The Journal of Supercomputing (2022) 78:16394–16424
https://doi.org/10.1007/s11227-022-04484-6

An efficient system using implicit feedback and lifelong learning approach to improve recommendation

Gautam Pal[1]

Accepted: 24 March 2022 / Published online: 6 May 2022

Abstract

This paper presents a new technique for contextual item-to-item Collaborative Filtering-based Recommender System, an improved version popularised by e-commerce giant *Amazon* two decades back. The concept is based on items also-viewed under the same browsing session. Users' browsing patterns, locations, and timestamps are considered as the *context* and *latent factors* for each user. The algorithm computes recommendations based on users' *implicit* endorsements by *clicks*. The algorithm does not enforce the user to log in to provide recommendations and is capable of providing accurate recommendations for *non-logged-in users* and with a setting where the system is unaware of users' preferences and profile data (non-logged-in users). This research takes the cue from human *lifelong incremental learning* experience applied to machine learning on a large volume of the data pool. First, all historical data is gathered from collectable sources in a distributed manner through big data tools. Then, a long-running batch job creates the initial model and saves it to Hadoop Distributed File System (HDFS). An ever-running streaming job loads the model from HDFS and builds on top of it in an *incremental* fashion. At the architectural level, this resembles the big data mix processing *Lambda Architecture*. The recommendation is computed based on a proposed equation for a weighted sum between near real-time and historical batch data. Real-time and batch processing engines act as autonomous Multi-agent systems in collaboration. We propose an ensemble method for batch-stream the recommendation engine. We introduce a novel *Lifelong Learning Model* for recommendation through Multi-agent Lambda Architecture. The recommender system incrementally updates its model on streaming datasets to improve over time.

Keywords Implicit feedback-based recommender system · Lifelong learning · Item to item collaborative filtering · Apache Storm · Context-aware systems

Gautam Pal
gautam.pal1947@gmail.com

[1] Department of Psychological Medicine, King's College London, London SE5 8AB, United Kingdom

Abbreviations

LML	Lifelong machine learning
LA	Lambda architecture
MALA	Multi-agent lambda architecture
KM	Knowledge miner
KB	Knowledge base
IFBRS	Implicit feedback-based recommender system
HDFS	Hadoop distributed file system

1 Introduction

Recommender systems are an effective substitute to *search algorithms* as they help users discover items in e-commerce sites users might not have found by themselves. Recommender systems adopt three types of approaches: (a) Content-based, (b) Collaborative, (c) Hybrid. Content-based filtering algorithms are based on a description of the item and profile of the user's preferences [1]. The system creates a content-based profile of users based on a weighted vector of item features. These are often combined with the user's feedback to assign a higher or lower weight to order the *top n recommendation*. For example, when a user views a mobile phone in the shopping portal, a few other similar mobile phones are recommended for him. On the other hand, Collaborative Filtering methods [2–5] are based on collating and analysing a large amount of information on all users' behaviours, activities, or preferences and predicting what users will tend to like based on their similarity to other similar users or items. The underlying philosophy assumes that people who agreed in the past will agree in the future. The hybrid approach makes content-based and collaborative-based predictions separately and blends them with weightage.

Recommender systems get input from different sources to make recommendations. The most common way of collecting input is through the user's feedback. Shopping portals, for example, collect ratings provided by the users. Nevertheless, explicit feedback may not be always available. Thus, a number of attempts were taken to build recommendations through users' implicit feedback [6].

The recommender system in e-commerce provides a prominent way to enhance the overall shopping experience by providing personalised or contextual advice through mining and discovering the interests and analysing patterns of customers. The method has made e-commerce more personal. No longer are products marketed to a mass audience, but are personalised based on individual unique needs. It also has a significant role in generating revenue for the website by the fact that users tend to purchase if a recommended product is relevant to his need.

2 Related work

2.1 Background of implicit feedback-based recommender system

Users provide feedback both in terms of *explicit* ratings as well as implicit endorsements like views, purchases, comments, shares, and likes. *Implicit* feedback is more common compared to explicit *ratings*. Despite that, *existing* recommendation systems rely on explicit feedback making them prone to sparsity problems and cold start situations [6, 7]. In comparison, *this work* does not have access to *ratings*. Recommendations are computed purely based on users' click-to-view items.

Most of the *existing* recommendation systems do not consider *contextual* information extracted from the user browser like geographic location, date and time. However, our works show that *contextual* information tends to improve recommendation accuracy by making more *personalised* recommendations. For instance, in e-commerce portals, buying patterns significantly influence users' geographic location, such as a colder or a warmer place. In a way, the current implementations work with two types of parameters—users and items but do not combine them into a *context*. The *context* is a multifarious construct that is studied in varying specialisations like natural language processing, geospatial data analysis, computer vision, etc. [8–12]. In this paper, to consider *context information* is considered as a latent factor to providing closely relevant recommendations as expressed by the following user-item-context relationship: Users × Items × Context → Implicit Feedback (views)-Based Recommender System (IFBRS).

The existing *item-to-user* rating-based recommender system suffers from issues like user cold start, item cold start, sparsity, scalability, etc. [13–15]. In user cold start, ratings cannot be predicted for a new item until similar users have rated the same item. In the case of sparsity, very few items are rated by users leading to a sparse item-user matrix, and hence finding similar users is difficult. Scalability issue occurs as the dataset grows with time, filling the sparse matrix becomes gradually extremely compute-intensive, and each batch run takes a longer time [13–15].

The proposed Implicit Feedback-Based Recommender System (IFBRS) overcomes the above-said issues easily by adopting an item-to-item collaborative filtering approach. It does not attempt to find similar users based on ratings. The model does not have access to ratings provided by the users and does not access users' preferences captured during registration with the portal. Implicit feedback is purely computed in terms of the item viewed by users. The proposed ensemble model considers *context information* as a latent factor providing more meaningful recommendations depicted as follows: Users × Items × Context → Implicit Feedback (clicks)-Based Recommendations.

2.2 Lifelong incremental learning

Despite the recent advancement in machine learning, it is still in an era of *Weak AI* rather than *Strong AI*. Current machine learning algorithms only know how to solve a specific problem without the knowledge of reusing past learning to solve a group of *related* problems. Therefore, lifelong machine learning (or simply lifelong learning) [16] was introduced to solve the difficulties of knowledge accumulation and reuse with an infinite sequence of related tasks. For a group of related problems, an integrated model with knowledge reusing could decrease the cost of computing, such as *sample annotation* in the sentiment classification problem. In the *sentiment classification*, to predict the sentiment (positive or negative) of a sentence or a document, traditional approaches need to train an independent model on each domain to obtain the best performance. Each domain needs to collect labelled data for supervised learning. This approach is considered as *weak AI*.

The work presented in the paper addresses the problem using a Lifelong Machine Learning model that is analogous to several machine learning models such as lifelong learning, incremental learning, multi-task learning, transfer learning, and streaming learning. Thrun [17] first studied supervised lifelong learning through the decade of the 90s. The work explored information sharing across multiple collaborating tasks through neural network binary classification. A neural network approach to LML was introduced and subsequently improved by Silver et al. through the years 1996 to 2015 [18–23]. Cumulative learning is explored in the form of LML, which builds a classifier to classify all the previous and the new classes by updating the old classifier [24]. Ruvolo et al. [25] proposed an LML system based on the multi-task learning developed by Kumar et al. Efficient Lifelong Machine Learning (ELLA) [26] presented by Ruvolo and Eaton. Compared with multi-task learning [27], ELLA is much more efficient due to its unique proposition with real-time learning that gradually refines the learning outcome over time to maximise performance across all tasks. Zhiyuan, etc. [28] improved the sentiment classification by involving knowledge. The object function was modified with two penalty terms corresponding with previous tasks. [29]. Here, the learning tasks are autonomous and distributed. In the area of lifelong Unsupervised Learning, Zhiyuan et al. Wang [30] proposed various lifelong modelling techniques to generate topics from a set of historical tasks and use past knowledge to develop better topics. A notable application area like the item recommender system using LML emerged [31]. In the field of lifelong Semi-Supervised Learning, a Never-Ending Language Learner is proposed by Carlson et al. [32], and Mitchell et al. [33]. Using continuous web crawling, a large volume of information is gathered representing entities and relations in this approach. A testing scheme for the LML system is proposed by Lianghao et al. [34], where the incremental learning ability of LML is tested to verify if the system becomes gradually more knowledgeable over time through accumulation, and transfer of knowledge in each iteration. Leveraging the previous research in the area of LML, the proposed recommendation model uses an Incremental Lifelong Learning (ILML) approach through Storm Streaming which initialises itself with batch offset at the starting.

This paper leverages Spark Streaming APIs for testing and validation. Spark Streaming provides an abstraction on streaming datasets called *DStreams*, or discretised streams. *DStream* is a sequence of data arriving over time [35, 36]. Internally, each DStream is represented as a sequence of RDDs arriving at each time step (hence the name *discretised*). DStreams can be created from various input sources, such as Flume, Kafka, or HDFS. Once built, they offer two types of operations: transformations, which yield a new *DStream*, and output operations, which write data to an external system. *DStreams* provide many of the same operations available on RDDs and additionally provide new operations related to time, such as sliding windows [37]. Spark MLlib library includes APIs for sliding window-based clustering on streaming data.

Datastream clustering was widely researched and improved over the years. Among the early works in this area, Guha [38] proposed the *STREAM* algorithm, which produces a constant factor approximation for the k-Median problem in a single pass and using a small space. Gupta et al. [39] presented a study for outlier detection approaches and case studies for different forms of temporal data. A time decay function that puts variable weights decreasing over time while updating the model is suggested. The proposed model extends the *STREAM* algorithm with an enhancement that allows further merging of a large static historical data pool with the latest and most updated streaming model.

