**An abstract architecture for collaborative filtering algorithms**

Loc Nguyen

Huong Duong Company, Ho Chi Minh city, Vietnam

**Abstract**

Collaborative filtering (CF) is hot topic of recommendation research because it is efficient approach which gives a lot of benefits to e-commerce system due to two reasons:

* Customers will be attracted by e-commerce websites if interesting products are recommended to them when they visit these websites. So revenue of such e-commerce companies grows up.
* CF algorithms don’t require much information about customers and products. Only database about customers’ ratings on products are necessary to be stored. This saves storage space of system.

However CF has a lot of different and heterogonous algorithms. Both experts and students cope with difficulties when they develop a new idea or begin to learn CF. This report proposes an abstract architecture which is the guideline for implementing and evaluating CF algorithms. Following are two main ideas of this report.

* Giving and explaining the abstract architecture together with basic components in detail.
* Specifying these basic components as software engineering standards in UML language. If researchers comply with such standards, it asserts to be easy for them to write and evaluate a new algorithm. They avoid inventing wheels again.

**1. Introduction**

Although this report focuses on the abstract architecture for collaborative filtering (CF) algorithms, we should glance over CF algorithms so as to give the reason that we propose such architecture. CF is algorithm used in recommendation system which recommends users items in existing database so that users like such items. Items which are to point to anything that user are to considering such as products, books, news papers, etc

The basic idea of CF is to mine social network so as to discover potential user interests. The existing database called as dataset is organized in form of rating matrix in which each row is a rating vector of user on items and each column is a concrete item. Each cell contains the rating value that concrete user gave on concrete item. CF is based on the assumption: “user like an item if her / his neighbors like such item”. There are a lot of CF algorithms, for instance:

* Neighbor algorithm: finding out nearest neighbors of user and estimating her / his rating based on rating values of such neighbors.
* Clustering algorithm: partitioning users into groups and the ratings of concrete user are the same to her / his group
* Bayesian algorithm: users are classified into classes by Naïve Bayesian or Bayesian network classification
* Latent algorithm: discovering latent aspect in rating matrix. Such potential (or latent) information will improve the quality of recommendation
* Dimensionality reduction algorithm: Two serious problems in CF are data sparseness and the huge rating matrix. Dimensionality reduction aims to remove redundant rows or columns but to keep principle or important rows/columns.
* MDP-based algorithm: this algorithm is based on Markov decision process (MDP). Recommendation can be considered as the sequential process including many stages. So recommendation task is the best action that recommender system must do at concrete stage so as to satisfy user’s interest.
* Hybrid algorithm which combine CF and content-based filtering (CBF) so as to take advantages of strong points of both of them and to overcome common weak points.
* ….

In general, each algorithm has individual and different features and there is no common standard or architecture for implementing them. Every researcher or developer built up her / his inventive algorithm according to her / his methodologies or interests. This is the cause of unnecessary complexity and it is very difficult to evaluate algorithms developed heterogeneously. That is the reason we propose an abstract architecture for CF algorithm. In this report, section 2 gives an overview of such architecture. Section *3* describes three basic components of architecture: abstract algorithm, knowledge base and dataset in detail. Section *4* explains the last important component: abstract evaluator and its measures. Section 5 is the conclusion.

**2. General architecture**

As discussed, that there are a lot of CF algorithms make researchers confused when they begin to study or write a new CF algorithm. Problem becomes more serious if they don’t know how to test or evaluate their own algorithm. Thesis proposes an abstract architecture along with a framework which assists them in implementing and evaluating CF algorithms. The framework is defined as infrastructure or software which implements this abstract architecture. The general architecture interpreted by UML language has *4* basic interfaces and classes: *Algorithm, KBase, Dataset* and *Evaluator*. Such interfaces and classes are considered as the software-engineering standard for CF algorithm. Researcher will conform to such standard when they apply this framework into writing a new algorithm.

