A Framework of E-commercial Recommendation Algorithms

Loc Nguyen[1]
[1]*Sunflower Soft Company, Ho Chi Minh, Vietnam*
ng_phloc@yahoo.com

Keywords: Recommendation Algorithm, Recommendation Server, Middleware Framework.

Abstract: Recommendation algorithm is very important to e-commercial websites when it can provide favorite products to online customers, which results out an increase in sale revenue. I propose the infrastructure for e-commercial recommendation solutions. It is a middleware framework of e-commercial recommendation software, which supports scientists and software developers to build up their own recommendation algorithms with low cost, high achievement and fast speed. This report is a full description of proposed framework, which begins with general architectures and then concentrates on programming classes. Finally, a tutorial will help readers to comprehend the framework.

1 INTRODUCTION

The product provides infrastructure for e-commercial recommendation solutions, named *Hudup*. This is a middleware framework of e-commercial recommendation software, which supports scientists and software developers to build up their own recommendation solutions. The term "recommendation solution" refers to computer algorithm that introduces online customer a list of items such as books, products, services, news papers, and fashion clothes on e-commercial websites with expectation that customer will like these recommended items. The goal of recommendation algorithm 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 excited but:

1. You cope with complicated computations when analyzing big data and there are a variety of heterogeneous models in recommendation study.
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 feasibility of your algorithm.

The innovative product Hudup supports you to solve perfectly three difficulties above 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 feasibility of your algorithm in real-time applications.

Hudup has another preeminent function which is to provide two optimized algorithms so that it is convenient for you to assess and compare different solutions. Hudup aims to help you, a scientist or software developer, to solve three core problems above. Hudup 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 (simulator), which helps you to test feasibility of your algorithm.

There are now some open source softwares similar to my product. The brief list of them is described as follows:

1. *Carleton* (Lew and Sowell, 2007) 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 implementation illustration of recommendation algorithms and it is not recommendation frame-

work. 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* (Lemire, 2003) simply implements and evaluates some recommendation algorithms. It is not recommendation framework. However, it is written by Java language (Oracle, 2014) and it works on various platforms. This is the strong point of Cofi.
3. *Colfi* (Brozovsky, 2006) is developed by Professor Lukas Brozovsky, 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) and content-based filtering (CBF) 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* (Caraciolo et al., 2011) is recommendation server written by programming language Python. It is developed at Muricoca Labs. The strong point of Crab is to build up a recommendation engine inside the server along with algorithm evaluation mechanism. When compared with the 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* (Telematica-Instituut, 2007) 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 performance of recommendation engine. When compared with the 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* (Smart-Agent-Technologies, 2013) 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 into 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* (Dato-Team, 2013) 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. GraphLab is very large and multi-functional. This strong point also implies its drawback. Developers who get familiar with GraphLab 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* (Ekstrand et al., 2013) is developed by 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* (Mahout-Team, 2013) 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* (Microsoft et al., 2013) is a software that recommends customers media products such as movies and pictures. The preeminent feature of MyMedia is to focus on multimedia en-

tertainment 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 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* (Gantner et al., 2013) 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* (Hahsler, 2014) 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* (Chen et al., 2012), 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* (Vogoo-Team and DROUX, 2008) 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 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. The trial version of Hudup product is available at http://www.locnguyen.net/st/products/hudup. Architectures relevant to Hudup framework are described in section 2.

2 GENERAL DESCRIPTION

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 called *simulator* 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.
- *Plugin manager*, an auxiliary module, is responsible for discovering and managing registered rec-

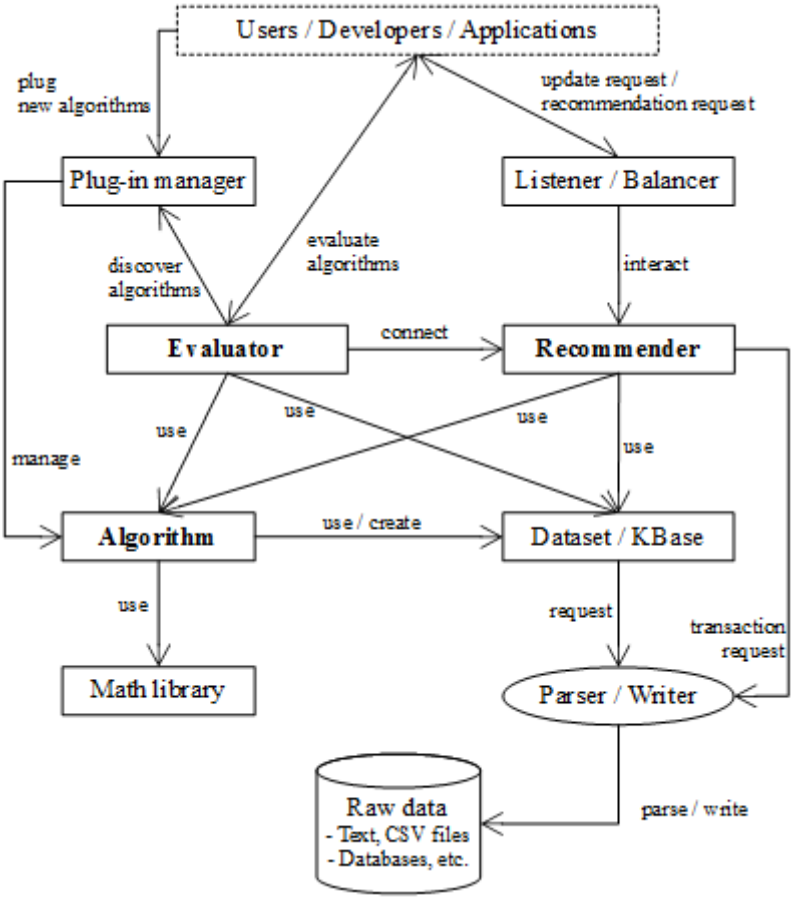

Figure 1: General architecture of Hudup

ommendation algorithms.

- *Parser*, which is an auxiliary module, is responsible for processing raw data. Raw data are read and modeled as *Dataset* by parser. *Evaluator* module evaluates 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 scenario including following steps:

1. Developer implements a recommendation algorithm *A* based on specifications defined by *Algorithm* module.
2. Developer plugs algorithm *A* into *Plugin manager*.
3. Scientist requires to evaluate algorithm *A* by calling *Evaluator*.
4. *Evaluator* discovers algorithm *A* via *Plugin 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. These metrics are managed by both metrics system and *Plugin 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 scientist with note that these results are formatted according to evaluation metrics aforementioned in step 5.

Please see subsection 3.3 and section 4 to comprehend the evaluation scenario.

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 are described in bottom-up order.

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

- Low-level format is structured as rating matrix whose each row consists of user ratings on items, often called raw data. Another low-level format is *Dataset* which consists of rating matrix and other information such as user profile, item profile and

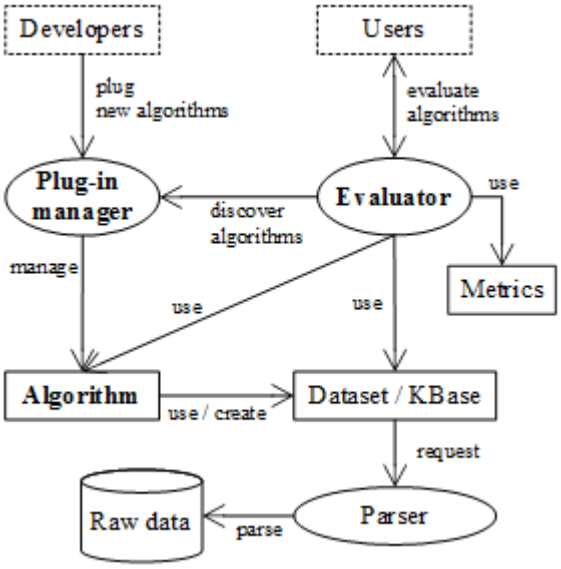

Figure 2: Sub-architecture of Evaluator

contextual information. *Dataset* can be considered as intermediate format when it is modeled as complex and independent entity. *Dataset* is the most popular format.

- High-level format contains fine-grained information and knowledge extracted from raw data and *Dataset*, for example, user interests and user purchasing pattern; besides, it may have internal inference mechanism which allows us to deduce new knowledge. High-level format structure is called *knowledge base* or *KBase* in short. *KBase* is less popular than *Dataset* because it is only used by recommendation algorithms while *Dataset* is exploited widely.

Within context of *Recommender* module, *Dataset* and *KBase* are data formats and here they do not refer to programming classes and interfaces. Because data layer processes directly read and write data operators, upper layers needs invoking data layer to access database. 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 operator requests from transaction layer.

