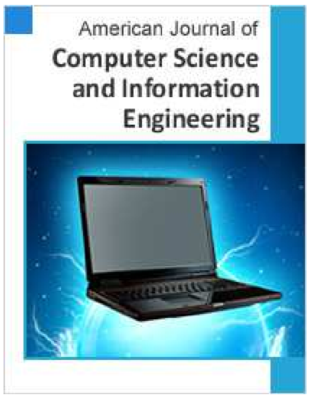
American Journal of Computer Science and Information Engineering
2015; 2(4): 33-44
Published online September 30, 2015 (http://www.aascit.org/journal/ajcsie)

American Association for Science and Technology



Introduction to a Framework of E-commercial Recommendation Algorithms

Loc Nguyen

Board of Directors, Sunflower Soft Company, Ho Chi Minh City, Vietnam

Email address
ng_phloc@yahoo.com

Citation
Loc Nguyen. Introduction to a Framework of E-commercial Recommendation Algorithms. *American Journal of Computer Science and Information Engineering*. Vol. 2, No. 4, 2015, pp. 33-44.

Keywords
Recommendation Algorithm, Recommendation Server, Middleware Framework

Received: March 29, 2015
Revised: September 6, 2015
Accepted: September 7, 2015

Abstract
Recommendation algorithm is very important for e-commercial websites when it can recommend online customers favorite products, which results out an increase in sale revenue. I propose the framework of e-commercial recommendation algorithms. This is a middleware framework or “operating system” for e-commercial recommendation software, which support scientists and software developers build up their own recommendation algorithms based on this framework with low cost, high achievement and fast speed.

1. Introduction

The product is the “Framework of e-commercial recommendation solutions”, named Hudup. This is a middleware framework or “operating system” for e-commercial recommendation software, which support scientists and software developers build up their own recommendation solutions based on this framework. The term “recommendation solutions” mentions the computer algorithms which aim to recommend online customers an introduction list of items such as books, products, services, news papers, fashion clothes, etc. on commercial websites with expectation that customers will like these recommended items. The goal of recommendation algorithms is to gain high sale revenue.

You need to develop a recommendation solution for online-sale website. You, a scientist, invent a new algorithm after researching many years. Your solution is excellent and very useful and so you are very exciting but:

1. You cope with complicated computations when analyzing big data and there are a variety of heterogeneous models in recommendation studies.
2. It is impossible for you to evaluate your algorithm according to standard metrics.
3. There is no simulation environment or simulator for you to test the feasibility of your algorithm.

The innovative product “Framework of e-commercial recommendation solutions” supports you to solve perfectly three above difficulties and so following are your achievements:

1. Realizing your solution is very fast and easy.
2. Evaluating your solution according to standard metrics by the best way.
3. Determining the feasibility of your algorithm in real-time applications.

The product has another preeminent function which is to provide two optimized algorithms so that it is very convenient for you to assess and compare different solutions. The product aims to help you, a scientist or software developer, to solve three above core problems. The product proposes three solution stages for developing a recommendation algorithm.

1. *Base stage*: builds up algorithm model and data model to help you to create new software with lowest cost.
2. *Evaluation stage*: builds up evaluation metrics and algorithm evaluator to help you to assess your own algorithm.
3. *Simulation stage*: builds up recommendation server or simulator, which helps you to test feasibility of your algorithm.

2. Related Works

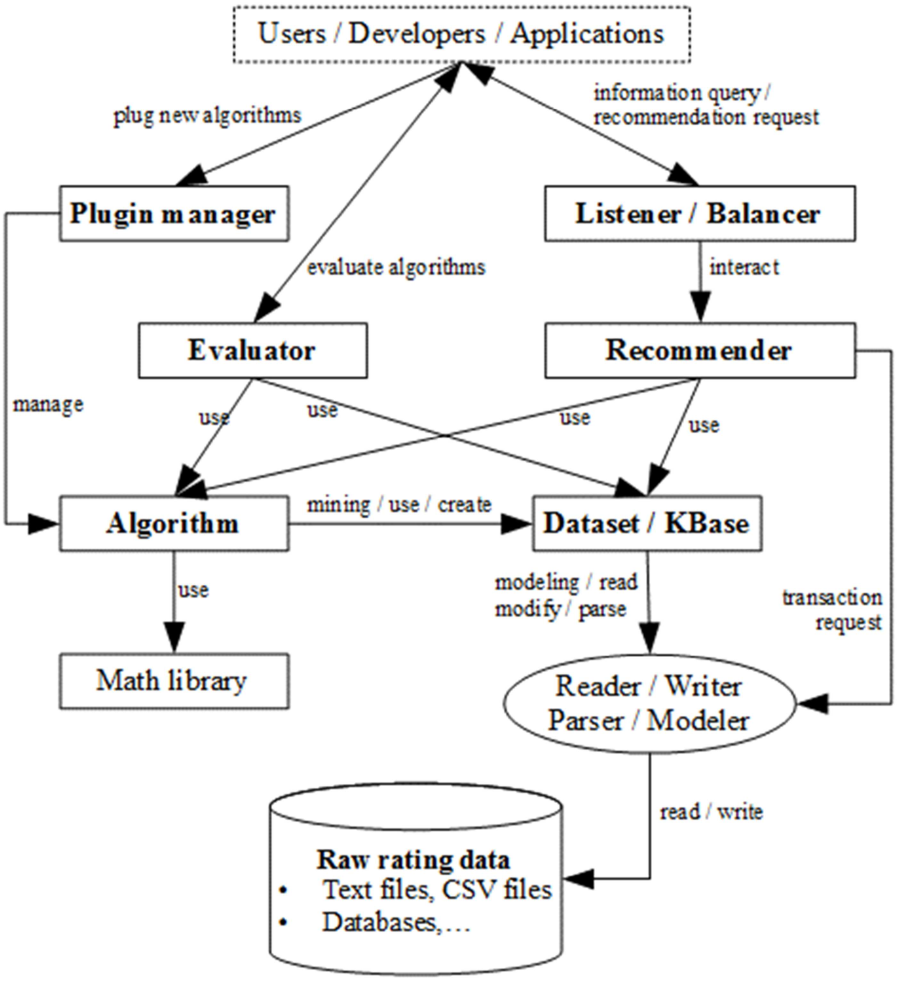


Fig. 1. *General architecture of product.*

There are now some other open source software similar to my product; the brief list of them is described as follows:

1. *Carleton* [11] is developed by Carleton College, Minnesota, USA. The software implements some recommendation algorithms and evaluates such algorithms based on RMSE metric. The software provides an implementing illustration of recommendation algorithms and it is not recommender system or framework. However, a significant feature of Carleton is to recommend courses to student based on their school reports. The schema of programming classes in Carleton is clear.
2. *Cofi* [10] simply implements and evaluates some recommendation algorithms. It is not recommender system or framework. However, it is written by Java language [14] and it works on various platforms. This is the strong point of Cofi.
3. *Colfi* [11] is developed by Professor Lukáš Brožovský, Charles University in Prague, Czech Republic. The software builds up a recommendation server for dating service. It is larger than Carleton and Cofi. Colfi implements and evaluates some collaborative filtering algorithms but there is no customization support of algorithms and evaluation metrics. Note that collaborative filtering (CF) algorithm and content-based filtering (CBF) algorithm are typical recommendation algorithms. The recommendation server is simple and aims to research purposes. However, the prominent aspect of Colfi is to support dating service via client-server interaction.
4. *Crab* [2] is recommendation server written by programming language Python. It is developed at Muriçoca Labs. The strong point of Crab is to build up a recommendation engine inside the server along with

algorithm evaluation mechanism. When compared with this proposed framework, Crab does not support developers to realize their solutions through three stages such as implementation, evaluation, and simulation. The architecture of Crab is not flexible and built-in algorithms are not plentiful. Most of them are SVD algorithm and nearest-neighbor algorithms.