* Interface *Algorithm* represents abstract algorithm. The main task that researchers does is to realize this interface according to their goals when they invent a new algorithm. In most cases, they implement directly two classes *MemoryBasedCF* and *ModelBasedCF* which are derived from *Algorithm*. *MemoryBasedCF* and *ModelBasedCF* represent memory-based CF algorithm and model-based CF algorithm, respectively.
* Interface *KBase* represents knowledge base which associates with a model-based algorithm *MemoryBasedCF*. The structure of *KBase* is very flexible and it depends on ideas and purposes of algorithm.
* Class *Dataset* is composed of a rating matrix and personal profiles. Each row of rating matrix is represented by class *RatingVector*. Personal profile is represented by class *Profile*. Framework is responsible for manipulating *Dataset*.
* Class *Evaluator* is used by framework in order to evaluate algorithm according to criterions so-called *measures* such as time, precision and recall. Such measures are defined inside *Evaluator*. Two classes derived from *Evaluator* are *QueryEvaluator* and *RecommendEvaluator*. *Evaluator* reads and feeds dataset on algorithm *Algorithm*. Finally, it evaluates such algorithm by calculating *Measures* based on result of executing algorithm.

If researchers invent a new algorithm, what they need to do includes two main tasks:

* Writing an algorithm class derived from *MemoryBasedCF* or *ModelBasedCF*. In case of inheriting from *ModelBasedCF*, they must implement *KBase* in accordance with their algorithm.
* Compressing algorithm class, *KBase* class and all artifacts into a file so-called *plugin* and copying this plug-in to the framework.

Please pay attention some notes:

* Terms “*algorithm, abstract algorithm, algorithm class, abstract algorithm class, algorithm interface, abstract algorithm interface*” refer to *Algorithm* interface, *MemoryBasedCF* class, *ModelBasedCF* class and all classes derived from them if there is no specific explanation.
* Terms “*knowledge base, kbase, KBase*” refer to *KBase* interface if there is no specific explanation
* Terms “*Dataset, dataset*” refer to *Dataset* class
* Terms “*Evaluator, evaluator*” refer to *Evaluator* class and its derivative classes.
* Terms “*user rating vector*, *rating vector, RatingVector*” refer to *RatingVector* class.
* Terms “*Measures, measures*” refer to *Measures* class.
* Method is denoted as *method*() or *Class*::*method*(). If class name is removed from this method format, such class is known in context.
* Property is denoted as *Property* or *Class*::*Properties*.

In general all classes, method and properties complying with UML standard are written in italic font. When they are implemented to constitute a framework or software, they are called components.



**Figure 2.1**: General architecture for CF algorithms

**3. Abstract algorithm, knowledge base and dataset**

The set of four basic interfaces and classes: *Algorithm, KBase, Dataset* and *Evaluator* is the heart of this abstract architecture. This section focuses on them. A CF algorithm is represented by the abstract interface *Algorithm* but it really modeled by *MemoryBasedCF* class or *ModelBasedCF* class. These classes which are abstract classes have two important methods which researchers must realize according to their ideas and goals.

* Method *estimate*() whose input parameters are user profile *Profile* and user rating vector *RatingVector*. Its output result is predictive (or estimated) rating value of an unknown item. Both user profile and user rating vector are stored in *Dataset*.
* Method *recommend*() whose input parameters are user profile and user rating vector. Its output result is an item which is recommended to users.



**Figure 3.1**: Abstract algorithm interfaces and classes

Ideas and features of algorithm are expressed by how methods *query* and *recommend* is implemented and what the result of their execution is.

All user-defined classes derived from *MemoryBasedCF* class or *ModelBasedCF* are also called as algorithm classes. *MemoryBasedCF-*derivedclasses representing memory-based CF algorithm use directly information in *Dataset* for recommendation task via executing *estimate*() method and *recommend*() method. Otherwise *ModelBasedCF-*derivedclasses representing model-based CF algorithm use both information and inference mechanism existing in *KBase* for recommendation task via executing *estimate*()method and *recommend*() method.

Beside two basic methods *query* and *recommend*, algorithm classes have some more extension methods:

* Method *getConfig*(): getting configuration setting of algorithm. This is the way researchers pass parameters to algorithm before it runs.
* Method *setup*() is reponsible for preparing or initializing something before algorithm runs.
* Method *createKBase*()exists only in *ModelBasedCF* class. Its role is to create knowledge base *KBase*. Each *ModelBasedCF* owns different *KBase*. So *KBase* is very flexible and its structure and complexity depends researchers’ design and implementation.