Transaction layer is responsible for managing

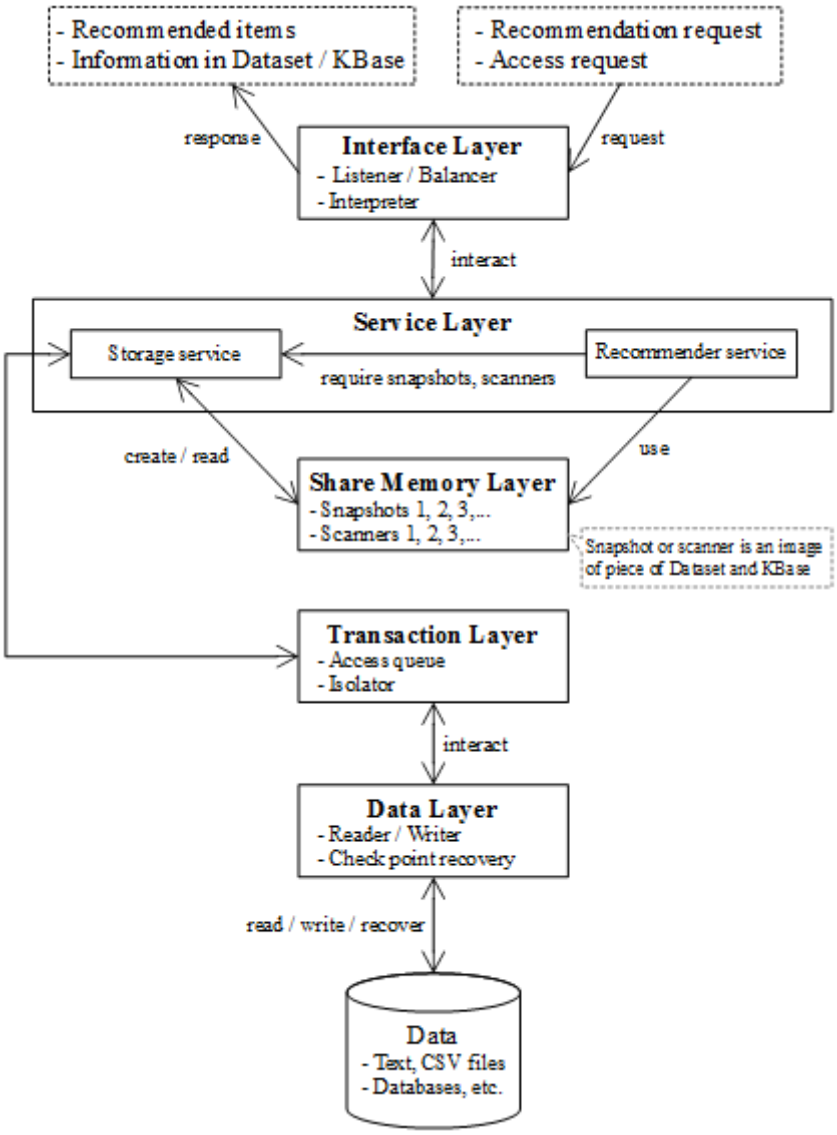

Figure 3: Sub-architecture of Recommender

concurrence data accesses. When many clients issue concurrently 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. Snapshot and scanner are read-only objects because they provide only read operator. The main responsibility of share memory 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* and *KBase* 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 realizes two goals of recommendation server: giving the list of recommended items in accordance with client request and supporting users to retrieve and update database. Such two goals are implemented by two respective services: *recommender service* and *storage service*. These 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 analyzes this request in order to understand its content (user ratings and user profile). 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 coordination of recommendation algorithms such as collaborative filtering and content-based filtering in accordance with coherent process so as to achieve the best result of recommendation. In simplest form, strategy is identified with a recommendation algorithm. Recommender service is the most complex service because it implements both algorithms and strategies and applies these strategies in accordance with 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 *KBase* by sending access request to transaction layer and receiving results returned.
- Requiring share memory layer to create snapshot or scanner.

Because recommendation algorithms execute on memory and recommender service cannot access *Dataset* and *KBase*, 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 require to retrieve or update *Dataset* and *KBase*.

User-specified request is parsed into interchangeable format like JSON (ECMA, 2013) 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 specify request as text, interpreter will parses such text into JSON object which in turn is sended 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, and modifying user profile. 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 nor 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 *KBase*. Raw result can be 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* module allows it to work effectively and stably with high customization; especially, its use case in co-operation with *Evaluator* module is very simple. Please see the section 4 for comprehending how to use *Recommender* and *Evaluator*.

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

1. *Data* package is responsible for standardizing and modeling data in abstract level. *Dataset* and *KBase* are built in *Data* package.
2. *Parser* package is responsible for analyzing and processing data.
3. *Algorithm* package is responsible for modeling recommendation algorithm in abstract level. *Algorithm* package supports mainly *Algorithm* module.
4. *Evaluation* package implements built-in evaluation mechanism of the framework. It also establishes common evaluation metrics. *Evaluation* package supports mainly *Evaluator* module.
5. *Client* package, *Server* package and *Listener* package provide *Recommender* module (recommendation server) in client-server network with essential support of *Algorithm* package.
6. *Logistic* package provides computational and mathematic utilities.
7. *Plugin* package manages algorithms and evaluation metrics. It supports mainly *Plugin manager* module.

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

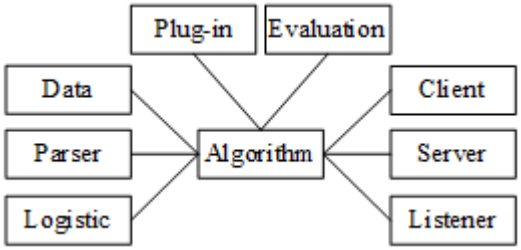

Figure 4: Nine packages of Hudup

includes many software classes constituting internal class diagrams. Section 3 will focus on these classes. 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 quality and feasibility of your own algorithm in real-time application. Suppose you want to set up a new collaborative filtering algorithm 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, and creating simulation environment; 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 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.

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% original expenditure. It is really exciting work.

3 CORE CLASSES AND INTERFACES

As aforementioned, Hudup is constituted of three main modules *Algorithm*, *Evaluator*, and *Recommender* which, in turn, are decomposed into 9 packages. Each package includes many programming components but there is the limited number of core classes and interfaces on which this section focuses. Although Hudup is now implemented by Java language, it is convenient to describe classes and interfaces by UML language (Duong, 2008). As a convention, all classes, interfaces, methods and properties complying with UML standard are written in italic font. Both class and interface are drawn as rectangles.

Class and interface are denoted as *Class*, *Interface*, *Package*::*Class*, *Package*::*Interface*. If package is ignored, it is known in context. Attribute is denoted as *attribute* or *Class*::*attribute*.

Method is denoted as *method*(), *method*(*parameters*), *Class*::*method*(*parameters*), or *Class*::*method*(*parameters*):*returned*. If class is ignored, it is known in context. For a short description, method's parameters "*parameters*" and returned value "*returned*" can be ignored. For example, notations:

- *Recommender*::*recommend*(*RecommendParam*, *int*):*RatingVector*
- *recommend*(*RecommendParam*, *int*):*RatingVector*
- *recommend*(*RecommendParam*, *int*)
- *recommend*()

indicate the same method *recommend*() of *Recommender* class. In fact, the method has two input parameters represented by *RecommendParam* class and integer number. It returns the value represented by *RatingVector* class.

UML class diagram relationships commonly used in the report are dependency, association, aggregation, composition, generalization (inheritance, derivation), and realization (implementation) as shown in figure 5.

When classes and interfaces are implemented, they are called objects or components. Some relevant classes and interfaces are grouped into a diagram and one package may own many possible diagrams. Commonly, classes and interfaces are identified with objects, subject, definitions, etc. on which they model.

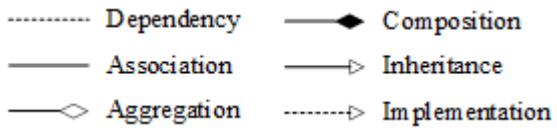

Figure 5: Common UML relationships

For example, *Recommender* (class) refers to recommendation algorithm. In general, core classes and interfaces of Hudup will be described according to their packages.

3.1 Algorithm package

The most important class of *Algorithm* package is *Recommender* class. It is abstract model of all recommendation algorithms. *Recommender* class has two most important methods which researchers must realize according to their ideas and goals, as follows:

- Method *estimate*(*RecommendParam*, *int*[]):*RatingVector* whose input parameters are a recommendation parameter and a set of item identifiers. Its output result is a set of predictive or estimated rating values of items specified by the second input parameter.
- Method *recommend*(*RecommendParam*, *int*):*RatingVector* whose input parameters are a recommendation parameter and a user identifier (user id). Its output result is a list of recommended items which is provided to the user specified by the user id.

The first input parameter of both methods represented by *RecommendParam* class includes user profile represented by *Profile* interface, user rating vector represented by *RatingVector* class, context information, etc. The output result of both methods is represented by *RatingVector* in which rating values of items are predicted (estimated). Please see subsection 3.2 for more details of *RatingVector*. Some algorithms calls *estimate*() method inside *recommend*() method; in other words, *recommend*() is dependent on *estimate*() in some cases. *Recommender* class makes recommendation task based on executing such two methods. Ideas and features of algorithm are expressed by how methods *estimate*() and *recommend*() are implemented. *Recommender* will call its *setup*(*Dataset*, *Object*[] *params*) method if it needs to prepare *Dataset* before making recommendation task.

Recommender class realizes directly *Alg* interface which represents any algorithm. *Alg* provides configuration methods such as *getConfig*() and *createDefaultConfig*() which allow programmers to pass customization settings to a given algorithm before it runs. Every algorithm has a unique name as returned value

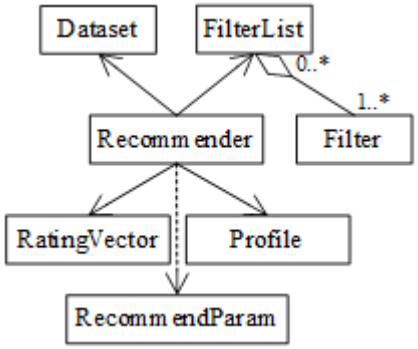

Figure 6: Recommender diagram

of *getName*() method of *Alg*. Plugin manager discovers automatically all algorithms via their names. *Alg* is the most general interface of Hudup; anything that is programmable and executable is *Alg*. As a convention, *Alg* is identified with any algorithm and *Recommender* is identified with recommendation algorithm. Moreover, *Recommender* also refers to *Recommender* module, simulator, and recommendation server aforementioned in section 2. Readers distinguish them according to context. The same notation implies that recommendation server is based on recommendation algorithm. In fact, recommendation algorithm is embedded in recommender service (see figures 3, 23).

Recommender class is executed on the dataset represented by *Dataset* which is the core interface of *Data* package. If programmer needs to do some prefiltering operations before *Recommender* class makes recommendation task, she/he can take advantages of *Recommender*::*getFilterList*() method which returns a list of filters. Each filter is represented by *Filter* interface. In general, *Recommender* class associates closely with classes and interfaces: *Dataset*, *RecommendParam*, *RatingVector*, *Profile*, *Filter*. Figure 6 shows their diagram. *Dataset*, *RatingVector*, and *Profile* belong to *Data* package, which are mentioned later.