5. *Duine* [16] is developed by Telematica Institute, Novay. This is really a solid recommendation framework. Its architecture is very powerful and flexible. The strong point of Duine is to improve the performance of recommendation engine. When compared with this proposed framework, Duine does not support developers to realize their solutions through three stages such as implementation, evaluation, and simulation. The algorithm evaluator of Duine is not standardized and its customization is not high.
6. *easyrec* [15] is developed by IntelliJ IDEA and Research Studios Austria, Forschungsgesellschaft mbH. Strong points of easyrec are convenience in use, supporting consultancy via internet, and allowing users to embed recommendation engine in a website in order to call functions of easyrec from such website. However, easyrec does not support developers to build up new algorithms. This is the drawback of easyrec.
7. *GraphLab* [4] is a multi-functional toolkit which supports collaborative filtering, clustering, computer vision, graph analysis, etc. It is sponsored by Office of Naval Research, Army Research Office, DARPA and Intel. This toolkit is very large and multi-functional. This strong point also implies its drawback. Developers who get familiar with this toolkit in some researches such as computer vision and graph analysis will intend to use it for recommendation study. However, GraphLab supports recommendation research with restriction. It only implements some collaborative filtering algorithms and it is not recommendation server.
8. *LensKit* [5] is developed by the research group GroupLen, University of Minnesota, Twin Cities, USA. It is written by programming language Java and so it works on various platforms. The strong point of LensKit is to support developers to construct and evaluate recommendation algorithms very well. The evaluation mechanism is very sophisticated. However, LensKit does not provide developers a simulator or a server that helps developers to test their solutions in client-server environment. Although the schema of programming class library is fragmentary, LenKits takes advantages of the development environment Maven. In general, LenKits is a very good recommendation framework.
9. *Mahout* [12] is developed by Apache Software Foundation. It is a multi-functional toolkit which supports data mining and machine learning, in which some recommendation algorithms like nearest-neighbor algorithms are implemented. Using algorithms built in Mahout is very easy. Mahout aims to end-users instead of developers. Its strong point and drawback are very similar to the strong point and drawback of GraphLab. Mahout is essentially a multi-functional toolkit and so it does not focus on recommendation system. If you intend to develop a data mining or machine learning software, you should use Mahout. If you want to focus on recommendation system, you should use the proposed framework.
10. *MyMedia* [13] is a software that recommends customers media products such as movies and pictures. The preeminent feature of MyMedia is to focus on multimedia entertainment data when implementing social network mining algorithms, recommendation algorithms, and personalization algorithms. MyMedia is a very powerful multimedia recommendation framework which aims to end-users such as multimedia entertainment companies. However, MyMedia does not support specialized mechanism of algorithm evaluation based on pre-defined metrics. MyMedia, written by the modern programming language C#, is developed by EU Framework 7 Programme Networked Media Initiative together with partners: EMIC, BT, the BBC, Technical University of Eindhoven, University of Hildesheim, Microgenesis and Novay.
11. *MyMediaLite* [6] is a small programming library which implements and evaluates recommendation algorithms. MyMediaLite is light-weight software but it implements many recommendation algorithms and evaluation metrics. Its architecture is clear. These are strong points of MyMediaLite. However, MyMediaLite does not build up recommendation server. There is no customization support of evaluation metrics. These are drawbacks of MyMediaLite. MyMediaLite is developed by developers Zeno Gantner, Steffen Rendle, Lucas Drumond, and Christoph Freudenthaler at University of Hildesheim.
12. *recommenderlab* [8] is developed by developer Michael Hahsler and sponsored by NSF Industry/University Cooperative Research Center for Net-Centric Software & Systems. The recommenderlab is statistical extension package of R platform, which aims to build up a recommendation infrastructure based on R platform. The preeminent feature of recommenderlab is to take advantages of excellent data-processing function built in R platform. Ability to evaluate and compare algorithms is very good. However, recommenderlab does not build up recommendation server because it is dependent on R platform. The recommenderlab is suitable to algorithm evaluation in short time and scientific researches on recommendation algorithms.
13. *SVDFeature* [3], written by programming language C++, is developed by developers Tianqi Chen, Weinan

Zhang, Qiuxia Lu, Kailong Chen, Zhao Zheng, Yong Yu. SVD is a collaborative filtering algorithm which processes huge matrix very effectively in recommendation task. SVDFeature focuses on implementing SVD algorithm by the best way. Although SVDFeature is not a recommendation server, it can process huge matrix data and speed up SVD algorithm. This is the strongest point of SVDFeature.

14. *Vogoo* [17] implements and deploys recommendation algorithm on webpage written by web programming language PHP. It is very fast and convenient for developers to build up e-commercial website that supports recommendation function. Although Vogoo is simple and not a recommendation server, the strongest point of Vogoo is that its library is small and neat. If fast development has top-most priority, Vogoo is the best choice.

After surveying 14 other typical products, my product is the unique and most optimal if the function to support scientists and software developers through 3 stages such as algorithm implementation, quality assessment and experiment is considered most. Moreover the architecture of product is very flexible and highly customizable. Evaluation metrics to qualify algorithms are standardized according to pre-defined templates so that it is possible for software developers to modify existing metrics and add new metrics.

3. Description of Product

The product is computer software which has three main modules such as *Algorithm*, *Evaluator* and *Recommender*. These modules correspond with solution stages such as base stage, evaluation stage and simulation stage. Figure 1 depicts the general architecture of product. As seen in figure 1, the product is constituted of following modules:

- *Algorithm*, *Evaluator* and *Recommender* are main modules. *Algorithm*, the most important module, defines and implements abstract model of recommendation algorithms. *Algorithm* defines specifications which user-defined algorithms follow. It is possible to state that *Algorithm* is the infrastructure for other modules. *Evaluator* is responsible for evaluating algorithms according to built-in evaluation metrics. *Evaluator* also manages these built-in metrics. *Recommender* is the simulation environment which helps users to test feasibility of their algorithms in real-time applications. Thus, *Recommender* is a real recommendation server. Figures 2 and 3 depict the general sub-architectures of *Evaluator* and *Recommender*, respectively.
- *Plug-in manager*, an auxiliary module, is responsible for discovering and managing registered recommendation algorithms.
- *Listener* and *balancer*, which are auxiliary modules, are communication means between users/applications and recommendation service.
- *Parser* and *modeler*, which are auxiliary modules, are responsible for processing raw data. Raw data are read and modeled as *Dataset* by parser and modeler. *Evaluator* module evaluated algorithms based on such *Dataset*. *KBase*, an abbreviation of *knowledge base*, is the high-level abstract model of *Dataset*. For example, if recommendation algorithm mines purchase pattern of online customers from *Dataset*, such pattern is represented by *KBase*.

The general sub-architecture of Evaluator shown in figure 2 implies the evaluation process including following steps:

1. Developer implements a recommendation algorithm *A* based on specifications defined by *Algorithm* module.
2. Developer plugs algorithm *A* into *Plug-in manager*.
3. User requires to evaluate algorithm *A* by calling *Evaluator*.
4. *Evaluator* discovers algorithm *A* via *Plug-in manager*. Consequently, *Evaluator* loads and feeds *Dataset* or *KBase* to algorithm *A*. If *KBase* does not exist yet, algorithm *A* will create its own *KBase*.
5. *Evaluator* executes and evaluates algorithm *A* according to built-in metrics. Note that these metrics are managed by both metrics system and *Plug-in manager*. In client-server environment, *Evaluator* executes remotely algorithm *A* by calling *Recommender* module where algorithm *A* is deployed. This is the most important step which is the core of evaluation process.
6. *Evaluator* sends evaluation results to user with note that these results are formatted according to evaluation metrics aforementioned in step 5.

