OdysseusRecSys: Collaborative Filtering based on a Data Stream Management System

Cornelius A. Ludmann, Marco Grawunder, and H.-Jürgen Appelrath

University of Oldenburg, Department of Computer Science
Escherweg 2, 26121 Oldenburg, Germany
`{cornelius.ludmann,marco.grawunder,appelrath}@uni-oldenburg.de`

Abstract. The development of algorithms for online Collaborative Filtering (CF) in the past few years enables to add new rating data to existing models. The Recommender System (RecSys) task changes from calculating recommendations from a static and finite dataset to continuously processing rating data. Instead of using stream processing frameworks to implement CF algorithms, we present a prototype that extends the open source Data Stream Management System (DSMS) *Odysseus* in a generic and domain-independent way. The user can build a custom RecSys that benefits from existing DSMS features by defining a continuous query with a declarative query language.

Keywords: Collaborative filtering, data stream management system.

1 Introduction

In recent years, a lot of papers about so-called *online* or *incremental* algorithms for model-based Collaborative Filtering (CF) have been published (e. g., [3, 5]). With these algorithms, new learning data can be added to an existing CF model without rebuilding it from scratch. The CF task changes from learning a model from a static and finite dataset to processing a continuous sequence of ratings.

In this paper, we introduce a reference architecture and a prototype of a Recommender System (RecSys) based on the open source Data Stream Management System (DSMS) *Odysseus*[1] [1]. Similar to Database Management Systems (DBMSs), DSMSs provide a set of operators that are used to build queries. This includes operators for a stream-based variant of the relational algebra, operators to find patterns in data streams, operators to bind and parse different data stream sources and sinks etc. The user writes queries by the use of a declarative query language. Then, the DSMS parses the query and transforms it into a logical query plan. This plan can be optimized (e. g., by changing the operator order) and is translated into a physical query plan. By this, for each logical operator the DSMS chooses a physical operator with an appropriate implementation. In contrast to one-time queries of a DBMS, a continuous query of a DSMS is deployed once and produces results until the query is removed.

[1] `http://odysseus.informatik.uni-oldenburg.de/`

Instead of using a developer framework for stream-based systems like Apache Storm, the usage of a generic and application-independent DSMS as a basis for a stream-based RecSys has the following advantages:

- Once the CF feature is implemented in the DSMS, users can use this function in their queries. They can build a customized RecSys without the need of writing code.
- A lot of operations needed for a RecSys can be covered by existing operators. This includes the access to data sources and sinks, the calculation and aggregation of values like the model error for the RecSys evaluation, the choice of the top-K recommendations etc.
- A RecSys can be enhanced by adding additional operators to the query. This can be useful to combine different data sources, to pre- and post-process data (e. g., normalization of input data), to include and process additional data (e. g., context data or open linked data), to combine different model learners to an ensemble learner etc.
- The RecSys developer can benefit from many DSMS features like query plan optimization, query sharing, fragmentation and distribution of queries, scheduling etc.

2 Architecture Overview of OdysseusRecSys

A stream-based RecSys interacts with its environment as follows: First, the user application transmits rating data[2] (u, i, r) for a user u that rates an item i with rating r. Second, the user application transmits requests for recommendations for a user u (e. g., when the user opens the recommendations page). Third, the RecSys sends sets of recommendations back to the user application. Additionally, the input of a RecSys can be extended by a user-feedback. In the following, we expect the feedback in the form of new rating data. Furthermore, a continuous evaluation of a RecSys leads to an output of a model error. This can be plotted in a dashboard to monitor the RecSys or written to a file.

To write queries for CF, we added CF-specific logical operators to achieve an additional abstraction level. Figure 1 gives an overview of these operators and their combination in a simple logical query plan. On the left we see the incoming and on the right the outgoing data streams as described above. The logical operators and the intermediate data flow (solid lines with arrows) are depicted in between. The CF operators are distinguished between learning, recommending, and evaluating operators. The learning operators get learning tuples and build models for the recommendation and evaluation. The recommending operators use the models to generate a list of recommendations every time a request for recommendations arrives. The evaluating operators get test data and models to calculate a stream of error values for the predicted ratings of the test tuples.

[2] In the following, we assume either an explicit rating or an implicit rating that indicates a usage (e. g., binary rating). Alternatively, our concept can be extended to calculate implicit ratings by the usage behaviour of the users.

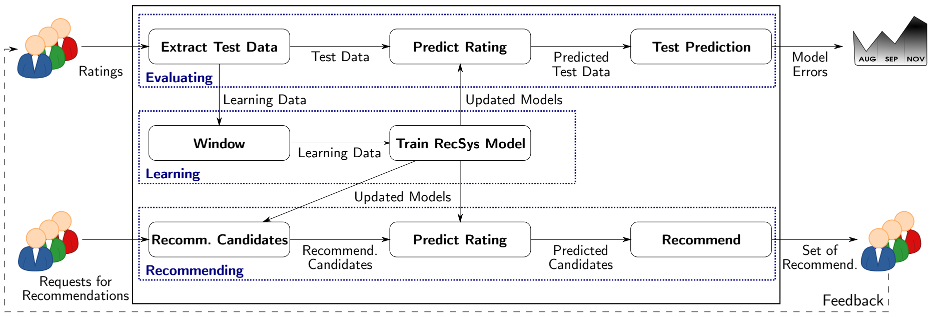


Fig. 1. Reference Architecture (Logical View) of a RecSys based on DSMS Operators

This generic composition can be used as a template and can be adapted and extended for the particular application. The logical CF operators are:

Window: We use a standard time-based window operator that allows to limit the validity of the rating data. This can be useful to learn a model that focuses only on the newest ratings and removes outdated ratings (to handle concept drifts). Additionally, it limits the ratings that need to be held in memory. This operator has to be removed if all ratings should be held in memory forever.