In recommendation study, there are two common trends such as content-based filtering (CBF) and collaborative filtering (CF) and each trend has two popular approaches such as memory-based and model-based. Correspondingly, *Recommender* class is derived directly by two abstract classes *MemoryBasedRecommender* and *ModelBasedRecommender*. The *MemoryBasedRecommender* class, in turn, is inherited by two classes *MemoryBasedCBF* and *MemoryBasedCF*. The *ModelBasedRecommender* class, in turn, is inherited by two classes *ModelBasedCBF* and *ModelBasedCF*. Another class, *CompositeRecommender*, is also derived directly from *Recommender* class. *CompositeRecommender* repre-

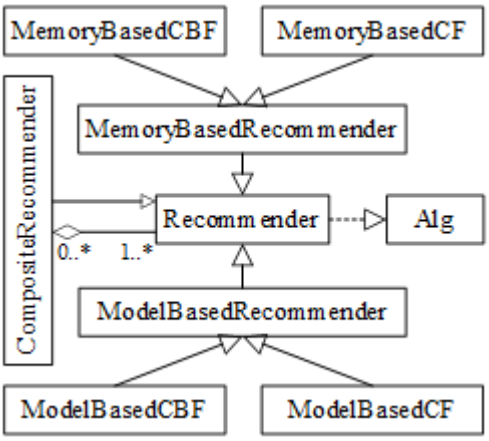

Figure 7: Recommendation algorithms

sents recommendation strategy aforementioned in section 2. *CompositeRecommender* is combination of other *Recommender* algorithms in order to produce the best list of recommended items. Figure 7 expresses inheritance relationships among these classes.

Especially, *ModelBasedRecommender* applies knowledge database represented by *KBase* interface into performing recommendation task. In other words, *KBase* provides both necessary information and inference mechanism to *ModelBasedRecommender*. Ability of inference is the unique feature of *KBase*. *ModelBasedRecommender* is responsible for creating *KBase* by calling its *createKBase*() method and so, every *ModelBasedRecommender* algorithm owns distinguishable *KBase*. For example, if *ModelBasedRecommender* algorithm uses frequent purchase pattern to make recommendation, its *KBase* contains such pattern. *ModelBasedRecommender* always takes advantages of *KBase* whereas *MemoryBasedRecommender* uses *Dataset* in execution. As aforementioned in section 2, *KBase* is the high-level format and *Dataset* is the low-level format. *KBase* is commonly created from *Dataset*. In general, *KBase* is a significant aspect of *Algorithm* package. Followings are essential methods of *KBase*:

- Method *setConfig*(*DataConfig*) is responsible for setting configurations for *KBase*. These configurations are used by other methods. The typical configuration is uniform resource identifier (URI) indicating where to store *KBase*, which is used by methods *load*() and *save*().
- Methods *load*() and *save*() are used to read/write *KBase* from/to storage system, respectively. Storage system can be files, database, etc.
- Method *learn*(*Dataset*, *Alg*) is responsible for creating *KBase* from *Dataset* which is the first input parameter. Because every *ModelBasedRecommender* algorithm owns distinguishable *KBase*, the second parameter is such algorithm. The association between *ModelBasedRecommender* and *KBase* is tight.
- Methods *clear*() and *isEmpty*() are responsible for cleaning out *KBase* and checking whether *KBase* is empty or not, respectively.

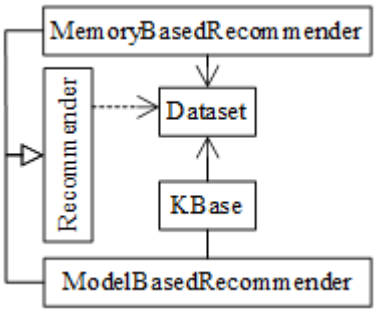

Figure 8: Relationships among Recommender, Dataset, and KBase

Methods of *ModelBasedRecommender* always using *KBase* are *setup*(), *createKBase*(), *estimate*() and *recommend*(). Especially, it is mandatory that *setup*() method of *ModelBasedRecommender* calls method *KBase*::*learn*() or *KBase*::*load*(). Figure 8 expresses relationships among *Recommender*, *Dataset*, and *KBase*. The association between *MemoryBasedRecommender* and *Dataset* indicates that all memory-based algorithms use dataset for recommendation task.

3.2 Data and Parser packages

Dataset is the core interface of *Data* package. *Dataset* is composed of rating matrix, user profiles, item profiles, and context information. Each row in rating matrix is a user rating vector which consists of rating values given to items by a concrete user. Rating vector is represented by *RatingVector* class. Within current implementation, *RatingVector* contains a set of ratings. Each rating represented by *Rating* class including three attributes as follows:

- The *rating value* that a user gives on an item. This value is represented by a real number.
- The *timestamp* identifies when such user rates on such item.
- The *context information* is represented by *Context* class, for example, the place where user makes a purchase, the persons with whom user makes a purchase.

Table 1: RatingVector class

Data::*RatingVector*
userid:*int* *ratings*:*Rating*[]
get(*int*):*Rating* *put*(*int*, *Rating*):*void* *remove*(*int*):*void* . . .

Moreover, *RatingVector* provides many methods to extract and update *Rating* (s). Table 1 lists some methods of user *RatingVector*. For example, *RatingVector*::*get*(*int*) method returns a *Rating* that user gives to an item specified by the input parameter as item identifier.

User profile is transcript of personal information: demographic information, career, etc. Item profile contains attributes of given item: name, item type, price, etc. Both user profile and item profile are represented by *Profile* class.

Dataset interface specifies a set of methods which provide easy access to rating matrix, user profiles, item profiles, context information, etc. *Dataset* is used directly by *MemoryBasedRecommender* class. *KBase* is created based on *Dataset*. *KBase* is also considered as essential model which is extracted or mined from *Dataset*. Some important methods of *Dataset* are listed below:

- Methods *getUserRating*(*int*) and *getUserProfile*(*int*) retrieve user rating vector *RatingVector* and user profile *Profile*, respectively given user identifier. Methods *getItemRating*(*int*) and *getItemProfile*(*int*) retrieve item rating vector *RatingVector* and item profile *Profile*, respectively given item identifier.
- Methods *fetchUserIds*() and *fetchItemIds*() allow us to get a set of user identifiers and a set of item identifiers, respectively.
- Method *profileOf(Context)* retrieves profile information of a specified context. Context will be mentioned later.

As aforementioned in section 2, two common implementations of *Dataset* are snapshot and scanner. Snapshot is represented by abstract class *Snapshot*, which is a piece of dataset stored in memory. Scanner is represented by abstract class *Scanner*, which is a reference to a range of dataset. It is faster to retrieve data from *Snapshot* but *Snapshot* consumes much more memory than *Scanner* does. By default implementation of *Snapshot*, rating matrix and item/user profiles are stored in hash table in which each *RatingVector* or *Profile* is identified by an integer number called key. Given hash table, snapshot access operators like reading *RatingVector* and *Profile* become faster with computational complexity $O(1)$ and so access time is instant. *Snapshot* and *Scanner* support share memory layer shown in figure 3. Figure 9 expresses diagram of *Dataset* and its relevant classes.

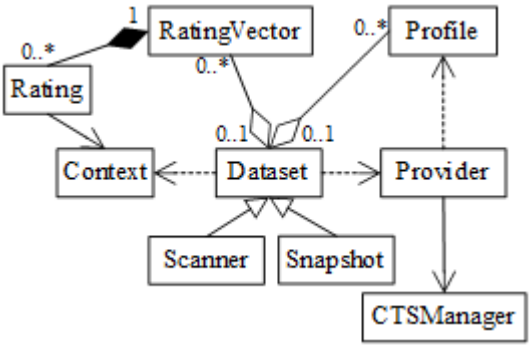

Figure 9: Dataset diagram

Table 2: Provider interface

Data::*Provider*
updateRating(*RatingVector*):*boolean* *updateUserProfile*(*Profile*):*boolean* *updateItemProfile*(*Profile*):*boolean* *getCTSManager*():*CTSManager* . . .

Dataset only provides read-only operations via "get" and "fetch" methods. Thus, *Provider* interface and its implementations support storage service in service layer to update and modify database via writing operations. *Provider* interacts directly with database, which is created in data layer. Service layer and data layer are shown in figure 3. Table 2 lists some methods of *Provider*. For example, *Provider*::*updateRating*(*RatingVector*) method saves rating values that user makes on items (specified by the input parameter *RatingVector*) to database.

Provider also provides read-only accesses to database. So, in many situations, *Scanner* uses *Provider* to retrieve information from database because *Scanner* does not store information in memory. Moreover, *Provider* manipulates context information via context template manager. Context template and context template manager are represented by interfaces *ContextTemplate* and *CTSManager*, which will be mentioned later. Figure 9 implicates relationships among *Provider*, *Scanner*, and *CTSManager*.

Context is additional information relevant to users' activities, for example, time and place that a customer purchases online. Context is modeled by *Context* class. It is necessary to context-aware recommendation. Concretely, context information

stored in *RecommendParam* is passed to *Recommender*::*recommend*(*RecommendParam*, *int*) method whenever a recommendation request is raised. Basic methods of *Context* are described as follows:

- Method *getTemplate*() returns the template of current context. Context template will be described later.
- Method *getValue*() returns the value of current context. This value can be anything and so, it is represented by *ContextValue* interface.
- Constructor *Context*(*ContextTemplate*, *ContextValue*) creates a context from a template and a value.
- Method *canInferFrom*(*Context*) indicates whether or not the current context can be inferred from the context specified by the input parameter. Method *canInferTo*(*Context*) indicates whether or not the current context can lead to the context specified by the input parameter. For example, current context "8th December 2015" implies context "December 2015", which means that method *canInferTo*("December 2015") returns true given the current context "8th December 2015".

Context can be categorized into three main types in order to answer three questions "when, where and who" as follows (Ricci et al., 2011, pp. 224-225):

- Time type indicates the time when user makes a purchase, for example: date, day of week, month, year.
- Location type indicates the place where user makes a purchase, for example: shop, market, theater, coffee house.
- Companion type indicates the persons with whom user makes a purchase, for example: alone, friends, girlfriend/boyfriend, family, co-workers.

Context type, considered as context template, is modeled by the *ContextTemplate* interface whose essential methods are described as follows:

- Methods *getName*() and *setName*(*String*) are used to get and set a name of context template.
- Method *canInferFrom*(*ContextTemplate*) indicates whether or not the current context template can be inferred from the context template specified by the input parameter. Method *canInferTo*(*ContextTemplate*) indicates whether or not the current context template can lead to the context template specified by the input parameter. These methods share the same meaning to ones of *Context*. For example, template "Year" can be inferred (extracted) from template "Date".