Please see the section "A Use Case of Proposed Framework" to comprehend these evaluation steps.

The general sub-architecture of *Recommender* – a recommendation server shown in figure 3 includes five layers such as interface layer, service layer, share memory layer, transaction layer, and data layer. These layers is described in bottom-up order.

Data layer is responsible for manipulating rating data organized into two formats:

- Low-level format is structured as rating matrix whose each row consists of user ratings on items. Another low-level format is Dataset which consists of rating matrix and other information such as user profile, item profile and contextual information. Dataset can be considered as high or intermediate format when it is modeled or implemented as complex and independent entity. Dataset is the most popular format.
- High-level format also stores user ratings as low-level format; besides, it has internal inference mechanism which allows us to deduce new knowledge such as user interests and user purchasing pattern from raw data like rating matrix, user profile and item profile. High-level format data is called knowledge base or KBase in short. Knowledge base is less popular than Dataset because it is only used by recommendation algorithms while Dataset is exploited widely.

Because data layer processes directly read and write so-called data operators, upper layers needs invoking data layer to access rating data. That data operators are transparent to upper layers provides ability to modify, add and remove components inside architecture. Data layer also supports checkpoint mechanism; whenever data is crashed, data layer will perform recovery tasks based on checkpoints so as to ensure data integrity. Note, checkpoint is the time point at which data is committed to be consistent. The current version of the product does not support recovery tasks yet. Process unit of this layer, namely read or write operator, is atomic unit over whole system. Data layer interacts directly with transaction layer via receiving and processing data operators request from transaction layer.

Transaction layer is responsible for managing concurrence data access. When many clients issue concurrently recommendation requests relating to a huge of data operators, a group of data operators in the same request is packed as an operator bunch considered as a transaction; thus, there are many transactions. In other words, transaction layer splits requests into data operators, which in turn groups data operators into transactions. Transaction is process unit of this layer. Transaction layer regulates transactions so as to ensure data consistency before sending data operators request down to data layer. Transaction layer connects directly to data layer and connects to service layer via storage service.

Share memory layer is responsible for creating snapshot and scanner according to requirement of storage service. Snapshot or scanner is defined as an image of piece of *Dataset* and knowledge base (*KBase*) at certain time point. This image is stored in share memory for fast access because it takes long time to access data and knowledge stored in hard disk. The difference between snapshot and scanner that snapshot copies whole piece of data into memory while scanner is merely a pointer to such data piece. Snapshot consumes much more memory but gives faster access and is more convenient. Snapshot and scanner are read-only objects because they provide only read operator. The main responsibility of this layer is to create snapshots and scanners and to discard them whenever they are no longer used. Recommendation service and storage service in service layer can retrieve information of *Dataset* or knowledge base by accessing directly to snapshot or scanner instead of interacting with transaction layer. Hence, the ultimate goal of share memory layer is to accelerate the speed of information retrieval.

Service layer is the heart of architecture when it realized two goals of recommendation server: giving the list of recommended items in accordance with client request and supporting users to retrieve and update rating database. Two these goals are implemented by two respective services: recommender service and storage service. Such services are main components of service layer. Recommender service receives request in the interchangeable format such as JSON format from upper layer – interface layer and analyze this request in order to understand its content such as who requires recommendation and what her / his profile is. After that recommender service applies an effective strategy into producing a list of favorite items which are sent back to interface layer in the same interchangeable format like JSON. Recommendation strategy is defined as the co-ordination of recommendation algorithms such as collaborative filtering and content-based filtering in accordance with the coherent process so as to achieve a best result of recommendation. In simplest form, strategy identifies to a recommendation algorithm. Recommender service is the most complex service because it implements both algorithms and strategies and applies these strategies in accordance of concrete situation. Recommender service is the core of aforementioned *Recommender* module shown in figure 1. Storage service is simpler when it has two responsibilities:

- Retrieving and updating directly *Dataset* and knowledge base by sending access request to transaction layer and receive results returned.
- Requiring share memory layer to create snapshot or scanner.

Because recommendation algorithms execute on memory and recommender service cannot access *Dataset* and knowledge base, recommender service will require snapshot or scanner from storage service. Storage service, in succession, requests share memory layer to create snapshot or scanner and receives back a reference to such snapshot or scanner. Such reference is used by recommender service.

Interface layer interacts with both clients (users and application) and service layer. It is the intermediate layer having two responsibilities:

- For clients, it receives request from users and sends back response to them.
- For service layer, it parses and forwards user request to service layer and receives back result.

There are two kinds of client request corresponding to two goals of recommendation server:

- Recommendation request is that users prefer to get favorite items.
- Access request is that users required to retrieve or update *Dataset* and knowledge base.

User-specified request is parsed into interchangeable format like JSON [18] because it is difficult for server to understand user-specified request in plain text format. *Interpreter*, the component of interface layer, does parsing function. When users type request as text, interpreter will parses such text into JSON object which in turn sends to service layer. The result, for example: a list of favorite items, is returned to interpreter in form of JSON object and thus, interpreter translates such JSON result into text result easy to be understood by users.

Because server supports many clients, it is more effective if deploying server on different platforms. It means that we can distribute service layer and interface layer in different sites. Site can be a personal computer, mainframe, etc. There are many scenarios of distribution, for example, many sites for service layer and one site for interface layer. Interface layer has another component – *listener* component which is responsible for supporting distributed deployment. Listener which has load balancing function is called *balancer*. For example, service layer is deployed on three sites and balancer is deployed on one site; whenever balancer receives user request, it looks up service sites and choose the site whose recommender service is least busy to require such recommender service to perform recommendation task. Load balancing improves system performance and supports a huge of clients. Note that it is possible for the case that balancer or listener is deployed on more than one site.

The popular recommendation scenario includes five following steps in top-down order:

1. User (or client application) specifies her / his request in text format. Typical client application is the *Evaluator* module shown in figure 2. *Interpreter* component in interface layer parses such text into JSON format request. *Listener* component in interface layer sends JSON format request to *service layer*. In distributed environment, *balancer* is responsible for choosing optimal service layer site to send JSON request.
2. *Service layer* receives JSON request from interface layer. There are two occasions:
   a. Request is to get favorite items. In this case, request is passed to recommender service. Recommender service applies appropriate strategy into producing a list of favorite items. If snapshot or scanner necessary to recommendation algorithms is not available in *share memory layer*, recommender service requires storage service to create snapshot or scanner. After that, the list of favorite items is sent back to interface layer as *JSON format result*.
   b. Request is to retrieve or update data such as querying item profile, querying average rating on specified item, rating an item, etc. In this case, request is passed to storage service. If request is to update data then an *update request* is sent to *transaction layer*. If request is to retrieve information then *storage service* looks up *share memory layer* to find out appropriate snapshot or scanner. If such snapshot (or scanner) does not exists or not contains requisite information then a *retrieval request* is sent to *transaction layer*; otherwise, in found case, requisite information is extracted from found snapshot (or scanner) and sent back to interface layer as *JSON format result*.
3. *Transaction layer* analyzes *update requests* and *retrieval requests* from *service layer* and parses them into transactions. Each transaction is a bunch of read and write operations. All low-level operations are harmonized in terms of concurrency requirement and sent to *data layer* later. Some access concurrency algorithms can be used according to pre-defined isolation level.
4. *Data layer* processes read and write operations and sends back *raw result* to transaction layer. Raw result is the piece of information stored in *Dataset* and knowledge base or the output variable indicating whether or not update (write) request is processed successfully. Transaction layer collects and sends back the raw result to service layer. Service layer translates raw result into *JSON format result* and sends such translated result to interface layer in succession.
5. The *interpreter* component in interface layer receives and translates *JSON format result* into text format result easily understandable for users.