As discussed *KBase* is knowledge base – one of essential components in algorithm architecture. It models valuable information about *Dataset* and model-based CF algorithm uses it to perform recommendation task instead of using directly *Dataset*.



**Figure 3.2**: Knowledge base

Although the model of knowledge base is very flexible so as to generalize about various internal data structures of model-based CF algorithms, its specification has six methods:

* Methods *load*()and *save*() are used to read knowledge base from storage system (files, database) and write knowledge base to storage system, respectively.
* Method *learn*() is responsible for creating knowledge base from *Dataset* which is its first input parameter. Researchers can pass auxiliary settings to algorithm via second input parameter.
* Methods *clear*() and *isEmpty*() are responsible for cleaning out knowledge base and checking whether knowledge base is empty or not, respectively.
* Methods *setStoreUri*() is used to set where to store knowledge base. The storage of *KBase* is specified through uniform resource identifier (URI) which is access point of any resource over internet.

Some methods of abstract algorithm always use *KBase* are *setup*(), *query*() and *recommend*().

Dataset is composed of a rating matrix and user profiles. 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. User profile is transcript of personal information such as demographic information, user interests… User profile is represented by *Profile* class. Both rating matrix and user profiles are stored in *map* form which is a data structure in which each rating vector or user profile is identified by an integer number so-called key or identifier.Datasetaccessoperatorssuch as reading rating vector, writing user profile become faster via *map* data structure.



**Figure 3.3**: Dataset

*Dataset* is used directly by memory-based CF algorithms. Some methods of such algorithm always use *Dataset* are *setup*, *query* and *recommend*. *KBase* is created based on *Dataset*. *KBase* is also considered as essential model which is extracted or mined from *Dataset*. Dataset has a lot of utility method but there are four important methods as below:

* Methods *getUserRating*() and *getUserProfile*() are used to retrieve user rating vector and user profile, respectively. Access time is instant with computational complexity *O*(*1*) because of using *map* data structure.
* Methods *getUserIds*() and *getItemIds*() allow us to get a set of user identifiers and a set of item identifiers, respectively.

**4. Abstract evaluator and evaluation measures**

*Evaluator* configures and feeds *Dataset* to *Algorithm* because *Algorithm* requires *Dataset* to perform recommend task. After that *Evaluator* activates two methods of *Algorithm*, namely *estimate*() and *recommend*(). *Evaluator* is the bridge between *Dataset* and *Algorithm* and it has four roles:

* Firstly, it is a loader which loads and configures *Dataset*.
* Secondly, it is an analyzer which analyzes and translates the result of algorithm execution into the form of evaluation measures (or measures, briefly). Note that the execution result is the output of *estimate*() or *recommend*() method.
* Thirdly, it is a listener. If external applications require receiving result from *Evaluator*, they need to register with it.
* Finally, whenever it finishes a call of *estimate*() or *recommend*() method, it issues an event so-called evaluation event and send back evaluation measures to external applications after executing algorithm. So it is also a provider.



**Figure 4.1**: *Evaluator* class

Evaluator has four important methods:

* Method *evaluate*() performs the main task of *Evaluator* which is to load *Dataset* and to activate *Algorithm*::*estimate*() or *Algorithm*::*recommend*() method on such *Dataset*. Note that *evaluate*() method do two tasks: loading *Dataset* and activating *estimate*() or *recommend*().
* Method *analyze*() is responsible for analyzing the result returned by method *estimate*() *or recommend*()*.* So such result is translated into evaluation measures (or measures, briefly), based on pre-defined criterions. Measures are used to evaluate CF algorithms and are discussed later.
* Method *issue*() issues an evaluation event and sends back such measures to external applications.
* Method *addListener*() allows external application to register with Evaluator as listener so as to receive evaluation events and measures.

Evaluator has two derived classes *EstimateEvaluator* and *RecommendEvaluator* for specialized tasks:

* Class *EstimateEvaluator* is dedicated to configure dataset and to call *estimate*() method of *Algorithm*; after that it receives and analyzes result from such method.
* Otherwise class *RecommendEvaluator* is allocated to co-operate with *recommend*() method of *Algorithm*.