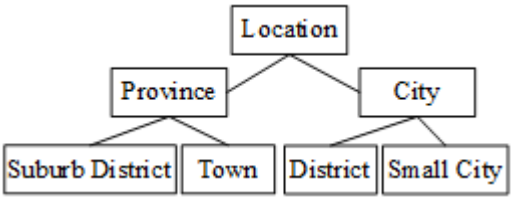

Figure 10: Hierarchical context templates

Contextual information is organized in two structures such as hierarchical and multi-dimensional (Ricci et al., 2011, pp. 225-228). The default implementation of *ContextTemplate* interface is *HierContextTemplate* class which conforms hierarchical structure. According to hierarchical structure, templates are arranged in a tree. Figure 10 shows some *HierContextTemplate* (s), in which template "Location" is the parent of templates "Province" and "City" which, in turn, are parents of templates "Suburb District", "Town", "District", "Small City".

A set of many *ContextTemplate* (s) compose a context template schema (CTSchema) which is specified by *ContextTemplateSchema* interface. Figure 10 is an example of template schema. *ContextTemplateSchema* defines methods to manipulate its *ContextTemplate* members, for example:

- Method *getRoot*() returns the root template. Method *addRoot*(*ContextTemplate*) adds a new root template.
- Method *getTemplateByName*(*String*) retrieves a *ConextTemplate* given a name.

ContextTemplateSchema is then managed by the aforementioned interface *CTSManager*. Functions of *CTSManager* are specified by its main methods as follows:

- Method *setup*(*DataConfig*) is responsible for initializing *ContextTemplateSchema* according to configurations specified in the input parameter.
- Method *commitCTSchema*() verifies and saves *ContextTemplateSchema* to database.
- Method *getCTSchema*() allows us to retrieve *ContextTemplateSchema*.
- Each *Context* is always associated with a *ContextTemplate* and a *ContextValue*, which implies a context is a context template evaluated by a value. So a *Context* is stored in database as a *Profile* which contains a *ContextTemplate* and a *ContextValue*. For example, context "8th December 2015" has template "Date" and value "8th December 2015" and hence, this context is stored in database as a profile with two fields "Date, 8th December 2015". Therefore, *CTSManager* pro-

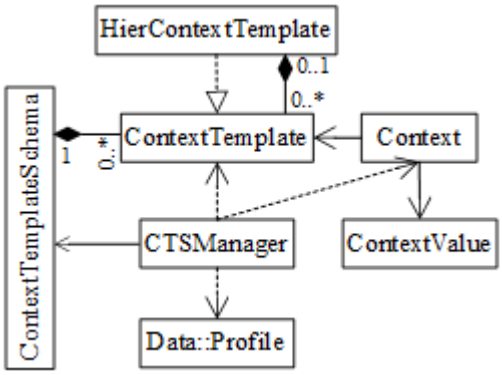

Figure 11: Context information diagram

vides method *profileOf*(*ContextTemplate*, *ContextValue*) to retrieve a *Context* given input parameters: context template and context value.

Figure 11 shows diagram of interfaces and classes relevant to context information.

Now main interfaces and classes of *Data* package are described. *Parser* package aims to support *Data* package. Concretely, *Dataset* is created from database by *DatasetParser* which is the core interface of *Parser* package. The main method of *DatasetParser* is *parse*(*DataConfig*):*Dataset*, which takes database configurations (database type, database connection specification, etc.) as input parameter to create *Dataset* from database as returned output. Following are other methods of *DatasetParser*.

- Method *getName*() returns name of *DatasetParser*, for example, "MyParser".
- Method *support*(*DataDriver*) checks whether or not a given *DatasetParser* supports the kind of database (CSV file, compressed file, relation database, etc.) specified by the input parameter *DataDriver*.

Snapshot and *Scanner*, two common implementations of *Dataset*, are parsed by *SnapshotParser* and *ScannerParser* which are two inherited interfaces of *DatasetParser*, respectively. *MovieLensParser*, an implementation of *SnapshotParser*, is responsible for reading MovieLens dataset (GroupLens, 1998). Figure 12 shows diagram of parsers.

DatasetParser and *CTSManager* are also *Alg* interfaces because they inherits from *Alg*. All *Alg* classes are managed and automatically discovered by plugin manager mentioned in subsection 3.5. This implies developer can add more custom parsers and context template managers. So the number of them is unlimited.

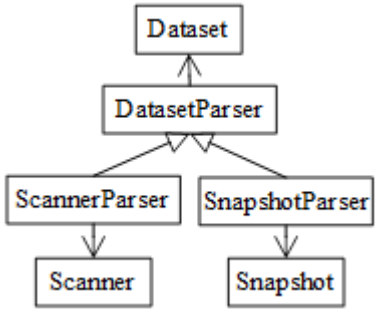

Figure 12: Parser diagram

3.3 Evaluation package

Evaluator is the core class of *Evaluation* package. Please distinguish *Evaluator* class from *Evaluator* module aforementioned in section 2 but pay attention that *Evaluator* class is the heart of *Evaluator* module and they are referred mutually. *Evaluator* class configures and feeds *Dataset* to *Recommender* class because *Recommender* requires *Dataset* via its method *Recommender*::*setup*(*Dataset*) before performing recommendation task. After that *Evaluator* activates two main methods of *Recommender*, namely *Recommender*::*estimate*() and *Recommender*::*recommend*(). As an evaluator of any recommendation algorithm, *Evaluator* is the bridge between *Dataset* and *Recommender* and it has six roles:

1. It is a loader which loads and configures *Dataset*.
2. It is an executor which calls methods *Recommender*::*estimate*() and *Recommender*::*recommend*().
3. It is an analyzer which analyzes and translates the result of algorithm execution into the form of evaluation metrics. The execution result is output of method *Recommender*::*estimate*() or *Recommender*::*recommend*(). Evaluation metric is represented by *Metric* interface. *Metrics* class manages a list of *Metric* (s).
4. It is a registry. If external applications require receiving result from *Evaluator*, they need to register with it. Such applications must implement *EvaluatorListener* interface. In other words, *Evaluator* contains a list of *EvaluatorListener* (s).
5. Whenever it finishes a call of method *Recommender*::*estimate*() or *Recommender*::*recommend*(), it issues a so-called evaluation event and send back evaluation metrics to external applications after executing algorithm. So it is also a provider. The evaluation event is wrapped by *EvaluatorEvent* class.

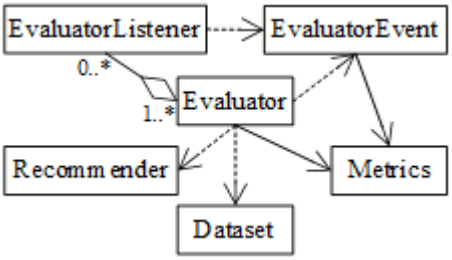

Figure 13: Evaluator diagram

6. It works as a service which allows scientists to start, pause, resume, and stop the evaluation process via its methods *start*(), *pause*(), *resume*(), and *stop*(), respectively.

Evaluator has four most important methods:

1. Method *evaluate*() performs main tasks of *Evaluator*, which loads *Dataset* and activates method *Recommender*::*estimate*() or *Recommender*::*recommend*() on such *Dataset*.
2. Method *analyze*() is responsible for analyzing the result returned by method *Recommender*::*estimate*() or *Recommender*::*recommend*() so as to translate such result into evaluation metric. Metrics are used to assess algorithms and they are discussed later. By default implementation, *analyze*() method will simply call *Metric*::*recalc*() method in order to calculate such metric itself.
3. Method *issue*() issues an evaluation event and sends back evaluation metrics to external applications. Method *issue*() is also named *fireEvaluatorEvent*().

These method are integrated together within current implementation of *Evaluator* but their purposes are kept intact. *Evaluator* is associated tightly with *Recommender*, *Dataset*, *Metrics*, and *EvaluationListener*, which is shown in figure 13.

Metric interface specifies the final result of algorithm evaluation. In other words, there are two kinds of result.

- Recommendation result is represented by the output of method *Recommender*::*estimate*() or *Recommender*::*recommend*(), which is used for recommendation process.
- Evaluation result is represented by *Metric* interface which is the output of *Evaluator*::*analyze*(). It is derived from recommendation result and used for qualifying algorithm.

Followings are essential methods of *Metric*:

- Method *recalc*(*Object*[]) is the most important one expressing how to calculate a concrete *Metric*. It is called by *Evaluator*::*analyze*(). Its input parameter is array of objects passed by *Evaluator*.
- Methods *getAccumValue*() and *getCurrentValue*() return accumulative value and current value of metric. After each time *Metric* is calculated by *recalc*(*Object*[]) method, accumulative value and current value can be changed. For example, a sample *Metric* receives values 3, 1, 2 at the first, second, and third calculations. At the fourth calculation if *recalc*(*Object*[]) produces value 2, the *getCurrentValue*() will return 2 and the *getAccumValue*() will return 3 + 1 + 2 + 2 = 8. Methods *getAccumValue*() and *getCurrentValue*() can return any thing and so their returned value is represented by *MetricValue* interface. How to implement *MetricValue* is dependent on concrete application.

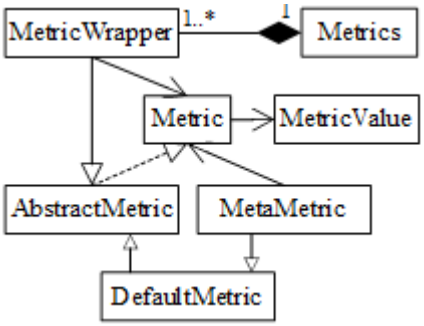

Figure 14: Metrics diagram

The abstract class of *Metric* interface is *AbstractMetric*. Three implemented classes of *Metric* which inherit from *AbstractMetric* are *DefaultMetric*, *MetaMetric*, and *MetricWrapper* as follows:

- *DefaultMetric* class is default implementation of single *Metric*.
- *MetaMetric* class is a complex *Metric* which contains other metrics.
- In some situations, if metric requires complicated implementation, it is wrapped by *MetricWrapper* class.

The *Metrics* class manages a list of *Metric* (s). It uses *MetricWrapper* for sophisticated management tasks. Exactly, *Metrics* contains *MetricWrapper* objects and each such object wraps a *Metric* template. Figure 14 shows diagram of metrics.