The separated multilayer architecture of *Recommender* allows it to work effectively and stably with high customization; especially, its use case in co-operation with *Evaluator* is very simple. Please see the section "A Use Case of Proposed Framework" for comprehending how to use *Recommender* and *Evaluator*.

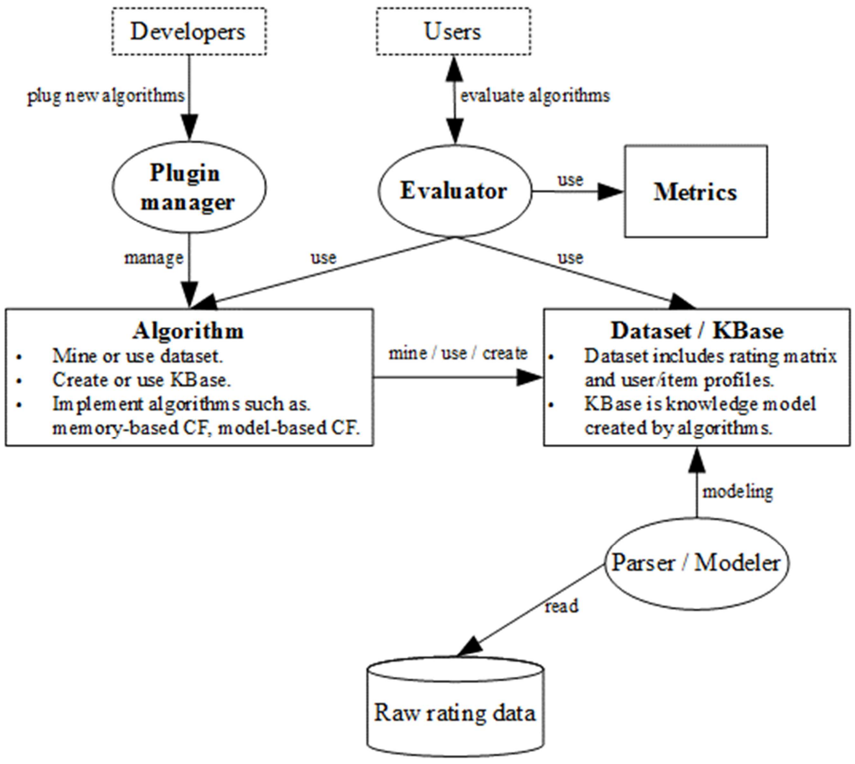


Fig. 2. *Architecture of Evaluator.*

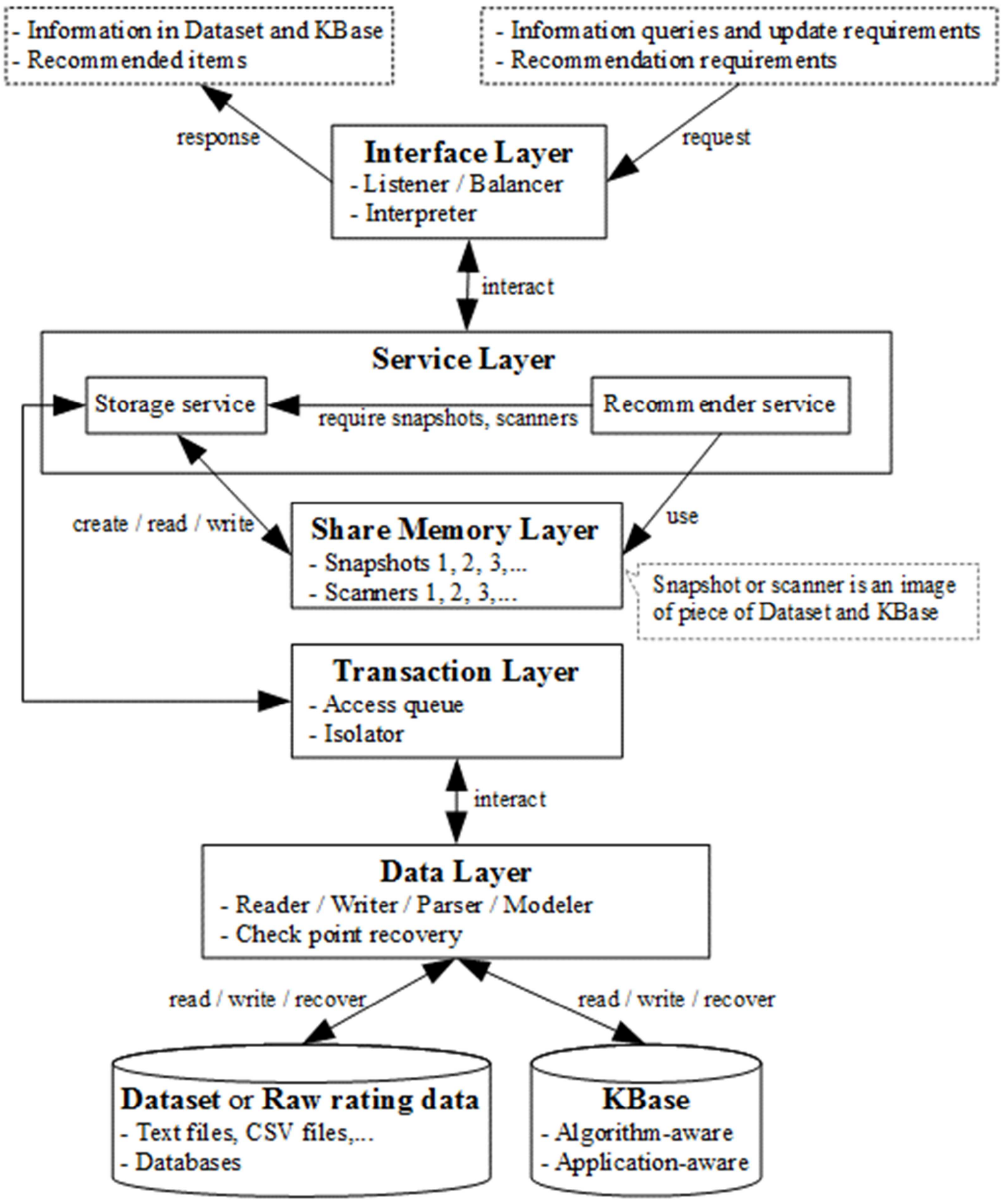


Fig. 3. *Architecture of Recommender (recommendation server).*

The general architecture of the product shown in figure 1 is decomposed into 9 packages as follows:

1. *Data* package standardizes and models data in abstract level. *Dataset* and *KBase* are built in *Data* package.
2. *Parser* package analyzes and processes data.
3. *Algorithm* package models recommendation algorithm in abstract level. *Algorithm* package supports mainly *Algorithm* module.
4. *Evaluate* package implements built-in evaluation mechanism of the framework. It also establishes common evaluation metrics. *Evaluate* package supports mainly *Evaluator* module.
5. *Client* package, *Server* package and *Listener* package provides *Recommender* module (recommendation server) in client-server network.
6. *Logistic* package provides computational and mathematic utilities.
7. *Plug-in* package manages algorithms and evaluation metrics. It supports mainly *Plug-in manager* module.