There are four measures to evaluate CF algorithms:

1. Time measure (*T*) represents the speed of algorithm and so it is the time in seconds which method *estimate*() or *recommend*() executes over *Dataset*.
2. Precise measure (P) tells us the precision of execution of *estimate*() method over *Dataset*. It is the inverse of deviation between estimated values which *estimate*() method delivers and the real ratings in *Dataset*.
3. Recall measure (*R*) indicates what percentage of missing value items which *estimate*() method can predict over dataset.
4. Usefulness measure (*U*) is the ratio of the number of interesting items to the number of best items. Best item is defined as the item which algorithm recommends to user and its rating value is highest. Interesting item is the item on which user rates with highest value. This measure is proposed because there is in case that some items are given to user and they are predicted by *estimate*() or *recommend*() method with high consistency and high precision but they are not interesting items for user. The usefulness indicator measures the level of user interest on such recommended items

These four measures are generalized by a mono-class so-called *Measures* class. This class is created by *analyze*() method. It has four properties:

* *Time* property represents time measure
* *Precise* property represents precise measure
* *Recall* property represents recall measure
* *Usefulness* property represents recall measure



**Figure 4.2**: *Measures* class

*Measures* class is the final result of algorithm execution. In other words, there are two kinds of result.

* Rough result is represented by the output of *Algorithm*::*estimate*() or *Algorithm*::*recommend*()
* Fine-grain result is represented by *Measures* class which is the output of *Evaluator*::*analyze*().

If external applications want to receive this class, they need to register with *Evaluator* by calling *Evaluator*::*addListener*() method. The execution process has five stages:

1. *Evaluator* calls *Evaluator*::*evaluate*() method to load and feed Dataset to *Algorithm*
2. *Algorithm*::*estimate*() or *Algorithm*::*recommend*() method is activated via calling *Evaluator*::*evaluate*() method to perform recommendation task.
3. *Evaluator*::*analyze*() analyzes the results returned by *Algorithm*::*estimate*() or *Algorithm*::*recommend*() and translates such results into *Measures* class.
4. External applications register with *Evaluator*  by calling *Evaluator*::*addListener*()
5. *Evaluator*::*issue*() method sends *Measures* to external applications.

Following figures is the view of such five stages

*Algorithm*

*Dataset*

1. load

2. *estimate*() or *recommend*()

*Evaluator*

4. *addListener*()

3. *analyze*()

5. *issue*()

External applications

*Measures*

**Figure 4.3**: Execution process

**5. Discussion and conclusion**

In conclusion, the general framework has *4* basic components: *Algorithm, KBase, Dataset* and *Evaluator* and an extra component *Measures*. Each component has individual 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 interactive and independent modules.
* *Flexibility*: the framework provides the high level of customization to researchers so that they realize easily their ideas. For instance, the *KBase* component has no structure and no shape; its manifest consists of abstract methods or rules so that researchers implement such rules. Almost components are abstract unit; so this architecture is called abstract architecture.

In CF domain, there is the problem that experimental rating databases such as Movielens, Jester Joke, Book Crossing, etc are heterogeneous; so their structures are very different. This problem makes researchers into trouble; they cannot focus on their creative ideas. The *Dataset* component gives the solution to this problem when it proposes an abstract of heterogeneous rating database. Researchers don’t need to consider what *Dataset* is (Movielens, Jester Joke or Book Crossing) and how to read it from file. The infrastructure is responsible for building up dataset from physical devices. What researchers do is to use methods of *Dataset*. This report only outlines *Dataset* with four methods *getUserRating*(), *getUserProfile*(), *getUserIds*() and *getItemIds*() although class *Dataset* has more utility methods that give support to researchers.

Finally, in this abstract architecture, we aim to normalize *4* basic components (or classes) as *4* CF standards. Such standards are used for software engineering, not for storage system. 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. Such methods are functions of *KBase* and they don’t specify how to load or store *KBase*. In other words, they are rules with which the infrastructure must comply.
* In similarly, method *learn*() tell us that *KBase* can be learned or built up by any approaches, for example: machine learning, data mining, artificial intelligence, statistics, etc.

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

**Kiến trúc trừu tượng cho giải thuật lọc cộng tác**

Nguyễn Phước Lộc

Công ty TNHH MTV Lập trình Hướng Dương, Hồ Chí Minh city, Việt Nam