Currently, two default metrics inherited from *DefaultMetric* class are *TimeMetric* class and *Accuracy* class. *TimeMetric* measures the speed of algorithm and so it is the time in seconds that methods *Recommender*::*estimate*() and *Recommender*::*recommend*() execute over *Dataset*.

Accuracy reflects goodness and efficiency of recommendation algorithms. There are three types of accuracy.

- Predictive accuracy, represented by *PredictiveAccuracy* class, measures how close predicted ratings returned from methods *Recommender*::*estimate*() and *Recommender*::*recommend*() are to the true user ratings (Herlocker et al., 2004, pp. 20-21). *PredictiveAccuracy* class derives directly from *Accuracy* class. *MAE*, *MSE*, *RMSE* (Herlocker et al., 2004, pp. 20-21) are typical predictive accuracy metrics.
- Classification accuracy, represented by *ClassificationAccuracy* class, measures the frequency that methods *Recommender*::*estimate*() and *Recommender*::*recommend*() make correct or incorrect recommendation (Herlocker et al., 2004, pp. 20-22). *ClassificationAccuracy* class derives directly from *Accuracy* class. *Precision* and *Recall* (Herlocker et al., 2004, pp. 22-25) are typical classification accuracy metrics.
- Correlation accuracy, represented by *CorrelationAccuracy* class, measures the ability that method *Recommender*::*recommend*() makes the ordering of recommended items which is similar to the ordering of user's favorite items (Herlocker et al., 2004, pp. 29-33). *CorrelationAccuracy* class derives directly from *Accuracy* class. *NDPM*, *Spearman*, *Pearson* (Herlocker et al., 2004, pp. 29-33) are typical correlation accuracy metrics.

Figure 15 lists these default metrics known as pre-defined or built-in metrics. *Metric* like *Algorithm*::*Recommender*, *Data*::*CTSManager*, *Parser*::*DatasetParser* is also *Alg* interface because it derives from *Alg*. So developers can define new metrics.

If external applications want to receive metrics, they need to register with *Evaluator* by calling *Evaluator*::*addListener*() method. The evaluation process has five steps:

1. *Evaluator* calls *Evaluator*::*evaluate*() method to load and feed *Dataset* to *Recommender*.
2. Method *Recommender*::*estimate*() or *Recommender*::*recommend*() is executed by *Evaluator*::*evaluate*() method to perform recommendation task.
3. Method *Evaluator*::*analyze*() analyzes the result returned by method *Recommender*::*estimate*() or *Recommender*::*recommend*() and translates such result into *Metric*. The *Metrics* class manages a list of *Metric* (s).

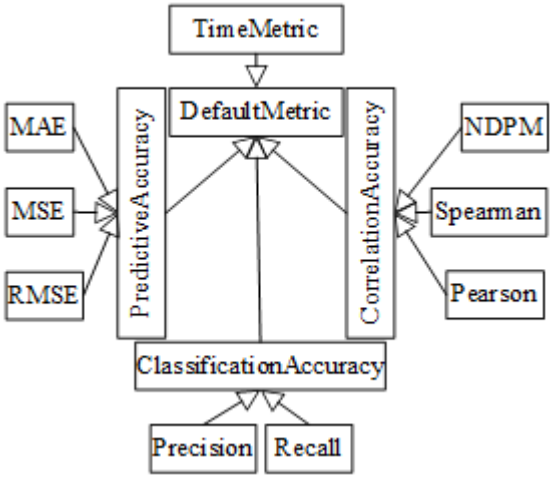

Figure 15: Default metrics diagram

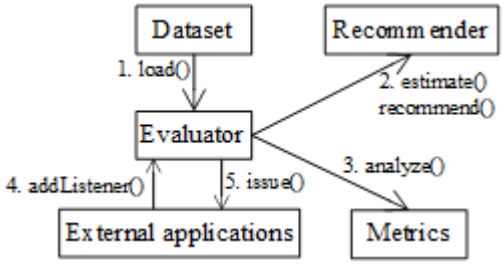

Figure 16: Evaluation process

4. External applications that implement *EvaluatorListener* interface register with *Evaluator* by calling *Evaluator*::*addListener*() method.
5. Method *Evaluator*::issue() sends *Metrics* to external applications.

Figure 16 is a view of such five steps.

3.4 Client, Server, and Listener packages

Client package defines classes and interfaces necessary to client-server network. Following are basic classes and interfaces in *Client* package:

- *Request* class represents user request. *Request* uses JSON format (ECMA, 2013) as exchangeable means in client-server network. As aforementioned in section 2, user request includes recommendation request, retrieval request, update request.
  - Recommendation *Request* is that user prefers to get favorite items. Exactly, this *Request* wraps *RecommendParam* which is the first input parameter of methods *Recommender*::*estimate*() and *Recommender*::*recommend*().

Table 3: Protocol interface

Client::*Protocol*
createRecommendRequest(*RecommendParam*):*Request* *createUpdateRatingRequest*(*RatingVector*):*Request* *createUpdateUserProfileRequest*(*Profile*):*Request* ...

  - Retrieval or update *Request* is that user wants to retrieve or update her/his ratings and profiles.
- *Response* class represents result of recommendation process. *Response* uses JSON format as exchangeable means.
  - In case of recommendation request, *Response* is the list of recommended items returned from service layer (see figure 3). Exactly, *Response* wraps the returned *RatingVector* of methods *Recommender*::*estimate*() and *Recommender*::*recommend*().
  - For update request, *Response* is the output variable indicating whether or not update request is successful.
  - For retrieval request, *Response* is the piece of information stored in *Dataset* or *KBase*. If information is stored in *Dataset*, *Response* often wraps a returned *Snapshot*.
- *Protocol* interface specifies methods to create *Request* in many possible cases. *Protocol* establishes an interaction protocol of Hudup client-server network, which is named "hdp". Table 3 shows some methods of *Protocol*.
- *Service* interface specifies methods to serve user requests. These methods focus on providing recommendation, inserting, updating and getting information such as user ratings, user profiles, and item profiles stored in database. Storage service and recommender service (see figure 3) implement *Service* interface. They are typical *Service* (s). Table 4 lists some methods of *Service*.
- *Server* interface defines abstract model of recommendation server. *Server* is responsible for creating *Service* to serve user requests. *Server* starts, pauses, resumes, and stops by methods *start*(), *pause*(), *resume*(), and *stop*(), respectively. It must be implemented by programmer. Both *Server* and *Service* constitute a proper recommendation server.

Association relationships among *Request*, *Response*, *Protocol*, *Service*, and *Server* are tight. Figure 17 shows diagram of *Client* package.

Server package focuses on implementing recommendation server. Following are classes and interfaces in *Server* package:

Table 4: Service interface

Client::*Service*
estimate(*RecommendParam*, *int*[]):*RatingVector* *recommend*(*RecommendParam*, *int*):*RatingVector* *updateRating*(*RatingVector*):*boolean* *getUserRating*(*int*):*RatingVector* *getUserProfile*(*int*):*Profile* *updateUserProfile*(*Profile*):*boolean* ...

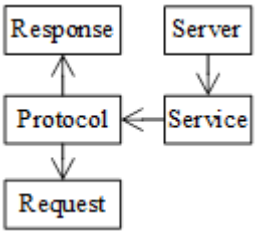

Figure 17: Client diagram

- *DefaultService* class is default implementation of aforementioned *Client*::*Service* interface. *DefaultService* uses *Recommender* and *Data*::*Provider* for processing recommendation request and update request, respectively. Developers are suggested to build up their own services.
- *DefaultServer* class is default implementation of aforementioned *Client*::*Server* interface. *DefaultServer* creates and manages *DefaultService*. Developers are suggested to build up their own servers.
- *Transaction* interface represents transaction layer (see figure 3) in the architecture of *Recommender* module. *Transaction* is responsible for managing concurrence data accesses. Currently, *DefaultServer* is responsible for implementing *Transaction* interface.
- *ActiveMeasure* interface specifies how to measure the active degree of *DefaultServer*. For example, there is a counter inside *ActiveMeasure*. Each time *DefaultServer* receives a user request, the counter is increased by 1. After *DefaultServer* finishes serving such user request, the counter is decreased by 1. If there are many *DefaultServer* (s) deployed on different sites, balancer aforementioned in section 2 uses *ActiveMeasure* to determine which *DefaultServer* is least busy in order to dispatch user request to such *DefaultServer* whose active counter is smallest.

Figure 18 shows diagram of *Server* package.

Listener package focuses on implementing listener and balancer aforementioned in section 2. Following are classes in *Listener* package:

- *Listener* class represents a listener shown in fig-

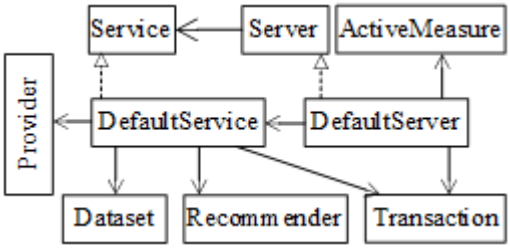

Figure 18: Server diagram

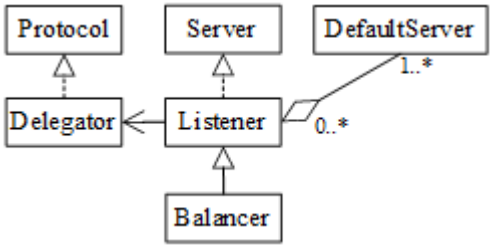

Figure 19: Listener diagram

ure 3. *Listener* also implements *Client*::*Server* but it is not recommendation server. *Listener* is deployed in interface layer, which supports distributed environment. *Listener* stores a list of binding *Server*::*DefaultServer* (s). It can bind a new *Server*::*DefaultServer* by calling its *rebind*(*Server*) method. In case that there are many *Server*::*DefaultServer* (s) and one *Listener* then, *Listener* is responsible for dispatching user requests to its proper binding *Server*::*DefaultServer*. *Listener* starts, pauses, resumes, and stops by its methods *start*(), *pause*(), *resume*(), and *stop*(), respectively.