Various improvement techniques were proposed for dimension reduction in large-scale high-dimensional data. Agarwal [40] presented a dimension reduction process for the online learning component where only the limited parameters are learned online, and the remaining item features are learned through an offline batch process. The model is proved through a recommender system implementation for Yahoo! front page and *My Yahoo!*.

In summary, although there has been significant progress in machine learning algorithms and frameworks through the period of the last two decades, much less emphasis has been put on how these methods and algorithms can be used to train over an extended period of time to incrementally become more knowledgeable through knowledge retainment and transfer. This paper addressed these research gaps by developing an incremental, transfer learning model for recommendation engines through a big data stream-batch mix processing approach. The proposed batch-stream ensemble method is based on MALA, a novel collaborative mixed processing framework. MALA uses Hadoop MapReduce at the batch layer and Apache Storm at the stream layer to develop a *lifelong learning machine*. The method is effective in high dimensional big data applications through in-memory data processing capability, unique dimension reduction technique, and incremental never-ending learning approach. Also, the paper intends to fill the research gaps in real-time machine learning for so-called *continuous lifelong learning*, where both training of the model as well as predictions happen in real-time or near real-time. Lifelong Machine Learning can continually accumulate knowledge during the learning phase as a never-ending learning process.

As a precursor to the current work [41] first introduces the concept of MALA in the context of Lifelong Learning architecture. [42] further improves the MALA architecture through a novel scheme for reasoning and root cause analysis, streaming *hybrid* clustering, dimension reduction techniques, and in-memory processing capabilities.

3 Our contribution

The novel aspect of the proposed model is a contextual item-to-item Collaborative Filtering-based recommender system that does not rely on users' ratings or preferences. The recommendation accuracy of the model is higher than that of rating-based methods such as Spark ALS and other machine learning methods such as SVM, Logistic Regression, etc.

Another key contribution is a new incremental lifelong machine learning system for recommender systems by exploiting the interaction of low-latency stream processing with comprehensive batch processing.

The key contribution areas are as described below:

1. **Lifelong learning model.** We propose a novel collaborative ensemble method that uses Hadoop MapReduce at the batch layer and Apache Storm at the stream layer to create a *lifelong learning machine*.
2. **A new technique for implicit feedback-based recommender system as described below:**
   - We develop an Implicit Feedback-Based Recommender System (IFBRS) using a lifelong machine learning approach. The dataset used is taken from *Movielens*: (https://grouplens.org/datasets/movielens/100k/).
   - The work redefines the item-to-item relationship by assigning more relevance to the current trend than the historical data. A hybrid method combines the results from Collaborative Filtering with content-based filtering on item similarity.
   - As an implicit feedback (clicks)-based recommender system, it interprets item-to-item collaboration in one browsing session by each user, which is in contrast to the traditional user-to-user similarity approach.
   - To make recommendations relevant, IFBRS extracts the user's browsing location as a latent factor. Context-aware recommendations tend to be more accurate.
   - This case study redefines the item-to-item relationship by putting more relevance on the current trend than the historical data.
   - Finally, based on the above research, the work introduces a novel architecture of an end-to-end recommender system with a host of online and offline big data ecosystem tools and their correlation as a multimodal interactional model. The proposed architecture can be extended to other big data domains taking advantage of both stream and batch processing capabilities.

4 Lambda architecture

Lambda Architecture [43] is a simultaneous mix processing paradigmcts list at n that *aggregates low-latency* streaming frameworks with *high-throughput* Hadoop batch processing in a large distributed environment.

To build the *recently viewed* products list at a on links; we usnear equal pace, users continue to click on links, we use the *Apache Storm* stream processing pipeline [44–46] to cleanse and transform data to store into Cassandra DB. The recently viewed list is built ad hoc through a low-latency Cassandra query when a user continues to click the next items. The JSON response is returned to a UI through Java Spring REST APIs to display the *recently viewed* list [47–49]. At the same time, a simultaneous data pipeline stores the data into HDFS and periodic batch jobs through Apache Hive, perform data mining to extract user *motifs* through n-grams, and builds a recommendation service items ordering through collaborative filtering [50], which uses both batch and historical data. See Fig. 1.

To generalise, Lambda architecture provided a *simultaneous* and *mixed processing* environment for data ingested into the system and dispatched to both the batch layer and the stream layer. The stream layer serves only low-latency queries. Data is further merged for the type of queries that require comprehensive historical data.

5 Computing recommendations through collaborative filtering

The primary motivation behind item-based recommendation algorithms depends on the fact that the two items viewed by the same user in the same session are likely to be related to each other. Collective co-occurrence count sorted in descending order for each item reveals the trend of which items users are likely to search together

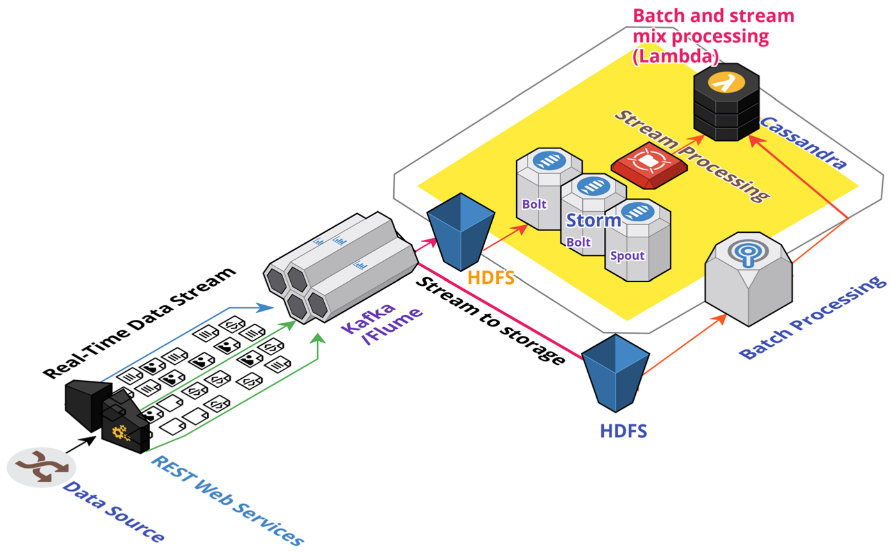


Fig. 1 Lambda Architecture is a simultaneous mix processing paradigm that parallel processes and *consolidates low-latency* streaming frameworks with *high-throughput* Hadoop batch processing in a large distributed environment. Observe that while data is being processed in real-time through Storm *Spouts and Bolts*, concurrent batch computing takes place by the stored data from HDFS. Cassandra stores the combined view from the batch and the stream

most. The associative or *collaborative filtering* rule is the central theme of the implicit feedback-based recommender. The idea of a collective count of items co-viewed under the same browsing session across all users' bases eliminates any reliance on user-provided ratings. User-provided ratings are sparse, but users' *clicks to view* data are always plentiful, and therefore the model is helpful in sparsity and cold start situations.

5.1 Application scenario

Consider the problem of building a personalised recommendation system for an e-commerce portal where the items listed are fairly time-sensitive, with the price and other parameters changing over time. The item parameters (e.g., price, category, etc.) are pre-stored in the database, and the user features are extracted through users' click data (for non-logged-in users) and profile data (for logged-in users). The overall objective is to maximise the number of clicks by showing the most relevant *personalised recommendations* to the users.

5.2 Model training

Let f_i be the feature vector of item i and f_u is the feature vector of user u. v_u denote the latent factor vector of user u and v_i is the latent factor vector of item i. When user u interacts with item i, CTR is predicted as $1/\left(1 + exp(-R_{ij})\right)$ by the Logistic model. Where the response score R_{ij} is given by

$$R_{ij} = f_i K f_u + v_i v_u \tag{1}$$

K is the regression coefficient matrix representing user-item interactions, extracted and learned from the input dataset. In implicit feedback-based systems, both latent factors v_i and v_u are frequently updated since parameters are derived from clickstream data.

5.2.1 Offline learning

Offline learner updates the model parameters and latent factors (K, f_i, f_u, v_i, v_u) from users clickstream data. The processing of a large data volume involves a parallel processing framework like Apache Spark. The offline process requires long-running jobs to complete, which executes every few hours interval. It ingests the trained models and parameters to the storage system using the Apache Flume [9, 11], from where the online learner incrementally updates the model.

5.2.2 Online learning

Online learners update the model parameters in real-time at a sliding window protocol to keep the model parameters in sync with the most recent changes. For known parameters, K, f_i, f_u, v_i, v_u; the online learning is about approximating a large number of independent regression problems for each individual item in i. The online regression process estimates v_i and v_u, while keeping f_iKf_u as an offset. The incremental online learning process is described in detail in Sect. 6.

5.3 Synchronising offline and online learning

The offline process is scheduled to initiate at an interval, 6 h in the current setup. When the offline learner finishes scheduled jobs, it stores the learned parameters K, f_i, f_u, v_i, v_u into a NoSQL storage. While the offline version provides a more comprehensive model training through the MapReduce framework, the online process keeps the model up to date with a near-real-time retraining process. The offline model is updated with newly learned model parameters in each iteration of a batch run, and the current iteration serves the end-users using the online model. The online process subsequently discards the old offline model and updates on top of the most recent version of the offline model.

5.4 Big data solution enabling rapid aggregation to build item co-occurrence matrix

The recommender system stores the stream data into the Cassandra DB. A co-occurrence job using Pig scripts or Java MapReduce is initiated at a periodic interval by an *Apache Ozzie* scheduler, generating the *items co-occurrence matrix*. Items for individual users are collated and subsequently merged with all other users' clicks to view data in the following sequence:

User views items A, B, C:

A–B (and B–A)
A–C (and C–A)
B–C (and C–B)

We can initialise the co-occurrence count to *one*. See Table 1.