In general, three main modules *Algorithm*, *Evaluator* and *Recommender* are constituted of such 9 packages. Figure 4 depicts such nine packages of product.

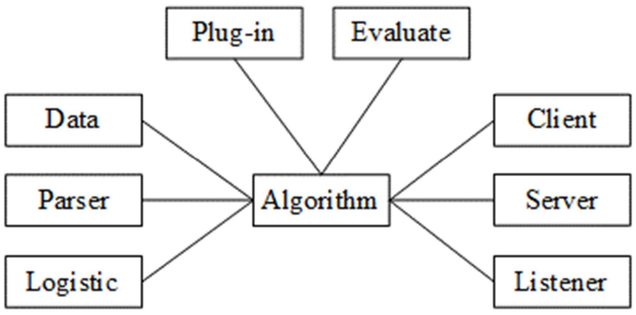
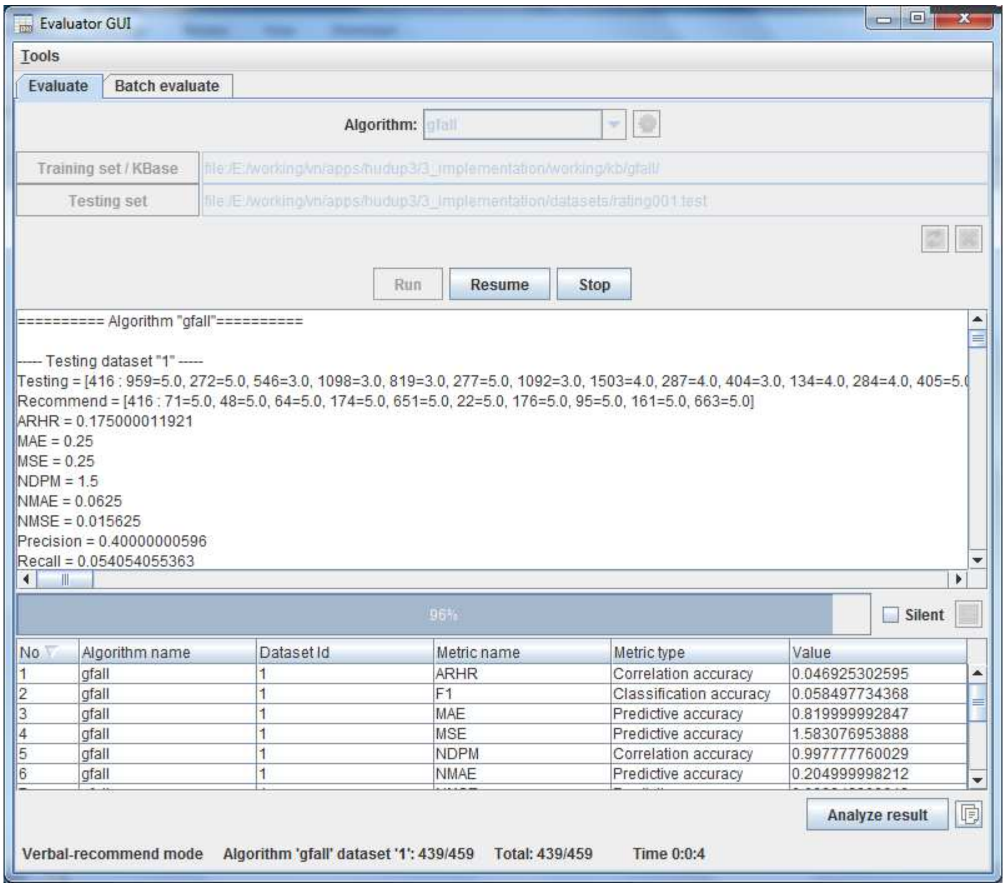


Fig. 4. *Nine packages of product.*

Each package includes many software classes constituting internal class diagrams. Especially, the *Algorithm* package provides two optimized algorithms such as collaborative filtering algorithm based on mining frequent itemsets and collaborative filtering algorithm based on Bayesian network inference.

The product helps you to build up a recommendation algorithm fast and easily. Moreover, it is very convenient for you to assess the quality and feasibility of your own algorithm in real-time application. Suppose you want to set up a new collaborative filtering algorithm so-called Green Fall, instead of writing big software with a huge of complicated tasks such as processing data, implementing algorithm, implementing evaluation metrics, testing algorithm, creating simulation environment and etc.; what you need to do is to follow three steps below:

1. Inheriting *Recommender* class in *Algorithm* package and hence, implementing your idea in two methods *estimate* and *recommend* of this class. Please distinguish *Recommender* class from *Recommender* module.
2. Starting up the *Evaluator* module (shown in figures 1 and 2) so as to evaluate and compare Green Fall with other algorithms via pre-defined evaluation metrics.
3. Configuring the *Recommender* module (recommendation server) in order to embed Green Fall into such service. After that starting up *Recommender* so as to test the feasibility of Green Fall in real-time applications.



Algorithm: gfall

Training set / KBase: file:/E:/working/vn/apps/hudup3/3_implementation/working/kb/gfall/

Testing set: file:/E:/working/vn/apps/hudup3/3_implementation/datasets/rating001.test

```
========== Algorithm "gfall"==========

----- Testing dataset "1" -----
Testing = [416 : 959=5.0, 272=5.0, 546=3.0, 1098=3.0, 819=3.0, 277=5.0, 1092=3.0, 1503=4.0, 287=4.0, 404=3.0, 134=4.0, 284=4.0, 405=5.(
Recommend = [416 : 71=5.0, 48=5.0, 64=5.0, 174=5.0, 651=5.0, 22=5.0, 176=5.0, 95=5.0, 161=5.0, 663=5.0]
ARHR = 0.175000011921
MAE = 0.25
MSE = 0.25
NDPM = 1.5
NMAE = 0.0625
NMSE = 0.015625
Precision = 0.40000000596
Recall = 0.054054055363
```

No	Algorithm name	Dataset Id	Metric name	Metric type	Value
1	gfall	1	ARHR	Correlation accuracy	0.046925302595
2	gfall	1	F1	Classification accuracy	0.058497734368
3	gfall	1	MAE	Predictive accuracy	0.819999992847
4	gfall	1	MSE	Predictive accuracy	1.583076953888
5	gfall	1	NDPM	Correlation accuracy	0.997777760029
6	gfall	1	NMAE	Predictive accuracy	0.204999998212

Verbal-recommend mode   Algorithm 'gfall' dataset '1': 439/459   Total: 439/459   Time 0:0:4

Fig. 5. *Screen shot of module Evaluator. The product home link is http://www.locnguyen.net/st/products/hudup.*

Operations in such three steps are very simple; there are mainly configurations via software graphic user interface (GUI), except that you require setting up your idea by programming code lines in step 1. Because algorithm model is designed and implemented very strictly, what you program is encapsulated in two methods *estimate* and *recommend* of *Recommender* class. The average time cost to build up and test an algorithm is around 2 years but it remains 1 week for you to realize your idea if you use my product. It means that the algorithm development cost decreased very much and so it only takes 1% origin expenditure. It is really exciting work. Figure 5 shows a screen shot of module *Evaluator*.