- *Delegator* class implements *Client*::*Protocol*, which is responsible for handling and processing user request represented by *Client*::*Request*. Each time *Listener* receives a user request, it creates a respective *Delegator* and passes such request to *Delegator*. After that *Delegator* processes and dispatches the request to the proper binding *Server*::*DefaultServer*. The result of recommendation process from *Server*::*DefaultServer*, represented by *Client*::*Response*, is sent back to users/applications by *Delegator*. In fact, *Delegator* interacts directly with *Server*::*DefaultServer*. However, *Delegator* is a part of *Listener* and the client-server interaction is always ensured.

- *Balancer* class represents a balancer shown in figure 3. *Balancer* derives from *Listener* and overrides the *rebind*(*Server*) method in order to support balancing function while it inherits all other functions of *Listener*. *Balancer* selects least busy binding *Server*::*DefaultServer* to which user requests are dispatched. Therefore, *Balancer* improves system performance more than *Listener* does. In general, *Listener* and *Balancer* are main components of interface layer.

Figure 19 shows diagram of *Listener* package.

In general, three packages *Client*, *Server*, and *Listener* with support of package *Algorithm* compose *Recommender* module - a complete recommendation server which is also called simulator.

3.5 Plugin package

PluginManager interface and *RegisterTable* class are main parts of *Plugin* package. Both of them are initialized whenever Hudup framework boots. Method *discover*() of *PluginManager* is executed at starting time of Hudup, which automatically discovers all algorithms that implement *Alg* interface and registers such algorithms in *RegisterTable*. *RegisterTable* then manages these algorithms. *RegisterTable* provides two important methods *register*(*Alg*) and *query*(*String*) as follows:

- Method *register*(*Alg*) registers a given algorithm. This method is called by *PluginManager*::*discover*() at starting time of Hudup.

- Method *query*(*String*) retrieves an algorithm by its name with attention that such algorithm is named by returned value of its *Alg*::*getName*() method. Method *query*(*String*) is often called by *Evaluation*::*Evaluator* and *Client*::*Service*.

Algorithm::*Recommender*, *Data*::*CTSManager*, *Parser*::*DatasetParser*, and *Evaluation*::*Metric* are typical *Alg* (s), managed by *RegisterTable*. This implies *Plugin* package supports the high customization of Hudup. Scientists and software developers can define any algorithm following *Alg* specifications and put such algorithm in Hudup framework (Hudup class path). Consequently, *PluginManager* discovers the algorithm automatically and *RegisterTable* stores it for later use. It is easy for *Evaluation*::*Evaluator* and *Client*::*Service* to get the user-defined algorithm by calling *RegisterTable*::*query*(*String*). *Plugin* package is small but it is very important for Hudup framework. Figure 20 shows diagram of *Plugin* package.

Core components of Hudup, classes and interfaces, are now described in details so that readers can comprehend the Hudup infrastructure. Section 4 is a tutorial on how to use Hudup product.

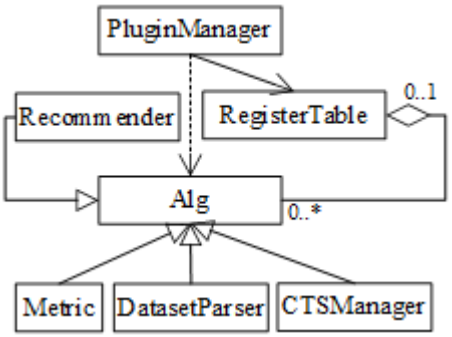

Figure 20: Plugin diagram

4 TUTORIAL ON PROPOSED FRAMEWORK

Suppose 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. This tutorial has three simple steps to implement, evaluate, and deploy your custom algorithm, corresponding to three stages aforementioned in section 1 such as base stage, evaluation stage, and simulation stage. These steps are explained in subsections 4.1, 4.2, and 4.3, respectively.

4.1 Implementing algorithm

Firstly, you create the Java project named Green Fall and import the core Java library package "hudup-core.jar" into Green Fall project; the core library contains all core classes and interfaces aforementioned in section 3. This starts the base stage proposed by Hudup framework. Note that Hudup is written by the object-oriented language Java (Oracle, 2014). You create the *GreenFall* class inheriting from *ModelBasedCF* class because Green Fall is model-based recommendation algorithm. Before implementing your idea in two methods *estimate*() and *recommend*() of *GreenFall*, you create its knowledge base *GreenFallKB* derived directly from *KBase*. *GreenFallKB* will contains frequent itemsets. Following is abstract code of *GreenFallKB* class in which you must override the *GreenFallKB*::*learn*() method to mine frequent itemsets from *Dataset*, according to code lines 6-10:

```
(01) class GreenFallKB
(02)       implements KBase {
(03)   ...
(04)   Itemsets frequentItemsets;
(05)
(06)   public void learn(
(07)     Dataset dataset, Alg alg) {
(08)     //Your code hear
(09)     //to mine frequent itemsets
(10)   }
(11)
(12)   Itemsets getFrequentItemsets() {
(13)     return frequentItemsets;
(14)   }
(15)   ...
(16) }
```

Your idea is realized in methods *estimate*() and *recommend*() of main class *GreenFall* as follows:

```
(01) public class GreenFall
(02)       extends ModelBasedCF {
(03)   ...
(04)   GreenFallKB kb;
(05)
(06)   public String getName() {
(07)     return "gfall";
(08)   }
(09)
(10)   public void setup(
(11)       Dataset dataset) {
(12)     kb = new GreenFallKB();
(13)     kb.learn(dataset, this);
(14)   }
(15)
(16)   public RatingVector estimate(
(17)       RecommendParam param,
(18)       int[] itemIds) {
(19)     Itemsets frequentItemsets =
(20)       kb.getFrequentItemsets();
(21)
(22)     RatingVector vEstimated=null;
(23)     //Your code here
(24)     //to estimate missing values
(25)     //based on frequent itemsets
(26)
(27)     return vEstimate;
(28)   }
(29)
(30)   public RatingVector recommend(
(31)       RecommendParam param,
(32)       int userId) {
(33)     Itemsets frequentItemsets =
(34)       kb.getFrequentItemsets();
(35)
(36)     RatingVector vRecommended=null;
(37)     //Your code here
(38)     //to produce recommended items
(39)     //based on frequent itemsets
(40)
(41)     return vRecommended;
(42)   }
(43)   ...
(44) }
```

According to the code lines 16-28 and 30-42 above, methods *GreenFall*::*estimate*() and *GreenFall*::*recommend*() apply frequent itemsets resulted from *GreenFallKB* into estimating rating values of given items and producing recommended items of given user, respectively. All things you do manually are to implement these two methods. The short name of Green Fall algorithm is "gfall" as the returned

value of *getName*() method (see code lines 6-8). Now 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 Hudup so that *PluginManager* can discover and register Green Fall algorithm in *RegisterTable*.

4.2 Evaluating algorithm

Secondly, you open *Evaluator* to assess Green Fall algorithm, as shown in figure 21. This is evaluation stage proposed by Hudup framework. Exactly, what you see in figure 21 is the graphic user interface (GUI) of *Evaluator* module. As a convention, this GUI refers to *Evaluator* in the tutorial.

The *Evaluator* discovers the Green Fall algorithm via the name "gfall". Exactly, *Evaluator* calls *RegisterTable*::*query*("gfall") to retrieve it. Now you evaluate the Green Fall algorithm by *Evaluator*. Database Movielens (GroupLens, 1998) is used for evaluation but you do not need to focus on its structure because *Evaluator* processes it automatically. Moreover, the complex evaluation mechanism and metrics system built in *Evaluator* are automatically applied into evaluating Green Fall algorithm according to pre-defined metrics. Four important pre-defined metrics aforementioned in subsection 3.3 are used in this evaluation: *MAE*, *Precision*, *Recall*, and *TimeMetric*. *TimeMetric* known as speed metric is calculated in seconds. *MAE* is predictive accuracy metric that measures how close predicted value is to rating value. The less *MAE* is, the high accuracy is. *Precision* and *Recall* are quality metrics that measure the quality of recommendation list how much the recommendation list reflects user's preferences. The large quality metric is, the better algorithm is.

Evaluator allows you to define custom metrics. Suppose we create a new metric called *MSE* which is a predictive accuracy metric. *MSE*, abbreviation of mean squared error, measures the average squared deviation between the predictive rating and user's true rating (Herlocker et al., 2004). Note that methods *Recommender*::*estimate*() and *Recommender*::*recommend*() return predictive (estimated) ratings. *MSE* is calculated by equation (1).

$$MSE = \frac{\sum_{i=1}^{n}(p_i - v_i)^2}{n} \quad (1)$$

Where n is the total number of recommended items while p_i and v_i are predictive rating and true rating of item i, respectively. In order to realize equation (1), you create the new *MSE* class derived from *PredictiveAccuracy* class and override *calc*() method, as follows:

```
(01) public class MSE
(02)     extends PredictiveAccuracy {
(03)   ...
(04)   public String getName() {
(05)     return "MSE";
(06)   }
(07)
(08)   protected MetricValue calc(
(09)     RatingVector vPredicted,
(10)     RatingVector vTrue) {
(11)
(12)     float sum = 0;
(13)     int n = 0;
(14)     int[] itemIds =
(15)       vPredicted.getItemIds();
(16)     for (int itemId : itemIds) {
(17)       float dev =
(18)           vPredicted.get(itemId) -
(19)           vTrue.get(itemId));
(20)       sum += dev * dev;
(21)       n++;
(22)     }
(23)
(24)     return new
(25)       FractionMetricValue(sum, n);
(26)   }
(27)   ...
(28) }
```

The *calc*() will be called by the *Metric*::*recalc*() which, in turn, is called by the *Evaluator*::*analyze*(). According to code lines 9-10, *vPredicted* and *vTrue* represent predictive ratings and user's true ratings. At lines 12-22, the sum $\sum_{i=1}^{n}(p_i - v_i)^2$ inside equation (1) is calculated. At lines 24-25, specific *FractionMetricValue* representing the value of *MSE* is returned. *FractionMetricValue* class with two atributes a and b, an implementation of *MetricValue* interface, represents a fraction in form of $\frac{a}{b}$. *MSE* metric is an *Alg* and so its name "MSE" is returned value of the *getName*() method at lines 4-6 so that it is registered by *PluginManager*.