Assume, users continue to view items *B, C, D* under the same browsing session. We can create or increment the association by count *one* and create the co-occurrence matrix for a single user:

B–C (and C–B)

Table 1 Co-occurrence table

Item1	Item2	Count
A	B	1
A	C	1
B	C	1
B	A	1
C	A	1
C	B	1

B–D (and D–B)
C–D (and D–C)

The count field is updated with each iteration. See Fig. 2.

The process is repeated across the all user base and is merged to create a single co-viewed matrix. Results of the co-viewed matrix may look like this: *iPhone, Galaxy phones were viewed together 100 times; iPhone, LG phones were viewed together 80 times in the same session by all users*. Note, items viewed under the same browsing session help to compute the related items searched by past users and, therefore, can be recommended to future users.

5.5 Computing recommended items

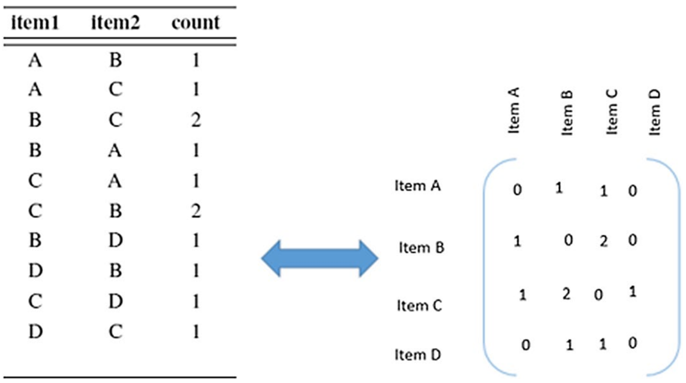
A weighted hybridisation strategy combines two or more factors by computing weighted sums of their individual recommendation scores.

Following are the different latent factors:

- Co-occurrence count
- User's location
- User preferences
- Timestamp of the click data in the recently viewed table.

Co-occurrence count is obtained in the previous step. The following section describes latent factors: location, user preferences, and timestamp. The overall recommendation score is computed for a user by Eq. (2):

$$R = \prod_{k=1}^{n} \omega_K \tag{2}$$



item1	item2	count
A	B	1
A	C	1
B	C	2
B	A	1
C	A	1
C	B	2
B	D	1
D	B	1
C	D	1
D	C	1

$$\begin{array}{c|cccc} & \text{Item A} & \text{Item B} & \text{Item C} & \text{Item D} \\ \hline \text{Item A} & 0 & 1 & 1 & 0 \\ \text{Item B} & 1 & 0 & 2 & 0 \\ \text{Item C} & 1 & 2 & 0 & 1 \\ \text{Item D} & 0 & 1 & 1 & 0 \end{array}$$

Fig. 2 The final Co-occurrence matrix based on items viewed together under the same browsing session across all users base

ω_K is the weight of each latent factor. Weight for co-occurrence count is normalised through the following equation:

$$\overline{\omega}_K = \frac{\omega_k}{max_k(\omega_k)} \tag{3}$$

where ω_k is the *view count* of the k$^{\text{th}}$ product and $max_k(\omega_k)$ is the maximum *view count* across all categories the user viewed. In the experimental setting, cumulative *view count* is considered for a period of *2 weeks*.

5.6 Users' location

An *active user* is the one for whom the recommendation is computed based on all passive users' collective historical records. First, the *active user's* physical location is retrieved from click data by parsing the *click to view* stream URL. Then, to compute the recommendation score, a higher weight is applied for the same location where an item is viewed, and a few other items are *also-viewed* by the *active user* under the same browsing session.

Suppose, if an active user is based at location l_1 and recommendations are shown for *viewed item* i_1, then we can compute *also-viewed* items for i_1 as i_2, i_3. If we consider i_1 and i_2 are viewed in the same location, we can assign more weight to the score of i_2. If i_1 and i_2 are viewed together n times and both are from the same location then the recommendation score for $i_2 = \mathrm{n} \times l_1$, where l_1 is the weight applied for a similar location. If i_1 and i_3 are from two different locations and the occurrence count between them is m then recommendation score is calculated for $i_3 = m \times l_2$, where l_2 is the weight applied for different location.

5.7 Users' preferences

The implicit feedback-based recommender is not reliant on the user's explicit preference settings that the user might have set while creating a profile with the portal. Rather, in an implicit model, preference is expressed as the user's *confidence* in a particular category of the product measured by repeated views by the same user. Suppose, a user views *smartwatches* multiple times, which is under the subcategory of the *wearable device* under category *electronics*. Then, recommending the user a few smartphones is a quite simple and effective strategy. We can apply greater weight to the same subcategory of recommended products in which the user shows more *confidence* by multiple views. View count is normalised by Eq. (3).

5.8 Weighted hybridisation strategy through time-variant data

In an e-commerce scenario *age of the data,* defined by the time order user viewed the items, can play an important parameter for the relevancy of recommended items [51]. *Data age* is therefore, an influencing factor ranking the *top n recommendations*. IFBRS prioritises recent data over old historical data. To achieve this, first, we compute the list of recently viewed items from Cassandra and order them reverse chronologically so that the most recent item is placed at the top of the list. Thereafter, a big data solution enables rapid aggregation to create a co-occurrence matrix for each recently viewed item, as discussed in the preceding section. Since recently viewed items are reverse chronologically ordered, when weight is applied in descending order, most recent items end up receiving higher weights. Real-time and historical batch processing components are two collaborative autonomous Multi-agent Systems. See Algorithm 1.

Algorithm 1 Implicit Feedback Based Recommender System

```
Input: User generated click stream event, time-decay-factor
Output: : Ordered list of recommendations with descending score
 1: item_decay_factor=y, item_timestamp_weight=x,
 2: for each user u who viewed item $I_i$ do
 3:    Compute a reverse chronological list $L_i$ of all the items recently-viewed
       $I_i$ (i=1 to n) in
 4:    for each $L_i$ do
 5:       Compute also-viewed item by user u as related item $RI_i$ under the same
          browsing session
 6:       for each item pair $I_i$ ,$RI_i$ do
 7:          $M_{ij} \leftarrow$ countCooccurrences($I_i, RI_i$) //$M_{ij}$ is the co-occurrence ma-
             trix between pair $I_i, RI_i$
 8:          $R \leftarrow top\_n\_Items(M_{ij})$
 9:          $R \leftarrow R \times \omega_l$ //applying location factor
10:          $R \leftarrow R \times \omega_p$ //applying user's preferences factor
11:          $R \leftarrow R \times \omega_{\overline{c}}$ //applying normalized co occurrence count factor
12:          $R \leftarrow R \times \omega_t$ //applying timestamp factor
13:          $\omega_t =$
             $\omega_t$ - $time\_decay\_factor$
14:       end for
15:    end for
16: end for
17: $R \leftarrow sort(R)$ //Final recommendation order of $I_i$ ,$RI_i$
18: return R //Ordered list of recommendations
```

5.9 Illustrative example

Assume, a user u_1 is in a location *loc-1*, opens an e-commerce portal and views 3 items: *A, B, C* at time t_1, t_2, t_3. Where $t_1 < t_2 < t_3$. Recommendations are built for the active user u_1.

First, we consider historical data pre-loaded into the database capturing cumulative click data from the entire user base. We can use the past dataset to create the *also-viewed* table, as shown in Fig. 2. The table contains items *also-viewed* across all user bases. A periodic batch job aggregates this data by adding the also-viewed count. For simplicity, we consider user preference count is *one* for all products; that is, the user has viewed items *A, B, and C* only once. See Table 2.

The recommendation score is computed based on Algorithm 1. A weighted hybridisation strategy computes the final score by applying different weightage schemes on location and time. If the item *viewed* and *items also-viewed* are from the same location, higher weight is applied to *also-viewed items*. Similarly, for the *timestamp* parameter, weights are adjusted in reverse chronological order. Thus, items with the latest timestamps are assigned to the highest weight. See Table 3.

Based on the recommendation score, the final order for the *top n recommendations* are:

D>E>H>F>G>I>J

5.10 A hybridisation strategy: computing recommendation through item similarity

The IFBRS overcomes user/item cold-start situations, sparsity, or scalability problems caused by *sparse rating data* available. However, where no historical *click view data* is available for a product (maybe a newly launched product), the user cannot see any recommendation since there are no similar items viewed together across all user bases. In such scenarios, the IFBRS can be extended to a *hybridisation strategy* to recommend similar items based on the item features. If the IFBRS does not return any item, the item similarity approach is pursued. In such scenarios, the IFBRS returns recommended items, then the items *not* similar to the viewed items are eliminated from final recommendations. Therefore, if a user views a smartphone, a few other *smartphones* appear to him as recommended items and not *shoes* or *watches,* which are not similar to a smartphone by category. Two methods can be adopted for item similarity: the first option is just to recommend the same category of products. That is, if a user views a *smartphone*, the recommendations are a few other *smartphones* ordered by popularity.

The second approach creates a user feature vector f_{it} and item feature vector f_{jt} at time *t*. The method needs to find the similarity between these vectors. There are several ways to measure the similarity between vectors. Starting with a simple setting where f_{it} and f_{jt} are data points in the same vector space; that is, both users and items are represented using the same set of features. We can measure the cosine similarity between two vectors. Nevertheless, the IFBRS does not consist of many users features since the system is designed for predicting

Table 2 Counting top *n* recommendations: Step 1

Timestamp	Item viewed	User preference count	Item also viewed	Similarity strength	Also viewed count
T1	A	1	D	0.98	10
T1	A	1	E	0.95	9
T1	B	1	F	0.88	11
T1	B	1	G	0.97	12
T2	B	1	H	0.77	15
T3	C	1	I	0.95	8
T3	C	1	J	0.94	6

recommendations with *non-logged-in* users. In reality, users expect to see recommendations in the same browser and device they use without the need to log in. Users tend to log in only during the checkout process. Thus, recommender systems can *not* have access to user features like gender, age, address, etc. Therefore, we can continue on the *method one,* which only builds upon the item features. The approach is computationally attractive, too, because of its simplicity and accuracy.