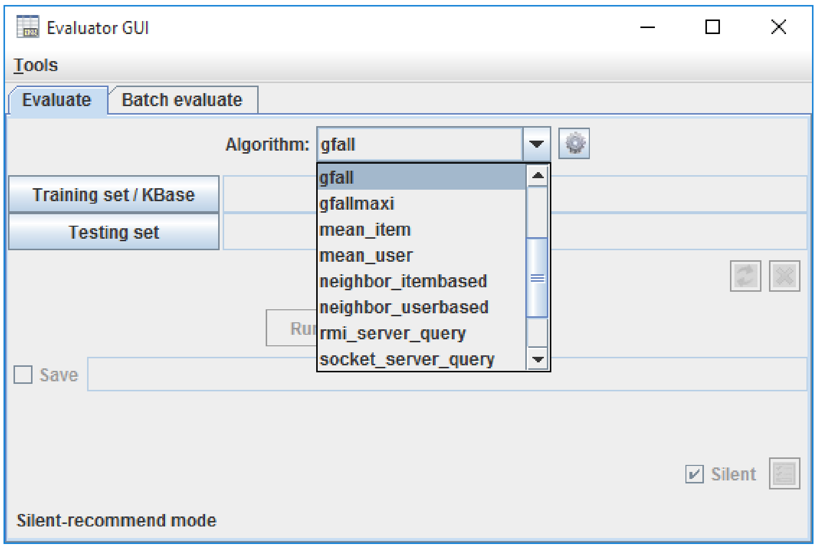
4. A Use Case of Proposed Framework

Recall that you want to set up Green Fall algorithm – a collaborative filtering algorithm based on mining frequent itemsets. You save a lot of efforts and resources when taking advantages of the proposed framework.

Firstly, you create the Java project named Green Fall and import the core Java library package "*hudup-core.jar*" into Green Fall project. You create the *GreenFall* class inheriting from *Recommender* class. This is base stage proposed by the

Hudup framework. Note that Hudup is written by the object-oriented language Java [14]. You implement your idea in two methods *estimate* and *recommend* of *GreenFall* class as follows:

```
public class GreenFall extends Recommender {
public String getName() {
return "gfall";
}
public RatingVector estimate(RecommendParam param,
Set<Integer> queryIds) {
  //Your code here…
  }
  public RatingVector recommend(RecommendParam param,
int maxRecommend) {
  //Your code here…
  }
  }
```

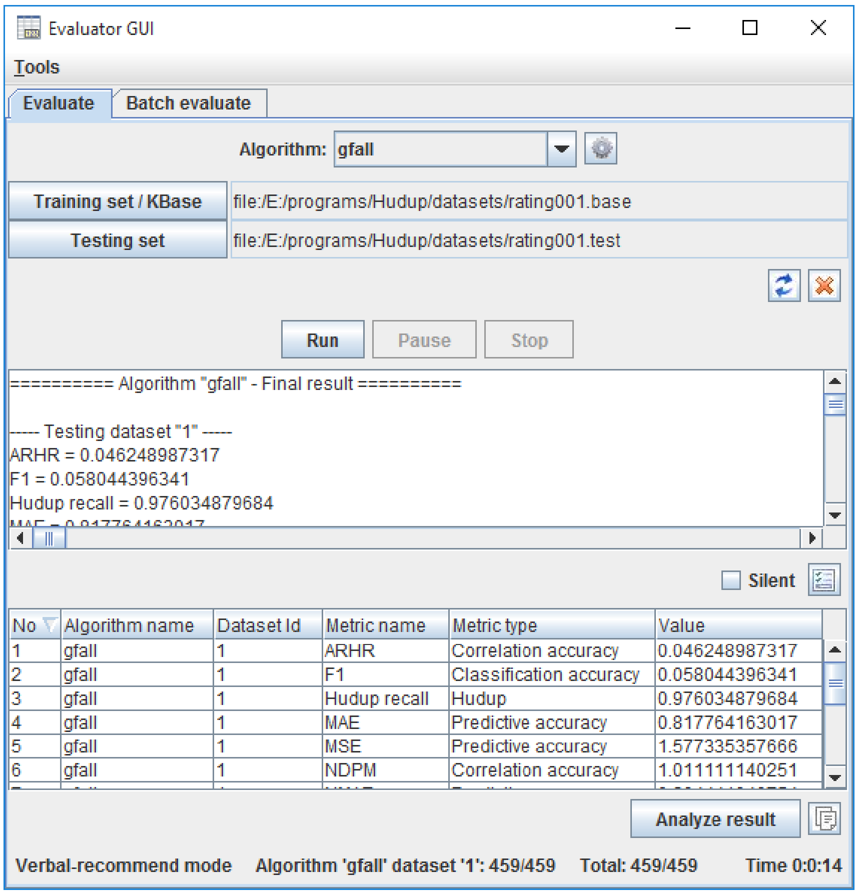


Algorithm: gfall

gfall
gfallmaxi
mean_item
mean_user
neighbor_itembased
neighbor_userbased
rmi_server_query
socket_server_query

Silent-recommend mode

Fig. 6. *Evaluator discovers the Green Fall algorithm.*



Algorithm: gfall

Training set / KBase: file:/E:/programs/Hudup/datasets/rating001.base

Testing set: file:/E:/programs/Hudup/datasets/rating001.test

```
========== Algorithm "gfall" - Final result ==========

----- Testing dataset "1" -----
ARHR = 0.046248987317
F1 = 0.058044396341
Hudup recall = 0.976034879684
```

No	Algorithm name	Dataset Id	Metric name	Metric type	Value
1	gfall	1	ARHR	Correlation accuracy	0.046248987317
2	gfall	1	F1	Classification accuracy	0.058044396341
3	gfall	1	Hudup recall	Hudup	0.976034879684
4	gfall	1	MAE	Predictive accuracy	0.817764163017
5	gfall	1	MSE	Predictive accuracy	1.577335357666
6	gfall	1	NDPM	Correlation accuracy	1.011111140251

Verbal-recommend mode Algorithm 'gfall' dataset '1': 459/459 Total: 459/459 Time 0:0:14

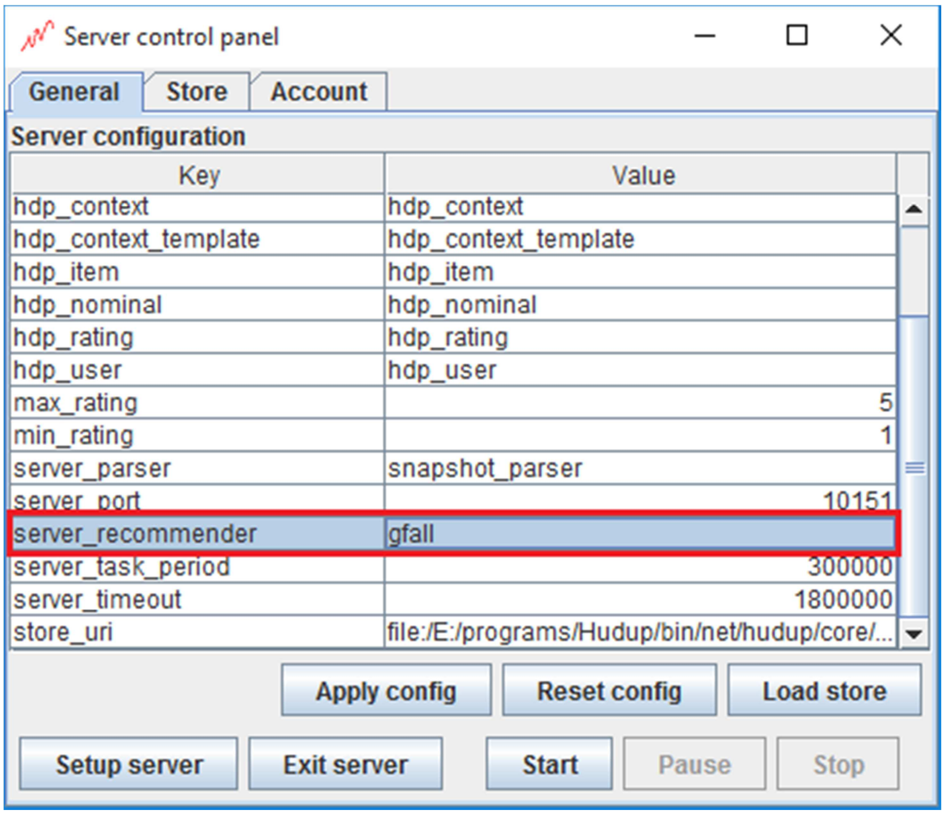
Fig. 7. *Evaluator lists evaluation result of Green Fall algorithm.*

The *estimate* method returns estimated rating values of given items and the *recommend* method returns a list of recommended items. Note that the short name of Green Fall algorithm is *gfall* as a returned value of the *getName* method. All things you do manually are to implement these two methods. Following tasks are configured via the user graphic interface (GUI) of the proposed framework. You compile the Green Fall project and compress it into the Java package *gfall.jar*. Subsequently, you put the package *gfall.jar* into library directory of the proposed framework.