Now you put *MSE* class into package "gfal.jar" so that *Evaluator* can discover it. Consequently, you execute *Evaluator* to evaluate Green Fall algorithm according to five metrics: *MAE*, *MSE*, *Precision*, *Recall*, and *TimeMetric*. Figure 22 shows the evaluation result in which the speed of Green Fall algorithm is 0.0055. Currently, iconic images of GUI (s) shown in figures 21, 22, 23, 24, 25, 26 are used temporarily and so they will be designed particularly.

4.3 Deploying algorithm

Finally, *Recommender* module, a recommendation server, supports you to deploy and test Green Fall algorithm in real-time application. This is simulation stage proposed by Hudup framework. When you put the package "gfall.jar" into library direc-

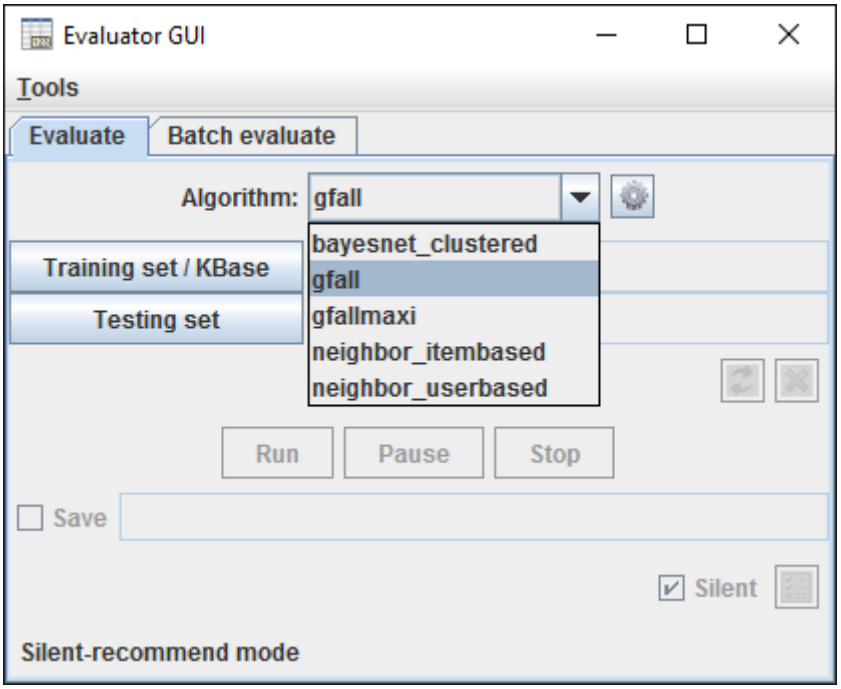

Figure 21: Evaluator discovers Green Fall algorithm

tory of Hudup, *Recommender* automatically discovers and deploys Green Fall on server, as seen in figure 23. Moreover, *Recommender* allows you to configure Green Fall for particular purposes. *Recommender* serves client requests at server port 10151.

You start up and configure *Listener* to connect with *Recommender*. As seen in figure 24, *Listener* connects with *Recommender* through server port 10151 and it serves client requests at listener port 10152. Consequently, you configure *Evaluator* to connect with *Listener*. As seen in figure 25, *Evaluator* connects successfully with *Listener* through listener port 10152. The connection between *Evaluator* and *Listener* complies with the basic socket protocol. Now *Listener* becomes a bridge between *Evaluator* and Recommender, which allows *Evaluator* to call Green Fall algorithm deployed remotely in *Recommender*.

Now you use *Evaluator* to test Green Fall algorithm via client-server environment with cooperation of *Evaluator*, *Listener*, and *Recommender*. Figure 26 shows again the evaluation result of Green Fall algorithm from remote execution. *Evaluator* interacts with *Listener* through listener port 10152 and *Listener*, in turn, interacts with *Recommender* through server port 10151. Due to network transportation, the speed of Green Fall algorithm is decreased, which is 0.0275 whereas it is 0.0055 given local execution as seen in figure 22. However, the decrease of speed is insignificant within two cases as follows:

- *Recommender* server is deployed on a powerful computer. Cost of network transportation is smaller than cost of algorithm execution.
- There are many *Recommender* servers deployed in distributed environment and *Listener* is replaced by *Balancer* which supports balancing mechanism aforementioned in subsection 3.4.

5 CONCLUSIONS

In general, 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 algorithms, is the best choice for you to build up a recommendation solution. Moreover, it is not only used for e-commercial software but also applied into any application in which recommendation is necessary.

The framework has 6 most essential components (classes and interfaces) such as *Algorithm*::*Recommender*, *Data*::*Dataset*, *Data*::*KBase*, *Evaluation*::*Evaluator*, *Evaluation*::*Metric*, and *Client*::*Service*. Each component has individual