5.11 Computing item similarity

We compute the cosine similarity between items by comparing identical features based on the item category. During the listing of items in the online portal, each item is placed into a category manually. Hence, item *noodle* goes to the food category and the fast-food sub-category, while the *smartwatches* are placed into a wearable device under the electronics category. Item similarity attempts to find how suitable it is to recommend a user *noodle* if he views a *smartwatch.*

Table 3 Counting top *n* recommendations: Step 2

Item also viewed	Also viewed count (norm) (c)	Item viewed location (l)	Also viewed location (l)	Recommendation score ($t \times l \times c \times p$)
D	1	Loc1	Loc1	$1 \times 1 \times 1 \times 1 = 1$
E	0.9	Loc1	Loc2	$1 \times 0.8 \times 0.9 \times 1 = 0.72$
F	0.73	Loc3	Loc3	$0.9 \times 1 \times 0.73 \times 1 = 0.65$
G	0.8	Loc3	Loc4	$0.9 \times 0.8 \times 0.8 \times 1 = 0.57$
H	1	Loc3	Loc5	$0.9 \times 0.8 \times 1 \times 1 = 0.72$
I	0.61	Loc6	Loc6	$0.8 \times 1 \times 0.61 \times 1 = 0.48$
J	0.46	Loc6	Loc7	$0.8 \times 0.8 \times 0.46 \times 1 = 0.29$

t timestamp, *l* location, *c* count, *p* preferences

The similarity between items is calculated based on how similar their categories are. A novel, computationally faster approach is introduced to achieve this. A numerical category value is assigned against each item. All electronics items are assigned into category values from 1 to 200; food items are assigned into category values of 201 to 400, and so on. Computing cosine similarity on numerical categories reveals how related the two items are. For instance, consider computing the similarity between a television and a camera. A television (product *A*) may fall under the following category: Appliances (category id 1) - LCD (category id 4) - 3D LCD (category id 6). And a camera (product B) is under the following category: Electronics (category id 1) - camera (category id 35) - DSLR camera (category id 95). We can calculate the item similarity strength of product *A* and *B* at a scale of 0 to 1 by the following equation:

$$S_{\mathrm{A,B}} = cos(\theta) = \frac{(A \times B)}{(|A||B|)} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sum_{i=1}^{n} \sqrt{A_i^2} \times \sqrt{B_i^2}} \tag{4}$$

$S_{\mathrm{A,B}}$ is the similarity between product A and B, $cos(\theta)$ is the angle between edges A and B.

Using Eq. (4), we can calculate similarity strength between a television and a camera represented as:

A = (1, 4, 6)

B = (1, 35, 95)

Numeric values represent feature vectors of items *A* and *B*.

Cosine Similarity $S_{\mathrm{A,B}} = \frac{1\times1+4\times35+6\times95}{\sqrt{1^2+4^2+6^2}\times\sqrt{1^2+35^2+95^2}} = 0.964.$

6 Lifelong learning model for recommender system

This section discusses lifelong learning in MALA through batch and stream ensemble. In the traditional big data systems, feature vectors initialise themselves with historical batch data and persist data into Hadoop Distributed File System (HDFS) during the training phase. A long-running batch schedule trains the vector, and responses are again stored in HDFS. One of the primary constraints with this setting is that the full dataset may not be available at the initiation of a training schedule. The other key challenge is, the vectors can be updated with new events coming over time. Therefore, an iterative method is proposed that transforms the model training process into a *lifelong learning machine* [52, 53]. The Streaming model initialises itself with *saved learning* with the batch mode by loading the trained model from the Hadoop Distributed File System (HDFS) into a distributed memory. The MALA updates its streaming model incrementally on each new wave of incoming data. The framework further allows the merging of static historical data pools with the most updated streaming model. The method removes any cold-start situations at the training initiation. Figure 3 shows a screenshot of the learning process running in the command line.

Let us consider a scenario with the entities *users* and *items*, in an e-commerce portal. The entities are represented as high-dimensional feature vectors. The Correlation between the user and item (r_{ui}) in the shopping portal can be defined as:

$$f(r_{ui}) = v_u v_i \tag{5}$$

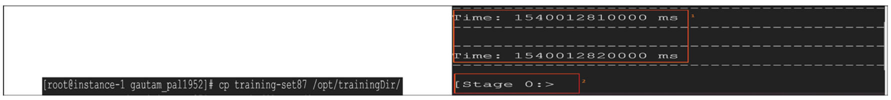
v_u and v_i are the feature vector for the user and item, respectively. Modelling the interaction between high-dimensional feature vectors (user and item) has a significant limitation of responsiveness, especially with big data. Also, the vectors are updated with new raw events arriving over time. Therefore, the constant need arises to revise the feature vectors leading to develop an iterative approach. With the iterative approach, the vectors initialise with historical batch data and persist them into Hadoop Distributed File System (HDFS). A stream processor loads the model from HDFS and incrementally builds on top of it. So, Eq. (5) is updated to accommodate the incremental (delta) learning as follows:

$$f(r_{ui}) = v_u v_i + \delta_u \delta_i \tag{6}$$

The model learns v_u, v_i offline and δ_u, δ_i at an iterative fashion with small amount of streaming data. δ_u and δ_i are the most current updates or fresh new rows of data. The dimensionality of δ_u (or, δ_i) is the same as v_u (or, v_i). However, the data size is much smaller, only the volume of most recent data collected at a streaming window length of a few minutes and therefore not very large. The method allows the model to keep updated at low-latency on a small amount of incremental data.

A *dimension reduction* approach can be taken for vectors [54], producing a faster training time. So, δ_i or δ_j need not be of the same dimension as the user or item vector. Instead, only the modified columns can update the training model through the online process. The approach reduces the dimension of streaming learning data and the online learning time since the online model only needs to learn the *correction* over the batch offset. Nevertheless, in the current dataset, the search for the rows and columns that requires an update and subsequently merging the incremental result with the batch is computationally expensive on a large volume of the data pool. Hence, the approach to *retraining* the model on a small window interval of data pool through distributed in-memory processing of Apache Spark provides a faster response time. The lifelong learning method is described in Algorithm 2.

Lifelong learning has brought several benefits to recommendation accuracy. The recommendation method described in the preceding section can be extended using



```
[root@instance-1 gautam_pal1952]# cp training-set87 /opt/trainingDir/
```

```
Time: 1540012810000 ms
Time: 1540012820000 ms
[Stage 0:>
```

Fig. 3 Screenshot showcasing the incremental streaming learning. MALA initialises with historical batch data and updates the streaming model incrementally on each new wave of incoming data. The model runs indefinitely at a 10 sec window interval. The training folder is updated incrementally with new data (Fig. a). The model is retrained simultaneously with a small amount of new data as they appear (Fig. b). Figure b shows the sliding window interval for the model retraining every 10 sec. The training starts with multiple stages as soon as a new dataset is copied to the training folder

lifelong learning to update recommendations to the newly arriving dataset at faster intervals. The knowledge Miner component of MALA enables an incremental learning environment.

Algorithm 2 presents the novel lifelong learning method for a recommendation engine using incremental updates.

Algorithm 2 Recommendation Using Lifelong Incremental Learning Algorithm

```
Input: historical datastore for batch $d_b$, collection of streaming dataset for
    the last *sliding window* $d_s = \sum_{i=1}^{n} d_i$, Kafka topic $k_t$, DB server IP $d_a$,
    Knowledge Base $k_b$
Output: : top $n$ recommended product list
1: $C_b \leftarrow$ CollaborativeFiltering($d_b$)
2: $S_b \leftarrow$ ItemSimilarity($d_b$)
3: subscribeKafka($k_t$, $d_a$);
4: for windowed dataset $d_i$ do
5:     Storm consumes Kafka queue
6:     $C_s \leftarrow$ CollaborativeFilteringRule($d_i$)
7:     $R_c \leftarrow$ Update($C_b, C_s$) //for the first loop
8:     $R_c \leftarrow$ Update($R_c, C_s$) //collaborative filtering results are updated with
       stream
9:     $S_s \leftarrow$ SimilarityRule($d_i$)
10:    $R_s \leftarrow$ Update($S_b, S_s$) //for the first loop
11:    $R_s \leftarrow$ Update($R_s, S_s$) //item similarity results are updated with stream
12:    R $\leftarrow R_c \cap R_s$
13:    $k_b \leftarrow R_c$, $R_s$, $R$
14:    return $k_b$
15: end for
```

Step 1 (Batch processing rule, lines 1–2): Collaborative Filtering and items similarly based rules compute the initial recommendations.

Step 2 (Stream processing rule, lines 7–9): Two forms of recommendations are computed at the streaming setting. Their respective results are merged with the batch counterpart. The method makes the model agile to update at shorter intervals.

Step 3 (Computing the final recommendations, lines 10–13): The elements common between Collaborative Filtering and the similarity rule are preserved, and the rest are discarded. The final training model is added back to the datastore for future use, and the Knowledge Base is subsequently updated.

7 Experiments

Users' *click-to-view* data is high in volume and velocity compared to data generated only from purchases. We develop a robust big data environment to support the enormous storage and processing requirement. Experiments are carried out in Amazon Cloud Services (AWS) for Cassandra, Kafka, and Hadoop installation. The Apace Storm is installed in the *Microsoft Azure cloud*.

Users' click to view items generates a considerable volume of clickstream data. One of the challenging aspects of MALA is ingesting a high volume of incoming data into its big data store. This section provides an insight into managing the high-velocity ingested data pool in a distributed and fault-tolerant manner.