Secondly, you open the *Evaluator*, as shown in figure 6. This is evaluation stage proposed by Hudup framework.

The *Evaluator* discovers the Green Fall algorithm via the name *gfall*. Now you evaluate the Green Fall algorithm by *Evaluator*. Database *Movielens* [7] including 100,000 ratings of 943 users on 1682 movies is used for evaluation. Database is divided into 5 folders, each folder includes training set over 80% whole database and testing set over 20% whole database. Training set and testing set in the same folder are disjoint sets. You do not need to consider database because *Evaluator* processes it automatically. Moreover, the complex evaluation mechanism and metrics system built in *Evaluator* are applied into evaluating Green Fall algorithm according to pre-defined metrics. There are 7 important pre-defined metrics [9] used in this evaluation: *MAE*, *MSE*, *precision*, *recall*, *F1*, *ARHR* and *time*. Note that time metric is calculated in seconds. MAE and MSE are predictive accuracy metrics that measure how close predicted value is to rating value. The less MAE and MSE are, the high accuracy is. Precision, recall and F1 are quality metrics that measure the quality of recommendation list – how much the recommendation list reflects user's preferences. ARHR is also quality metric that indicates how well recommendation list is matched to user's rating list according to rating ordering. The large quality metric is, the better algorithm is. *Evaluator* also allows you to define custom metrics. Figure 7 shows the evaluation result of Green Fall algorithm.

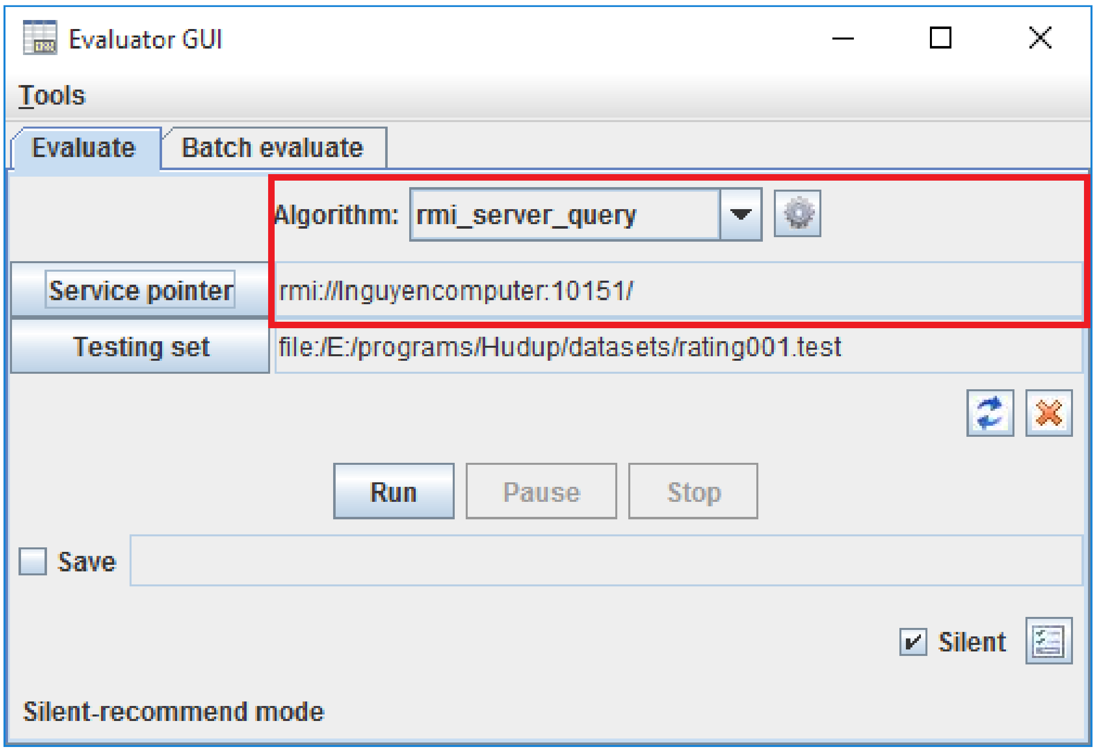
Finally, you configure *Recommender* (recommendation server) to deploy and test Green Fall algorithm in real-time application. This is simulation stage proposed by Hudup framework. Figure 8 shows up how to deploy Green Fall algorithm by *Recommender*.



Key	Value
hdp_context	hdp_context
hdp_context_template	hdp_context_template
hdp_item	hdp_item
hdp_nominal	hdp_nominal
hdp_rating	hdp_rating
hdp_user	hdp_user
max_rating	5
min_rating	1
server_parser	snapshot_parser
server_port	10151
server_recommender	gfall
server_task_period	300000
server_timeout	1800000
store_uri	file:/E:/programs/Hudup/bin/net/hudup/core/...

Fig. 8. *Recommender Service deploys Green Fall algorithm.*

Now you use the *Evaluator* to test Green Fall algorithm via client-server environment. *Evaluator* calls Green Fall algorithm from *Recommender* with note that Green Fall algorithm is now separated from *Evaluator* because it is deployed inside *Recommender*. Figure 9 indicates how to guide *Evaluator* to connect *Recommender* in order to call Green Fall algorithm remotely.