Train RecSys Model: This operator uses rating data to build a CF model. It outputs updated models as data stream elements when new rating data arrives. The learning algorithms can respect temporal dynamics. This operator can use different learning algorithms configured by parameters.

Recomm. Candidates: For each request for recommendations of a user, this operator determines a set of recommendation candidates. These are usually all items that have not been rated by the user.

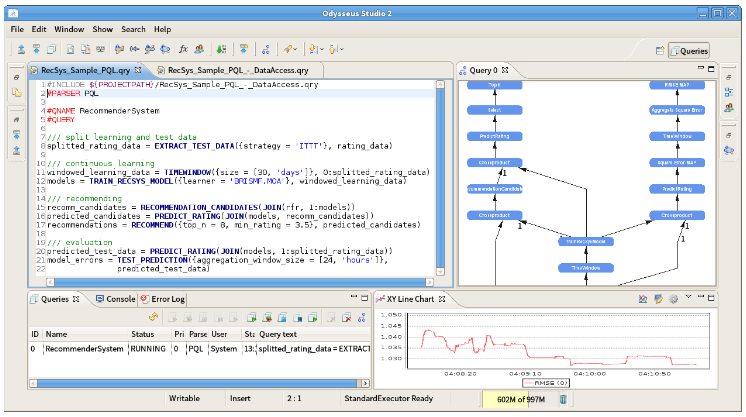
Predict Rating: This operator uses the models to predict the rating score for each recommendation candidate resp. for each test tuple. It ensures a temporal matching of models and recommendation candidates resp. test tuples. This allows a deterministic matching of models.

Recommend: This operator chooses the items that should be recommended. These are usually the top-K items.

Extract Test Data: To evaluate a CF algorithm, this operator outputs test data. A simple implementation routes 10 % of the learning data as test data (hold-out). An alternative is *Interleaved Test-Then-Train* (ITTT) [4].

Test Prediction: This operator implements an evaluation metric, e. g., RMSE. It compares the predicted and the true rating resp. the predicted and the true ranking position and aggregates an overall or moving average.

Odysseus is designed to be modular and extendable. It supports an editor for the SQL-like query language CQL [2], the functional query language PQL [1], and a graphical query editor, as well as a dashboard to visualize the data and a GUI to control the DSMS. Additionally, developers can add new (domain specific) languages, new operators, new data source and sink connectors, and new dashboard parts by the use of the Odysseus framework.



```
#INCLUDE ${PROJECTPATH}/RecSys_Sample_PQL_-_DataAccess.qry
#PARSER PQL

#QNAME RecommenderSystem
#QUERY

/// split learning and test data
splitted_rating_data = EXTRACT_TEST_DATA({strategy = 'ITTT'}, rating_data)

/// continuous learning
windowed_learning_data = TIMEWINDOW({size = [30, 'days']}, 0:splitted_rating_data)
models = TRAIN_RECSYS_MODEL({learner = 'BRISMF.MOA'}, windowed_learning_data)

/// recommending
recomm_candidates = RECOMMENDATION_CANDIDATES(JOIN(rfr, 1:models))
predicted_candidates = PREDICT_RATING(JOIN(models, recomm_candidates))
recommendations = RECOMMEND({top_n = 8, min_rating = 3.5}, predicted_candidates)

/// evaluation
predicted_test_data = PREDICT_RATING(JOIN(models, 1:splitted_rating_data))
model_errors = TEST_PREDICTION({aggregation_window_size = [24, 'hours']},
               predicted_test_data)
```

ID	Name	Status	Pri	Parse	User	Sta	Query text
0	RecommenderSystem	RUNNING	0	PQL	System	13:	splitted_rating_data = EXTRACT

Fig. 2. Screenshot of the Prototype *OdysseusRecSys* (with query editor, generated physical query plan, deployed and running RecSys query, and plot of the RMSE)

We added the operators to Odysseus (Fig. 2), which allows to use them in arbitrary queries. New transformation rules translate them to *existing* physical operators. Exceptions are new physical operators (1) for model learning that implements arbitrary learning algorithms, (2) to determine the recommendation candidates, and (3) to predict rating scores by the use of the RecSys models.

3 Preliminary Evaluations

We evaluated the feasibility by comparing the RMSE for the MovieLens dataset with the results of the stream mining framework *Massive Online Analysis* (MOA). To make our results comparable, we added the MOA implementation of the BRISMF algorithm [5], removed the window operator, and calculated the overall RMSE after each tested tuple with ITTT. The error values are exactly the same as those of MOA, which shows that the matching of learning data, models, and test data as well as the implementation of the evaluation operate correctly.

References

1. Appelrath, H.J., Geesen, D., Grawunder, M., Michelsen, T., Nicklas, D.: Odysseus: A highly customizable framework for creating efficient event stream management systems. In: DEBS'12. pp. 367–368. ACM (2012)
2. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. The VLDB Journal 15(2), 121–142 (2006)
3. Diaz-Aviles, E., Drumond, L., Schmidt-Thieme, L., Nejdl, W.: Real-time top-n recommendation in social streams. In: ACM RecSys. pp. 59–66. ACM (2012)
4. Gama, J., Zliobaite, I., Biefet, A., Pechenizkiy, M., Bouchachia, A.: A Survey on Concept Drift Adaptation. ACM Comp. Surveys 1(1) (2013)
5. Takács, G., Pilászy, I., Németh, B., Tikk, D.: Scalable collaborative filtering approaches for large recommender systems. JMLR 10, 623–656 (2009)