7.1 Real-time clickstream data ingestion

Clickstream is the activity recording of parts of the screen a user views while web surfing (user footprint) [55]. User activity is picked up from the client-side browser. Therefore, clickstream data is the URL generated from each user's click data. In an online shopping portal, a clickstream data may look like this: http://smartbuy.com:/electronics/phone/items?userID=id10&productName=iPhonexs&cost=700&geolocation=NewYork. Where each context parameter like *product name*, *price,* and *location* are appended to each clickstream event.

In a context-aware recommendation system, we design a big data event capture framework that ingests every end-user click from a customer touchpoint. The customer's location and event time are extracted from clickstream and appended into each session object. A session object is associated with each time the user newly opens the e-commerce website. For each click event, a *session ID* and a *context ID* are created. The context is a uniquely derived object created from the session object generated at the JavaScript layer. *Context ID* is associated with each user's click event stored in the datastore and used in computing recommendations.

Kafka is a popular option for stream data retrieval. Kafka is able to achieve ingesting a high-velocity, a large volume of data that requires rapid, fault-tolerant, distributed pipelines. Kafka, as a distributed client-server-oriented publisher-subscriber message bus system, substitutes the conventional message buses like IBM MQ, Rabbit MQ, etc., owing to its higher throughput, failover, and replication abilities. Kafka is the principal hub for our real-time processing environment when applied along with Apache Storm stream processing APIs [56]. Refer Fig. 4 for the Kafka cluster we used in the data ingestion layer.

Recommender systems get input from different sources to make recommendations. The most common way of collecting input is through the user's feedback. Shopping portals, for instance, collect ratings provided by the users. Nevertheless, *explicit feedback* may not be available always. Thus, a number of techniques were adopted to build recommendations through users' *implicit feedback* [6]. Users' click data or clickstream is one of the primary sources for deriving implicit feedback. Clickstream contains the user's location data which is derived from the client's IP address. Country, region, state, and city are part of location data. Each user clicks on an item generates an event and clickstream, which is captured by the Apache Kafka message queue, and data is moved from the source system to the analytics processing system on a near real-time basis. Apache Storm processes data in real-time and stores it into a Cassandra data store. Context-aware, real-time data stored in Cassandra is processed through a batch job that runs periodically at an interval of 6 h. The result of the batch job is an item recommendation table which is again stored back in Cassandra DB. The recommendation table is accessed from UI through RESTful API written in the Java Spring framework. The end-to-end big data ingestion framework is shown in Fig. 5.

7.2 MovieLens dataset

The IFBRS is examined on a real-world dataset from movielens.org [57], and performance is compared with Spark MLlib ALS API [58]. The *Movielns data* contains up to 27,000 movies by 138,000 users. Since the IFBRS does not predict the ratings, instead, it ranks the recommended products on the overall score (Table 4). Subsequently, ranks are validated against the ratings predicted by Apache Spark ALS API.

7.3 Setting up a storm cluster

For the Storm setup, the *Microsoft Azure HDInsight* Storm cluster is used for setting up a *nine* nodes Linux setup on Ubuntu OS. Unlike Farahabady [3], a heterogeneous Storm cluster was not considered for Nimbus or Supervisor to make the most out of the default scheduler and load balancer. See Fig. 6.

The recommender application is developed as Java Spring REST Web service and deployed into Tomcat through Datastax driver [5], which is a popular Cassandra Java client driver.

The Movielens DB has three tables with schema shown in Table 5.

7.4 Data preparation

Table schema 5 is used to create a user-movie table containing the following column: UserID, MovieID, and Zip-code. Hadoop Pig scripts transform data consisting of *movies also-viewed* across all the user bases through MapReduce. However, in the current model with the *Movielens Database*, we cannot determine if each user's *also-viewed* products are generated in a single session or multiple sessions since the algorithm is based on the assumption that items are considered *also-viewed* only if those items are viewed in the same browsing session. Therefore, it is a known limitation with the current data source that can be overcome with the availability of production clickstream data. The user's *location* is the *state* from where clickstream data is generated. The *state* as the location is computed from the zip code. Also-viewed products are computed based on the scenarios where products are viewed, and products also-viewed are from the *same or different location* (State). Based on that, Algorithm 1 computes the *top n recommended products*.

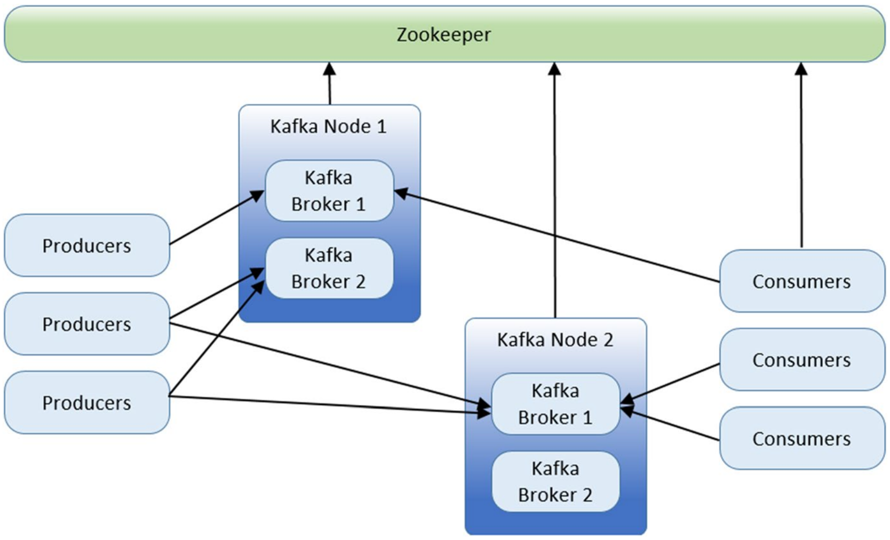


Fig. 4 Multi-node, multi-broker Kafka cluster. A *Broker* is the actual Kafka process. Producer ingests data into multiple broker components for load distribution and parallel processing. The distributed architecture provides a robust failover and faster processing through load balancing

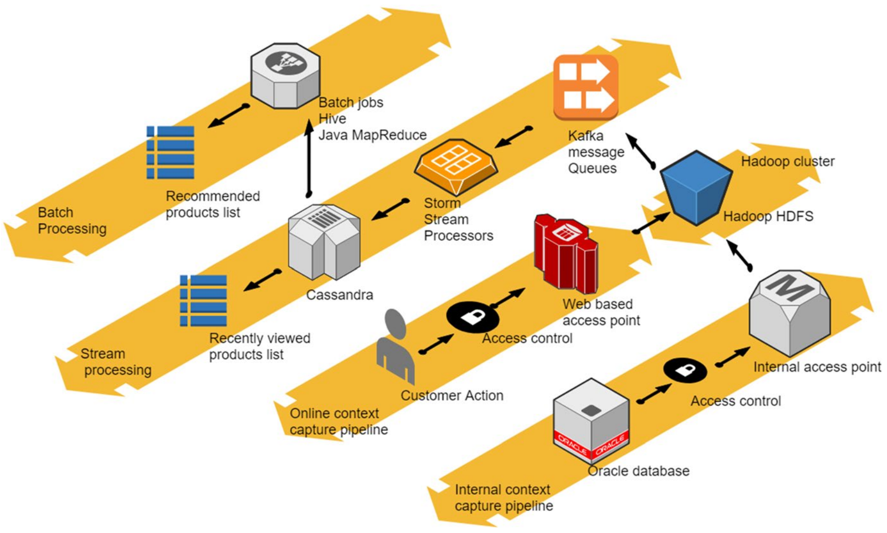
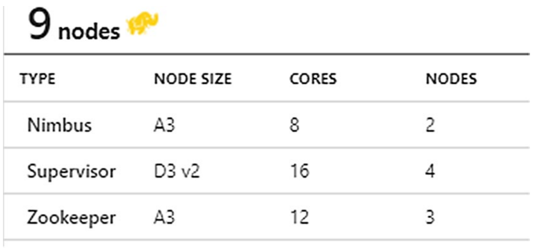


Fig. 5 End-to-end flow: Ingestion to the recommendation. Each user's click on an item generates an event and clickstream, which is captured by the Apache Kafka message queue, and data is moved from the source system to the analytics processing system on a near real-time basis. Apache Storm processes data in real-time and persists them into a Cassandra data store. An online context-capture pipeline stores users' click data, while an internal context-capture pipeline stores item features as historical data as recorded manually for each item. An efficient blend of historical batch data with a real-time stream creates a lifelong learning environment for a recommender system

Table 4 AWS instance types

Instance type	Instance count	vCPUs	Memory	Instance storage	EBS optimised bandwidth
m1.large	3	2	7.5 GB	500 GB	Moderate


9 nodes

TYPE	NODE SIZE	CORES	NODES
Nimbus	A3	8	2
Supervisor	D3 v2	16	4
Zookeeper	A3	12	3

Fig. 6 Microsoft azure HDInsight storm cluster

7.5 Evaluating IFBRS

Evaluation of implicit-feedback-based recommender systems can be tricky. Numeric scores are available in explicit rating-based recommender systems as a benchmark, and precise accuracy can be measured using the root mean squared error. However, evaluating the success of an implicit model cannot use traditional evaluation methods in the absence of ratings. The implicit model rather can use *users' reaction* to the recommendation by clicking or viewing the recommendations. The implicit model can as well compare the ranking orders of the recommendations with actual ratings provided by the user in the test dataset. The easiest way to evaluate the efficiency of an implicit-feedback-based recommender system is by verifying if a user actually clicked the displayed recommended items. The percentage of recommended items clicked and subsequently carried until purchase provides a measurable conversion rate, i.e., click-through rate (CTR). Nevertheless, in the experiment setup, any CTR data is not available in the absence of real users *clicking* on the recommendations. Thus, an offline method is used to examine the overall accuracy of IFBRS. Spark MLlib ALS APIs [58] are used to predict future ratings. Developed in Scala, parameters are configured as shown in the Listing 1:

Table 5 Movielens database schema

Table	Column names
Rating	UserID, MovieID, Rating, Timestamp
Users	UserID, Gender, Age, Occupation, Zip-code
Movies	MovieID, Title, Genres

Listing 1: Recommendation parameters for Spark ALS

```
val als = new ALS()
  .setMaxIter(5)
  .setRegParam(0.01)
  .setUserCol("UserID")
  .setItemCol("MovieID")
  .setRatingCol("Rating")
val alsModel = als.fit(training)
val predictions = alsModel.transform(test)
```

Predicted ratings by the Spark ALS are Validated with the IFBRS rankings. If any predicted ALS ratings match with the actual ratings in the test database, the corresponding records are compared with the ranking produced by the IFBRS. The method is considered successful if an item rated as *five* by ALS is ranked among the top *three* by the ranking algorithm. See Table 6 and Fig. 7.