Algorithm: rmi_server_query

Service pointer: rmi://lnguyencomputer:10151/

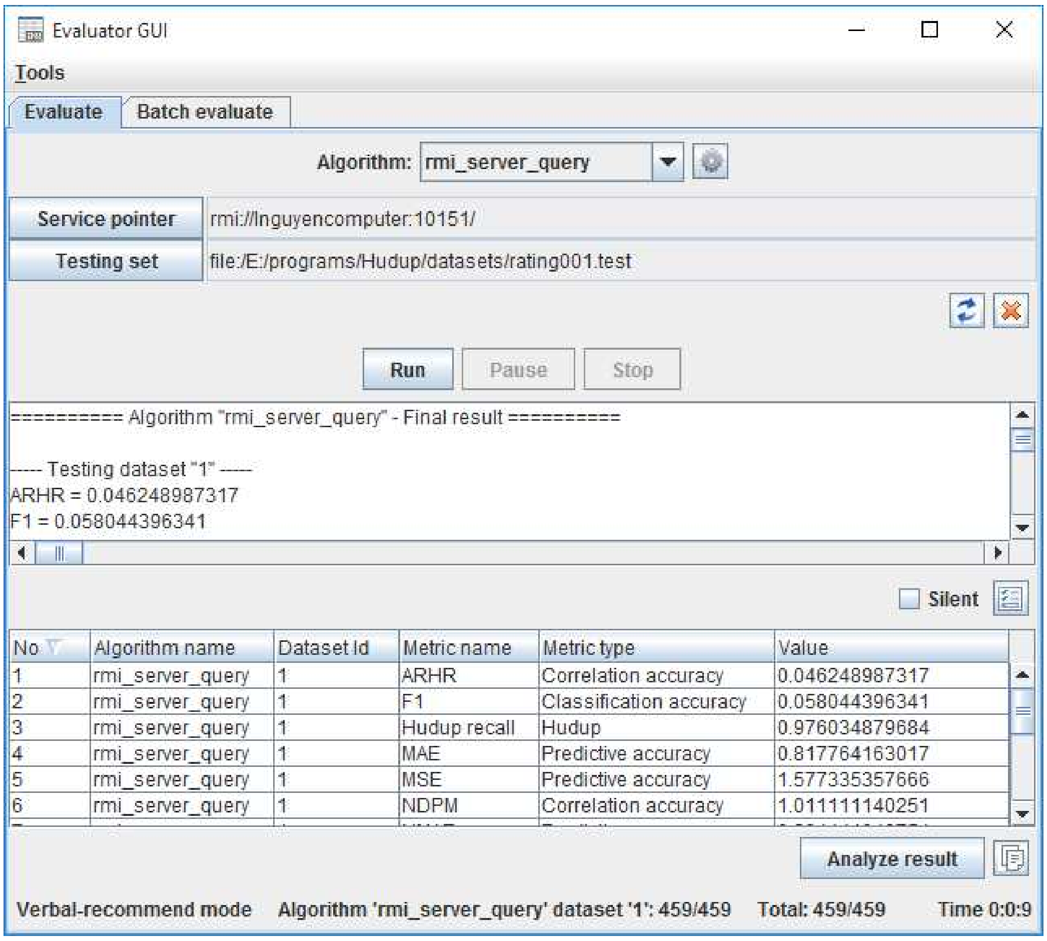
Testing set: file:/E:/programs/Hudup/datasets/rating001.test

☑ Silent

Silent-recommend mode

Fig. 9. Configure Evaluator to connect Recommender.

Figure 10 shows again the evaluation result of Green Fall algorithm from remote execution.



Algorithm: rmi_server_query

Service pointer: rmi://lnguyencomputer:10151/

Testing set: file:/E:/programs/Hudup/datasets/rating001.test

```
========== Algorithm "rmi_server_query" - Final result ==========

----- Testing dataset "1" -----
ARHR = 0.046248987317
F1 = 0.058044396341
```

No	Algorithm name	Dataset Id	Metric name	Metric type	Value
1	rmi_server_query	1	ARHR	Correlation accuracy	0.046248987317
2	rmi_server_query	1	F1	Classification accuracy	0.058044396341
3	rmi_server_query	1	Hudup recall	Hudup	0.976034879684
4	rmi_server_query	1	MAE	Predictive accuracy	0.817764163017
5	rmi_server_query	1	MSE	Predictive accuracy	1.577335357666
6	rmi_server_query	1	NDPM	Correlation accuracy	1.011111140251

Verbal-recommend mode Algorithm 'rmi_server_query' dataset '1': 459/459 Total: 459/459 Time 0:0:9

Fig. 10. Evaluation result of Green Fall algorithm from remote execution.

It is concluded that it is easy for you to implement, evaluate and deploy your solution by taking advantages of the proposed framework. Most tasks are configured via friendly GUI. Complicated operations are processed by services and you only focus on realizing your ideas. Hudup – a framework of e-commercial recommendation solutions" is the best choice for you to build up a recommendation solution.

Acknowledgements

This product is the place to acknowledge Ms. Do, Phung T. M. – University of Information Technology, Vietnam National University and Mr. Vu, Dong N. who gave me valuable comments and advices. These comments help me to improve this product.

References

[1] Brožovský, L. (2006, August). ColFi - Recommender System for a Dating Service. (V. Petříček, Ed.) Prague, Czech Republic: Charles University in Prague.

[2] Caraciolo, M., Melo, B., & Caspirro, R. (2011). Crab - Recommender systems in Python. Muriçoca Labs.

[3] Chen, T., Zhang, W., Lu, Q., Chen, K., Zheng, Z., Yu, Y., et al. (2012). SVDFeature: A Toolkit for Feature-based Collaborative Filtering. 1.2.2. China: APEX Data & Knowledge Management Lab.

[4] Dato-Team. (2013, October 15). GraphLab Create™. (C. Guestrin, Ed.) Seattle, Washington, USA: Dato, Inc.

[5] Ekstrand, M., Kluver, D., He, L., Kolb, J., Ludwig, M., & He, Y. (2013). LensKit - Open-Source Tools for Recommender Systems. Group Lens Research, University of Minnesota.

[6] Gantner, Z., Rendle, S., Drumond, L., & Freudenthaler, C. (2013). MyMediaLite Recommender System Library. 3.10. University of Hildesheim and The European Commission 7th Framework Programme.

[7] GroupLens. (1998, April 22). MovieLens datasets. (GroupLens Research Project, University of Minnesota, USA) Retrieved August 3, 2012, from GroupLens Research website: http://grouplens.org/datasets/movielens.

[8] Hahsler, M. (2014, 08 18). recommenderlab: Lab for Developing and Testing Recommender Algorithms. 0.1-5. NSF Industry/University Cooperative Research Center for Net-Centric Software & Systems.

[9] Herlocker, J. L., Konstan, J. A., Terveen, L. G., & Riedl, J. T. (2004). Evaluating Collaborative Filtering Recommender Systems. ACM Transactions on Information Systems (TOIS), 22 (1), 5-53.

[10] Lemire, D. (2003). Cofi: A Java-Based Collaborative Filtering Library. Canada: National Research Council of Canada.

[11] Lew, D., & Sowell, B. (2007). Carleton Recommender systems. (D. Musicant, Ed.) Northfield, Minnesota, USA: Carleton College.

[12] Mahout-Team. (2013). Apache Mahout™. The Apache Software Foundation.

[13] Microsoft, My Media PC, RichHanbidge, My Media Wp 2 Lead, My Media Wp 3 Lead, My Media Wp 4 Lead, et al. (2013). My Media Dynamic Personalization and Recommendation Software Framework Toolkit. MyMedia project funded through the EU Framework 7 Programme Networked Media initiative.

[14] Oracle. (n. d.). Java language. (Oracle Corporation) Retrieved December 25, 2014, from Java website: https://www.oracle.com/java.

[15] Smart-Agent-Technologies. (2013). easyrec. Research Studios Austria Forschungsgesellschaft mbH.

[16] Telematica-Instituut. (2007, June 29). Duine Framework. Telematica Instituut/Novay.

[17] Vogoo-Team, & DROUX, S. (2008). Vogoo PHP LIB. 2.2. Source Forge.

[18] ECMA, "The JSON Data Interchange Format," ECMA International, Geneva, 2013.

Loc Nguyen
Board of Directors, Sunflower Soft Company, Ho Chi Minh City, Vietnam

Loc Nguyen is a Director at Sunflower Soft Company, Vietnam. Currently, he is interested in computer science, statistics, and mathematics. He serves as reviewer and editor in a wide range of international journals. He is a volunteer of Statistics Without Borders of American Statistics Association. He finished Postdoctoral research in Computer Science at Sunflower Soft Company. He has published more than 35 papers. He is author of 2 scientific books, 11 scientific products, 1 verse story, 7 poem collections and 2 music albums. Ultimately, he is very attractive, enthusiastic and creative. His favourite statement is "Creative man is The Creator".