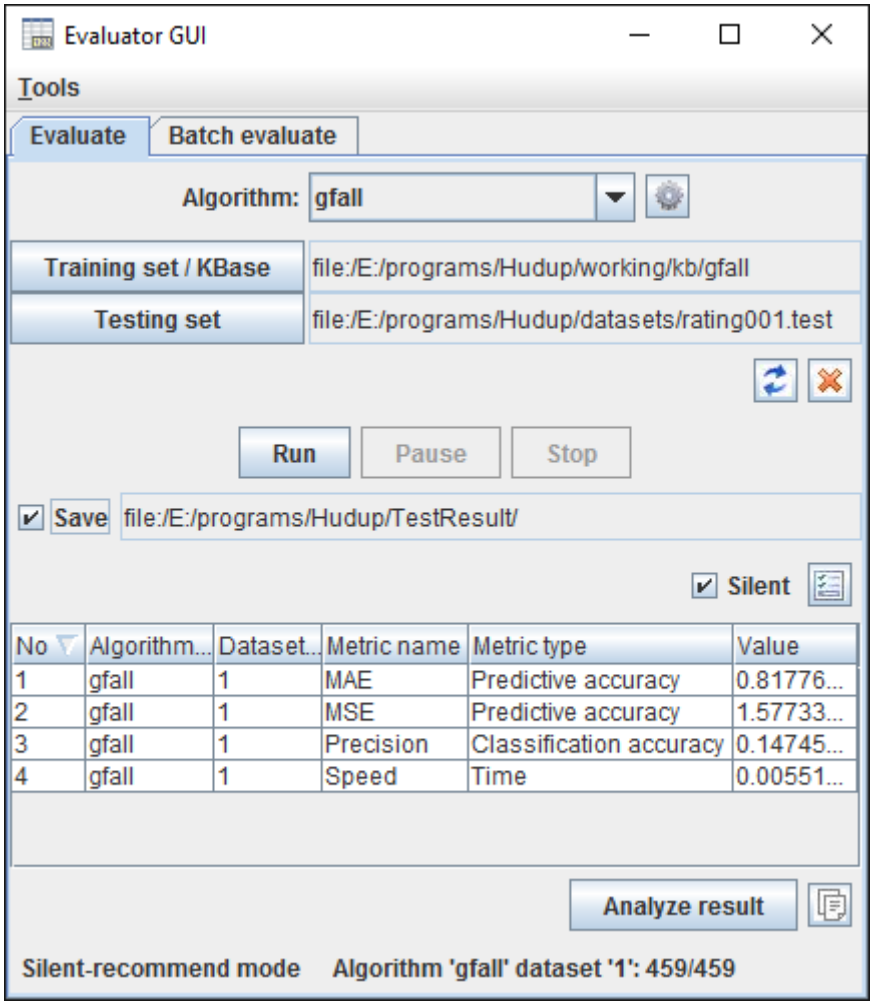

Figure 22: Evaluator lists evaluation result of Green Fall algorithm

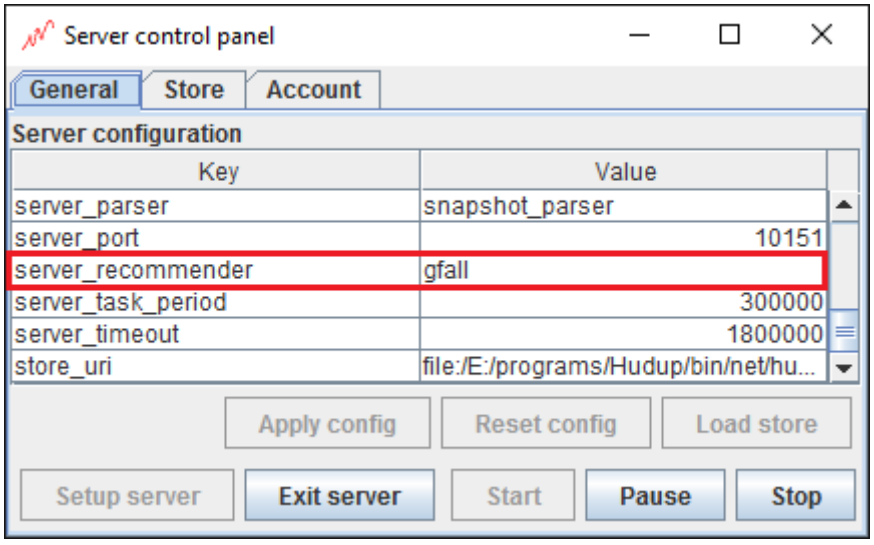

Figure 23: Recommendation server deploys Green Fall algorithm

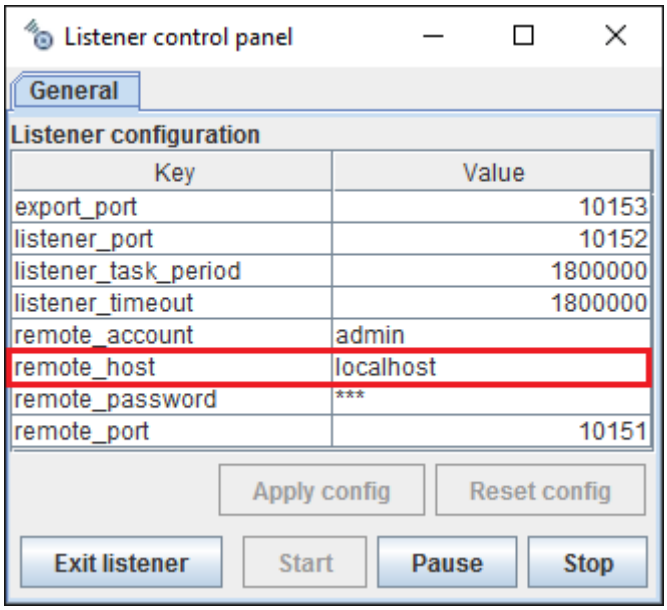

Figure 24: Listener connects with recommendation server

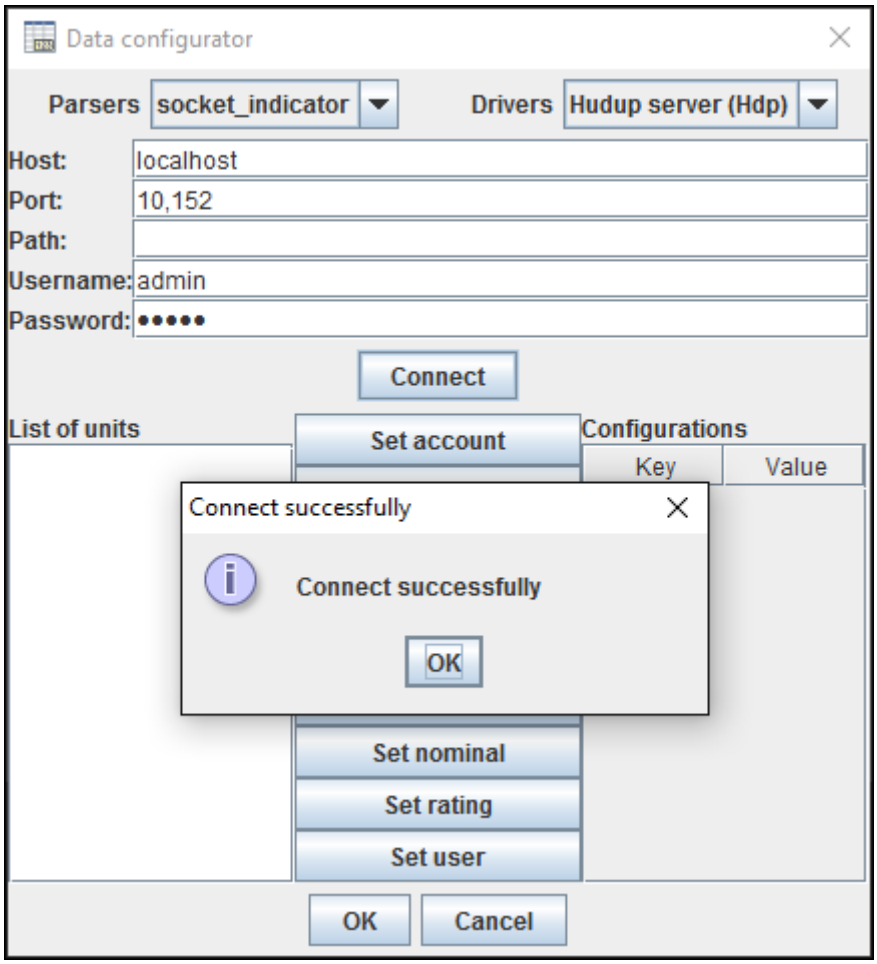

Figure 25: Evaluator connects with Listener

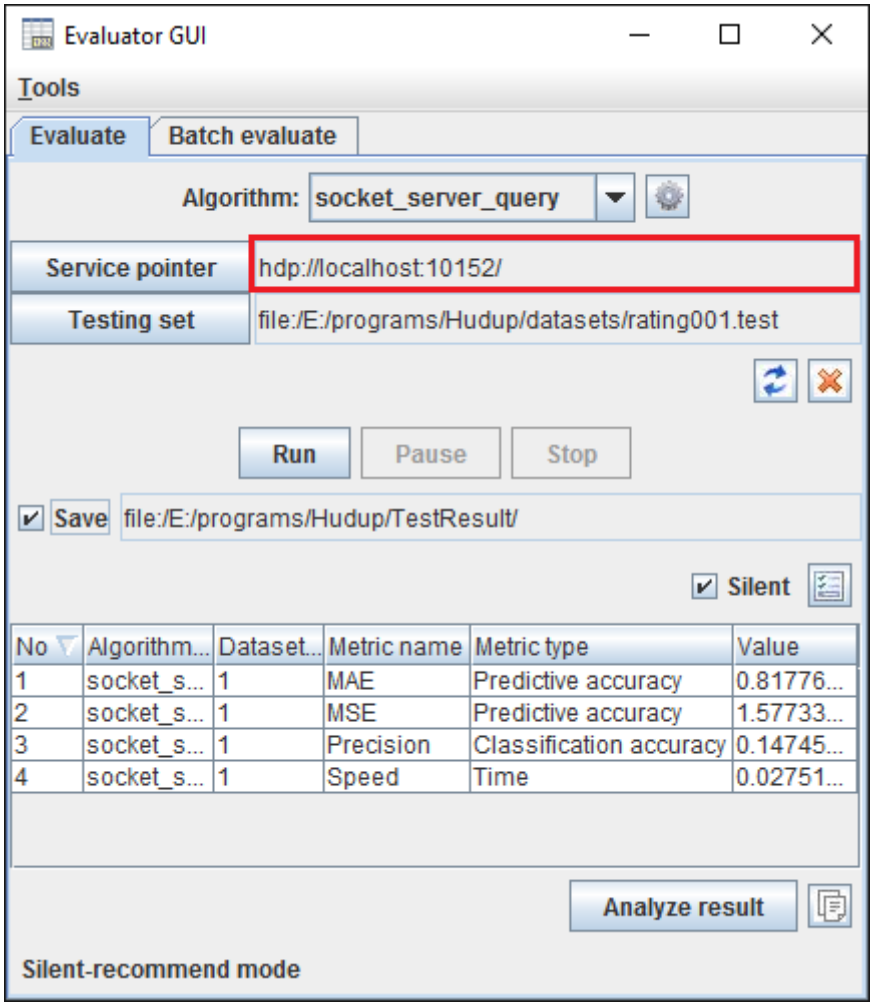

Figure 26: Evaluation result of Green Fall algorithm from remote execution

aspects and roles. They are both strongly interactive and independent so that the general architecture achieves two goals:

- Coherency: The algorithm execution and evaluation are processed continuously and completely. There is no interruption in stages of such process. All components are highly interactive. As a result, Hudup supports smoothly scientists to develop a recommendation solution through three stages: algorithm implementation, quality assessment, and simulation.
- Flexibility: The framework provides the high level of customization to researchers so that they realize easily their ideas. For instance, *KBase* has no structure and no shape; its manifest consists of abstract methods or rules so that researchers implement such rules. Most components are abstract units; so this architecture is called abstract architecture.

In recommendation domain, there is the problem that experimental rating databases such as Movielens (GroupLens, 1998), Jester Joke, and Book Crossing are heterogeneous; so their structures are very different. This problem makes researchers into trouble; they cannot focus on their creative ideas. For example, *Dataset* gives the solution to this problem when it proposes an abstract model of heterogeneous rating database. Researchers don't need to consider what *Dataset* is (Movielens, Jester Joke, or Book Crossing) and how to read it. The infrastructure is responsible for building up dataset from physical devices. Researchers enjoy methods of *Dataset*. Similarly, *Recommender* is abstract model of any recommendation algorithm. Researchers only need to implement their ideas in methods *Recommender*::*estimate*() and *Recommender*::*recommend*(). The flexibility also leads to ability of high customization. For instance, *Evaluation*::*Metric* interface represents any evaluation metric. Researchers can totally define their own metrics following specifications of *Evaluation*::*Metric*. Concretely, researchers need to override the method *Evaluation*::*Metric*::*recalc*(). The infrastructure automatically discovers and applies such user-defined metrics into evaluation process based on the *Evaluation*::*Evaluator* component.

Finally, in this abstract architecture, we aim to normalize such 6 components as 6 standards for recommendation study. Such standards are used for study of software engineering. For instance, the manifest of *KBase* has following aspects:

- The methods *load*() and *save*() indicate that *KBase* can be loaded from and saved to storage system. They don't specify how to load and store *KBase*. In other words, they are rules with which the infrastructure must comply.
- Similarly, method *learn*() tells us that *KBase* can be learned by any approaches: machine learning, data mining, artificial intelligence, statistics, etc.

Hudup framework is ongoing; more features and utilities are supported in future but its two goals, coherency and flexibility, are keeping constant.

ACKNOWLEDGEMENTS

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

REFERENCES

Brozovsky, L. (2006). Colfi - recommender system for a dating service.

Caraciolo, M., Melo, B., and Caspirro, R. (2011). Crab - recommender systems in python.

Chen, T., Zhang, W., Lu, Q., Chen, K., Zheng, Z., Yu, Y., and Yang, D. (2012). Svdfeature: A toolkit for feature-based collaborative filtering.

Dato-Team (2013). Graphlab create tm.

Duong, T. Q. (2008). *UML 2 and Rational Rose 2003*, volume 8 of *Analyzing and Designing Enterprise Information System*. SAHARA, Ho Chi Minh, Vietnam.

ECMA (2013). *The JSON Data Interchange Format*. Geneva, Swiss, 1 edition.

Ekstrand, M., Kluver, D., He, L., Kolb, J., Ludwig, M., and He, Y. (2013). Lenskit - open-source tools for recommender systems.

Gantner, Z., Rendle, S., Drumond, L., and Freudenthaler, C. (2013). Mymedialite recommender system library.

GroupLens (1998). Movielens datasets.

Hahsler, M. (2014). recommenderlab: Lab for developing and testing recommender algorithms.

Herlocker, J. L., Konstan, J. A., Terveen, L. G., and Riedl, J. T. (2004). Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*, 22(1):5–53.

Lemire, D. (2003). Cofi: A java-based collaborative filtering library.

Lew, D. and Sowell, B. (2007). Carleton recommender systems.

Mahout-Team (2013). Apache mahout tm.

Microsoft, MyMediaPC, RichHanbidge, MyMediaWp2Lead, MyMediaWp3Lead, MyMediaWp4Lead, and MyMediaWp5Lead (2013). Mymedia dynamic personalization and recommendation software framework toolkit.

Oracle (2014). Java language.

Ricci, F., Rokach, L., Shapira, B., and Kantor, P. B. (2011). *Recommender Systems Handbook*. Springer.

Smart-Agent-Technologies (2013). easyrec.

Telematica-Instituut (2007). Duine framework.

Vogoo-Team and DROUX, S. (2008). Vogoo php lib.