simple time-biased knn-based recommendations," in *Workshop on Context-Aware Movie Recommendation*. ACM, 2010, pp. 20–23.

[31] C. Chen, H. Yin, J. Yao, and B. Cui, "Terec: A temporal recommender system over tweet stream," in *VLDB*, 2013, pp. 1254–1257.

[32] Y. Ding and X. Li, "Time weight collaborative filtering," in *CIKM*, 2005, pp. 485–492.

[33] N. Lathia, S. Hailes, and L. Capra, "Temporal collaborative filtering with adaptive neighbourhoods," in *SIGIR*, 2009, pp. 796–797.

[34] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *AAIM*, 2008, pp. 337–348.

[35] K. Yu, S. Zhu, J. Lafferty, and Y. Gong, "Fast nonparametric matrix factorization for large-scale collaborative filtering," in *SIGIR*, 2009, pp. 211–218.

[36] J. Koomey, "Growth in data center electricity use 2005 to 2010," 2011.

[37] S. Harizopoulos, M. A. Shah, J. Meza, and P. Ranganathan, "Energy efficiency: The new holy grail of data management systems research," in *CIDR*, 2009.

[38] F. Soldo, A. Le, and A. Markopoulou, "Predictive blacklisting as an implicit recommendation system," in *INFOCOM*, 2010, pp. 1640–1648.