7.6 Evaluation of precision

The actual Click Through Rate (CTR) cannot determine if a user actually clicks on the recommended items in the test environment. Therefore, an alternative method is proposed to calculate the precision of the recommended items. User Satisfaction Index (USI) [10] is used to define precision as follows:

$$USI(i) = \frac{|A_i \bigcap B_i|}{min(|A_i|, |B_i|)} \times 100\% \tag{7}$$

The equation reveals how closely recommended items are related to the actual user provided ratings. In particular, if an item is ranked among the top *five* recommended items A_i, then the actual rating (B_i) is expected as *four* or *five*. Precision is defined as follows:

$$P = \frac{1}{n}\sum_{i=1}^{n} USI(i) \tag{8}$$

In principle, precision denotes the average satisfaction index across all the user bases [10].

Table 7 is based on the precision function defined in Eq. (8). Precision accuracies in IFBRS shown in bold are 2.4% greater than SVM, 8.6% greater than Logistic Regression, and more than 30% greater than all other algorithms. Therefore, compared with alternative models, IFBRS improves classification and regression accuracy significantly on multiple scales and parameters. See Fig. 8.

7.7 Evaluation: an alternative approach

An alternative and *more measurable* approach is presented in this section. Spark ALS ratings are converted into ranks and compared with IFBRS. The method evaluates metrics more effectively by comparing accuracy under the same scale, i.e., ranking order (see Fig. 7). Also, the Root Mean Square Error (RMSE) is computed based on the ranking system under the latent factors such as location, time, and user preferences. We can verify how each factor independently and together affects the overall accuracy of recommendations. The following steps are performed for computing RMSE: (i) The ratings are computed for each of the recommended products with Spark ALS, (ii) ratings are converted into relative ranks for each of the latent factors, (iii) RMSE are computed using user-provided actual rating and Spark ALS predicted rating. Similarly, RMSE for IFBRS rankings is obtained with each of the latent factors while keeping the user-provided rating as the reference (Fig. 9). Observe, the recommendation provided by the IFBRS is improved in accuracy after applying latent factors. The accuracy is significantly improved by employing all of the factors together. Figure 10 shows RMSE for IFBRS batch and *lifelong learning model*. Incremental lifelong learning produces better accuracy compared to one-shot batch learning.

7.8 Load tests of MALA

This subsection provides system performance under stress. Apache Storm, the stream processor unit, and Cassandra, the storage engine of MALA, are evaluated for load testing. Three million records perform the write operation first and run a mixed type of load (read plus write) for another 3 million records. The server setup is shown in Table 8. Refer to Figs. 11 and 12 for test results using Datastax OPSCenter platform. The results validate the scalability of the system under stress.

Table 6 Comparing IFBRS with Spark ALS

Item number	Actual rating	Spark ALS prediction	Top *n* recommendation algorithm order
2987	4	4.1	10
1250	5	4.3	1
3791	4	3.9	12
858	3	2.6	9
1304	3	3.2	8
3791	2	2.5	18
2746	4	4.3	5
260	5	4.6	1
150	4	3.7	4
2987	1	2.5	19
3448	3	3.8	12

Evaluation compares Spark ALS with IFBRS rankings for accuracy. All *five* ratings are expected to rank among the top *three*

8 Discussion

8.1 Comparing IFBRS with spark ALS

As shown in Sect. 7.7, 9, the IFBRS is considerably more accurate than Spark ALS APIs when context parameters, e.g., location and time decay function, are considered. The algorithm's approach putting more weight on recent data over historical data increases the *relevancy* of the recommendations and results in a higher *Click Through Rate*. Results assert our claim that the user's *context parameters* significantly improve recommendation quality in personalised recommendations.

8.2 Trade-off between low-latency and high-accuracy

The MALA consists of both low-latency real-time components as well as its high-accuracy batch counterpart, which works simultaneously on the same dataset. The cost of a batch and a real-time algorithm is defined by its cost (time, space, and resource utilisation complexity). We calculate the competitive ratio of a streaming algorithm against a batch algorithm. A smaller than *one* cost ratio will prove the superiority of the streaming process. However, the competitive ratio due to time complexity turns out to be greater than one in all scenarios and particularly worse in a certain dataset. This is less surprising because the streaming algorithm processes in mini-batches without the foresight of the entire dataset. On the contrary, real-time algorithm benefits from space and resource utilisation complexity due to much less volume of data processing in each mini-batches. A real-time algorithm is termed α competitive if there are positive factors α and γ so that:

$$v_i \leq \alpha v_b + \gamma \tag{9}$$

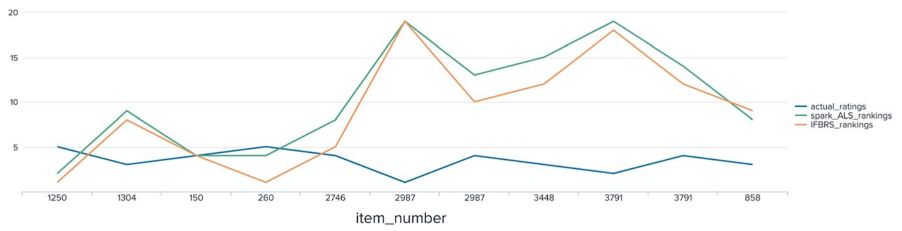


Fig. 7 IFBRS recommendation distribution is significantly closer to the user-provided actual rating compared to the Spark ALS. The results compare favourably to IFBRS than the standard Apache Spark ALS

Table 7 Model prediction accuracy for recommended items

Algorithm	Precision	Recall	Accuracy	F1 score
GaussianNB	0.15	0.27	0.19	0.14
Bernoulli NB	0.42	0.45	0.44	0.51
Decision tree classifier	0.55	0.64	0.70	0.60
SVM	0.79	0.57	0.63	0.57
Logistic regression	0.74	0.54	0.53	0.51
Proposed IFBRS	**0.81**	**0.82**	**0.82**	**0.81**

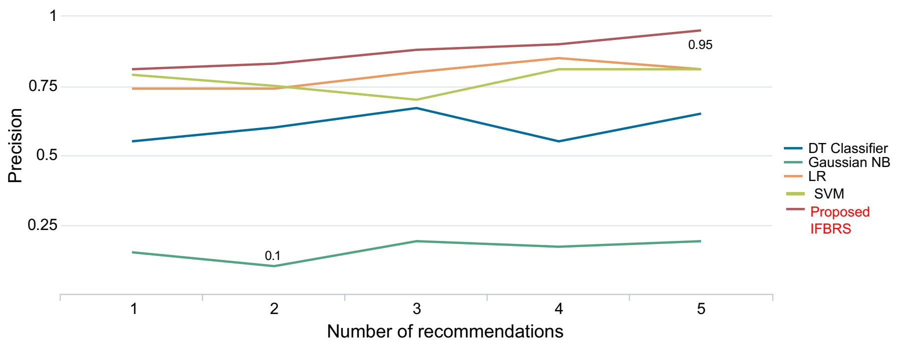


Fig. 8 Model precision comparison against the number of recommended items

v_i is the cost for the streaming algorithm. v_b is the cost at the batch setting.

From Eq. (9), we conclude, an α competitive real-time algorithm has cost no inferior to α times to the optimal batch algorithm (v_b) plus some initial advantages (γ)

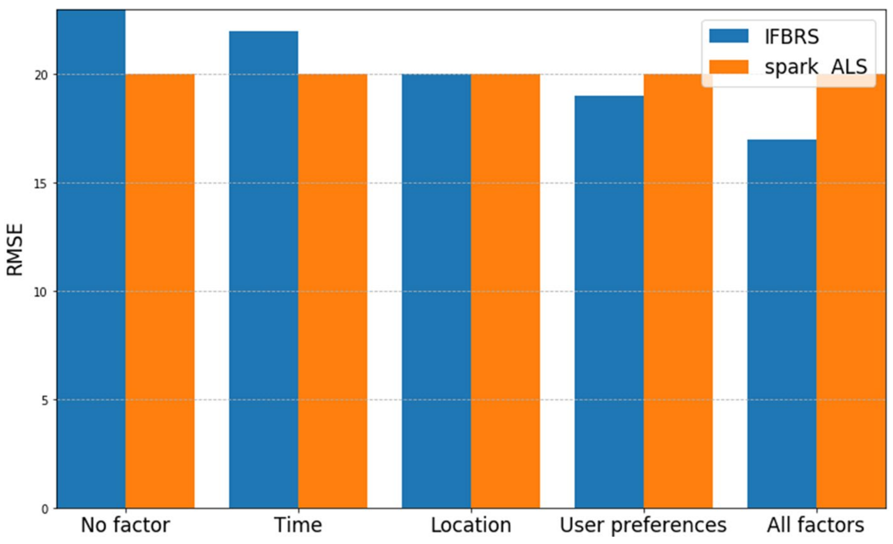


Fig. 9 RMSE for IFBRS and Spark ALS. RMSE is computed by applying the latent factors. The best accuracy is achieved when all of the factors are put together

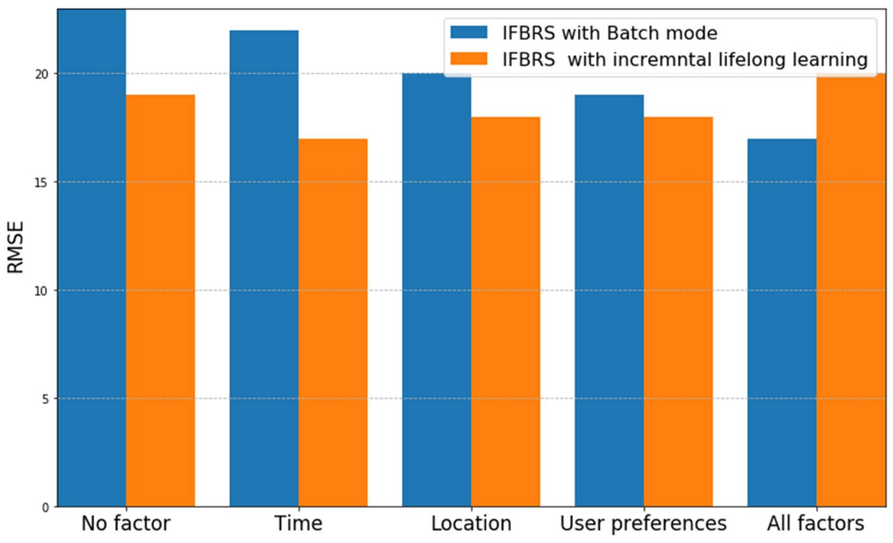


Fig. 10 RMSE for IFBRS one-shot batch learning and *lifelong learning model*. RMSE is computed for the batch and incremental lifelong learning models of IFBRS over the latent factors. The lifelong learning model produces better results by integrating both batch and stream processing methods

assigned to an optimised batch algorithm due to its prior knowledge of the entire dataset.

In a classic trade-offs between low-latency or high-accuracy, our real-time case study selects low-latency, real-time responses over *high accuracy* and *strong consistency*. Our model uses Cassandra DB to support low-latency requirements for the real-time component. Cassandra initiates a *read repair* to update the inconsistent data in a situation with a higher consistency setting. The client read-write processes, therefore, must wait before discrepancies are eliminated. Big Data systems needing swift turnaround time at a near-real-time puts the lowest likely value for consistency (which is, CONSISTENCY ONE) due to the negative effect of consistency levels on general responsiveness. As Cassandra allows tunable consistency settings,

Table 8 Cassandra Setup

Cassandra vendor	Number of records	Replication factor
DSE	6 Million writes	3

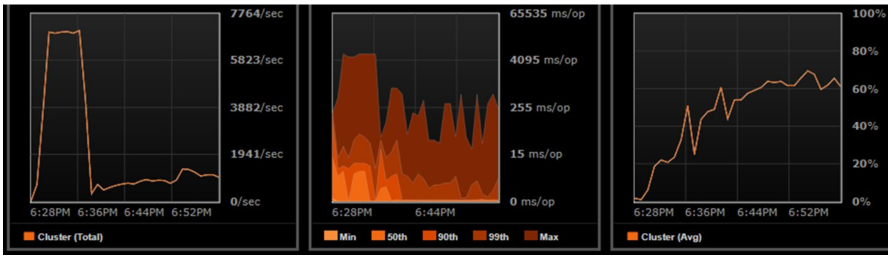


Fig. 11 Statistics showing (i) Read Requests, (ii) Read Request Latency, and (iii) OS Disc Utilisation. *(i) Read Requests:* per second read requests count on all coordinating nodes in the cluster. It Analyses the number of requests for a given period that reveals the system read overhead and usage trends. *(ii) Read Request Latency (Percentiles):* 99th, 90th percentiles, min, max, the median for a client reads. When a node accepts a client read request, the time period initiates, and it terminates while the node replies back to the client. Depending on the replication factor and consistency setting, this might include the network delay from the replicas. *(iii) OS Disc Utilisation*: CPU time used by disc I/O. The time unit is in milliseconds

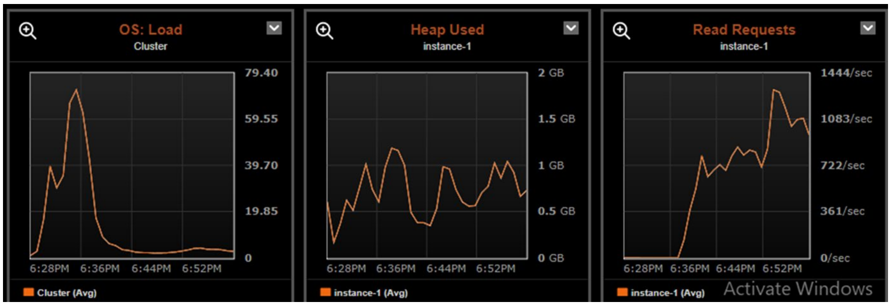


Fig. 12 Statistics shown for: (i) OS Load, (ii) Heap Used, and (iii) TP Flushes Completed. *(i) OS Load:* Operating system load average for every 1 min. *(ii) Heap Used:* Average of Java heap space utilised.*(iii) TP Flushes Completed:* Number of memtables flushed to disc since the nodes are started

this enables the proposed model to let Cassandra act like CP (partition tolerant and consistent) or AP (partition tolerant and available) system with regards to the CAP theorem.

8.3 Validation by click-through rate (CTR)

For validation of recommendation accuracy, we have adopted an offline strategy where 80% of data was used for the recommendation and 20% for validation. However, the Click-Through Rate (CTR) and purchase rate (PR) are arguably definitive ways to verify recommendation accuracy. CTR and PR determine users' preferences on viewing or buying from recommended items versus non-recommended items. As CTR or PR verification will need a production deployment with a large set of the real user base, it was not a feasible option for us at this stage and a known limitation of the model.

9 Conclusions

We developed an improved version of the implicit feedback-based contextual recommender system through Big Data collaborative Multi-agent Lambda Architecture and a lifelong learning approach. The aim is to provide a solution for an intelligent blend of historical batch data with a real-time stream towards developing a lifelong learning machine. The Architecture allows us to use past knowledge, update and accumulate information iteratively over a large volume of data pool through a host of big data tools and methods. Graceful interaction of stream and historical batch data provides deeper insights at low-latency.

We brought a number of unique features to the *Implicit Feedback-Based Recommender System*. The novelty of the recommender approach is in the use of *contextual parameters* and a *weighted hybridisation* strategy to mix historical batch data with near-real-time data predicting the recommendation accurately. The trained model keeps improving over time with an incremental lifelong learning method. Another important distinction in the algorithm is that the model does not rely on the explicit rating provided by the user; rather, it computes on user's passive endorsements by click data and does not suffer from the sparsity problem of rating-based user similarity approaches. The algorithm does not enforce the user to log in to provide recommendations and is capable of providing accurate recommendations for non-logged-in users. Distinct advantages of the proposed methods over Hadoop MapReduce and Spark ML APIs are in terms of improved accuracy, response time, reduced training time, handling cold start situations, and significant infrastructure cost savings.

This work introduces *hybrid lifelong learning through Lambda Architecture* which leaves a few interesting open questions further to investigate the lifelong learning model in our future works.

Funding This research was funded by Accenture Technology Labs, Beijing, China. Grant Number: RDF 15-02-35. Project code: RDS10120180003.

References

1. Yao L, Sheng QZ, Ngu AHH, Yu J, Segev A (2015) Unified collaborative and content-based web service recommendation. IEEE Trans Serv Comput 8(3):453–466. https://doi.org/10.1109/TSC.2014.2355842
2. Kim H, Madhvanath S, Sun T (2015) Hybrid active learning for non-stationary streaming data with asynchronous labeling, In: IEEE International Conference on Big Data (Big Data), pp 287–292. https://doi.org/10.1109/BigData.2015.7363766
3. Lee CH, Lin CY (2017) Implementation of lambda architecture: a restaurant recommender system over apache mesos, In: IEEE 31st International Conference on Advanced Information Networking and Applications (AINA), pp 979–985. https://doi.org/10.1109/AINA.2017.63
4. Batyuk A, Voityshyn V (2018) Apache storm based on topology for real-time processing of streaming data from social networks, In: 2016 IEEE First International Conference on Data Stream Mining & Processing (DSMP), pp 345–349. https://doi.org/10.1109/DSMP.2016.7583573
5. Hanif M, Yoon H, Jang S, Lee C (2017) An adaptive sla-based data flow mechanism for stream processing engines, In: International Conference on Information and Communication Technology Convergence (ICTC), pp 81–86. https://doi.org/10.1109/ICTC.2017.8190947
6. Hu Y, Koren Y, Volinsky C (2008) Collaborative filtering for implicit feedback datasets, In: Eighth IEEE International Conference on Data Mining, pp 263–272. https://doi.org/10.1109/ICDM.2008.22
7. Collaborative filtering - RDD-based API. https://spark.apache.org/docs/2.2.0/mllib-collaborative-filtering.html. Accessed 20 Sept 2021
8. Wang J, Peng X, Xing Z, Fu K, Zhao W (2017) Contextual recommendation of relevant program elements in an interactive feature location process, In: IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp 61–70. https://doi.org/10.1109/SCAM.2017.14
9. Ren Y, Tomko M, Salim FD, Chan J, Clarke C, Sanderson M (2017) A location-query-browse graph for contextual recommendation. IEEE Trans Knowl Data Eng 30(2):204–218. https://doi.org/10.1109/TKDE.2017.2766059
10. Rahman MM (2013) Contextual recommendation system, In: International Conference on Informatics, Electronics and Vision (ICIEV), pp 1–6. https://doi.org/10.1109/ICIEV.2013.6572542
11. Kharrat FB, Elkhleifi A, Faiz R (2016) Recommendation system based contextual analysis of facebook comment, In: IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA), pp 1–6. https://doi.org/10.1109/AICCSA.2016.7945792
12. Domingues MA, Sundermann CV, Manzato MG, Marcacini RM, Rezende SO (2014) Exploiting text mining techniques for contextual recommendations, In: IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), Vol 2, pp 210–217. https://doi.org/10.1109/WI-IAT.2014.100
13. Xie F, Xu M, Chen Z (2012) Rbra: A simple and efficient rating-based recommender algorithm to cope with sparsity in recommender systems, In: 26th International Conference on Advanced Information Networking and Applications Workshops, pp 306–311. https://doi.org/10.1109/WAINA.2012.11

14. Sharifi Z, Rezghi M, Nasiri M (2014) A new algorithm for solving data sparsity problem based-on non negative matrix factorization in recommender systems, In: 4th International Conference on Computer and Knowledge Engineering (ICCKE), pp 56–61. https://doi.org/10.1109/ICCKE.2014.6993356
15. Reshma R, Ambikesh G, Thilagam PS (2016) Alleviating data sparsity and cold start in recommender systems using social behaviour, In: International Conference on Recent Trends in Information Technology (ICRTIT), pp 1–8. https://doi.org/10.1109/ICRTIT.2016.7569532
16. Thrun S (1998) Lifelong learning algorithms. Learning to learn. Springer, Boston, MA, pp 181–209
17. Thrun S (1996) Explanation-based neural network learning: a lifelong learning approach. Kluwer Academic Publishers, Boston, MA
18. Silver DL (1996) The parallel transfer of task knowledge using dynamic learning rates based on a measure of relatedness. Connect Sci 8(2):277–294. https://doi.org/10.1080/095400996116929
19. Silver DL, Mercer RE (2002) The task rehearsal method of life-long learning: overcoming impoverished data. In: Cohen R, Spencer B (eds) Advances in artificial intelligence. Springer, Berlin, Heidelberg, pp 90–101
20. Silver DL, Poirier R (2004) Sequential consolidation of learned task knowledge. In: Tawfik AY, Goodwin SD (eds) Advances in artificial intelligence. Springer, Berlin, Heidelberg, pp 217–232
21. Silver DL, Mason G, Eljabu L (2015) Consolidation using sweep task rehearsal: overcoming the stability-plasticity problem. In: Barbosa D, Milios E (eds) Advances in artificial intelligence. Springer International Publishing, Cham, pp 307–322
22. Hong X, Wong P, Liu D, Guan S-U, Man KL, Huang X (2018) Lifelong machine learning: outlook and direction, In: Proceedings of the 2nd International Conference on Big Data Research, ACM, pp 76–79
23. Hong X, Pal G, Guan S-U, Wong P, Liu D, Man KL, Huang X (2019) Semi-unsupervised lifelong learning for sentiment classification: less manual data annotation and more self-studying, In: Proceedings of the 2019 3rd High Performance Computing and Cluster Technologies Conference, HPCCT 2019, ACM, New York, NY, USA, pp 87–92. https://doi.org/10.1145/3341069.3342992
24. Fei G, Wang S, Liu B (2016) Learning cumulatively to become more knowledgeable, In: Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, ACM, New York, NY, USA, pp 1565–1574. https://doi.org/10.1145/2939672.2939835
25. Ruvolo P, Eaton E (2013) ELLA: An efficient lifelong learning algorithm, In: Dasgupta S, McAllester D (eds.), Proceedings of the 30th International Conference on Machine Learning, Vol. 28 of Proceedings of Machine Learning Research, PMLR, Atlanta, Georgia, USA, pp 507–515. http://proceedings.mlr.press/v28/ruvolo13.html
26. Ruvolo P, Eaton E (2013) Ella: an efficient lifelong learning algorithm, In: International Conference on Machine Learning, pp 507–515
27. Caruana R (1997) Multitask learning. Mach Learn 28(1):41–75
28. Chen Z, Ma N, Liu B (2015) Lifelong learning for sentiment classification, In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers), Vol 2, pp 750–756
29. Kumar A, Daume III H Learning task grouping and overlap in multi-task learning, arXiv preprint arXiv:1206.6417
30. Wang S, Chen Z, Liu B (2016) Mining aspect-specific opinion using a holistic lifelong topic model, In: Proceedings of the 25th International Conference on World Wide Web, International World Wide Web Conferences Steering Committee, pp 167–176
31. Liu Q, Liu B, Zhang Y, Kim DS, Gao Z (2016) Improving opinion aspect extraction using semantic similarity and aspect associations. In: Thirtieth AAAI conference on artificial intelligence
32. Carlson A, Betteridge J, Wang RC, Hruschka Jr ER, Mitchell TM (2010) Coupled semi-supervised learning for information extraction, In: Proceedings of the Third ACM International Conference on Web Search and Data Mining, ACM, pp 101–110
33. Mitchell T, Cohen W, Hruschka E, Talukdar P, Yang B, Betteridge J, Carlson A, Dalvi B, Gardner M, Kisiel B et al (2018) Never-ending learning. Commun ACM 61(5):103–115
34. Li L, Yang Q (2015) Lifelong machine learning test, In: Proceedings of the Workshop on Beyond the Turing Test of AAAI Conference on Artificial Intelligence
35. Salloum S, Dautov R, Chen X, Peng PX, Huang JZ (2016) Big data analytics on apache spark. Int J Data Sci Anal 1(3–4):145–164

36. Solaimani M, Iftekhar M, Khan L, Thuraisingham B, Ingram JB (2014) Spark-based anomaly detection over multi-source vmware performance data in real-time, In: IEEE Symposium on Computational Intelligence in Cyber Security (CICS), IEEE, pp 1–8
37. Rettig L, Khayati M, Cudré-Mauroux P, Piórkowski M (2015) Online anomaly detection over big data streams, In: IEEE International Conference on Big Data (Big Data), IEEE, pp 1113–1122
38. Guha S, Mishra N, Motwani R, O'Callaghan L (2000) Clustering data streams, In: Foundations of computer science, proceedings. 41st annual symposium on, IEEE, pp 359–366
39. Gupta M, Gao J, Aggarwal CC, Han J (2014) Outlier detection for temporal data: a survey. IEEE Trans Knowl Data Eng 26(9):2250–2267
40. Agarwal DK, Chen B-C (2016) Statistical methods for recommender systems. Cambridge University Press, New York
41. Pal G, Li G, Atkinson K (2018) Big data ingestion and lifelong learning architecture, In: IEEE International Conference on Big Data (Big Data), IEEE, pp 5420–5423
42. Pal G, Li G, Atkinson K (2018) Multi-agent big-data lambda architecture model for e-commerce analytics. Data 3(4):58
43. Heidrich J, Trendowicz A, Ebert C (2016) Exploiting big data's benefits. IEEE Softw 33(4):111–116. https://doi.org/10.1109/MS.2016.99
44. Xiang D, Wu Y, Shang P, Jiang J, Wu J, Yu K (2017) Rb-storm: resource balance scheduling in apache storm, In: 6th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI), pp 419–423. https://doi.org/10.1109/IIAI-AAI.2017.63
45. Farahabady MRH, Samani HRD, Wang Y, Zomaya AY, Tari Z (2016) A qos-aware controller for apache storm, In: IEEE 15th International Symposium on Network Computing and Applications (NCA), pp 334–342. https://doi.org/10.1109/NCA.2016.7778638
46. Yan L, Shuai Z, Bo C (2017) Multi-sensor data fusion system based on apache storm, In: 2017 3rd IEEE International Conference on Computer and Communications (ICCC), pp 1094–1098. https://doi.org/10.1109/CompComm.2017.8322712
47. Apache Cassandra 3.0 for DSE 5.0 (2021). https://docs.datastax.com/en/cassandra/3.0/. Accessed 20 Sept
48. Carpenter J, Hewitt E (2018) Chapter 12: Performance tuning. In: Cassandra: the definitive guide, 2nd edn. O'Reilly Media, Inc.
49. Thottuvaikkatumana R (2015) Data modeling considerations. In: Cassandra design patterns, 2nd edn. Packt Publishing Ltd.
50. Mass G, Garillot F (2018) Streaming application design, Chap 3. In: Learning spark streaming, O'Reilly Media, Inc.
51. Xia C, Jiang X, Sen L, Zhaobo L, Zhang Y (2010) Dynamic item-based recommendation algorithm with time decay. Sixth International Conference on Natural Computation, vol 1, pp 242–247. https://doi.org/10.1109/ICNC.2010.5582899
52. Thrun S (1996) Explanation-based neural network learning: a lifelong learning approach. Kluwer Academic Publishers, Boston, MA
53. Xia R, Jiang J, He H (2017) Distantly supervised lifelong learning for large-scale social media sentiment analysis. IEEE Trans Affect Comput 8(4):480–491. https://doi.org/10.1109/TAFFC.2017.2771234
54. Agarwal K, Chen B (2015) Statistical Methods for Recommender Systems. Cambridge University Press, Cambridge
55. Hanamanthrao R, Thejaswini S (2017) Real-time clickstream data analytics and visualization, In: 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT), pp 2139–2144. https://doi.org/10.1109/RTEICT.2017.8256978
56. https://www.linkedin.com/pulse/flume-kafka-real-time-event-processing-lan-jiang/, Accessed: 20 Sept. (2021)
57. https://grouplens.org/datasets/movielens/100k/ , Accessed: 20 Sept. (2021)
58. Winlaw M, Hynes MB, Caterini A, Sterck HD (2015) Algorithmic acceleration of parallel als for collaborative filtering: speeding up distributed big data recommendation in spark, In: IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), pp 682–691. https://doi.org/10.1109/ICPADS.2015.91