Chapter 7

A Generic Recommender Server

7.1 Introduction

Recommender systems have been widely applied in various domains, *e.g.*, e-commerce, proactive information retrieval, personalized search, online entertainment etc. The core problem of recommendation is selecting and presenting items from a usually large item space, for which many effective algorithms and interfaces have been designed. Software tools, libraries, systems also have been developed for building recommender systems in various applications. Recommender system research has reached a tipping point where user-centric and algorithmic research can be closely combined to improve the user experience of recommender systems. However, we need new software tools to better support this type of work. The key to provide support is to accelerate the process of going from modeling and experimenting in an offline setting to online environments where users are interacting with the design of the system in real-time. We designed and developed a server framework to fulfill this goal. This chapter focuses on answering the following questions: "what is going from offline to online for recommender systems, why we want to go from offline to online and how?"

What is going from offline to online in recommender systems? Offline research or work suggests that it does not involve a regularly running system in production used by people. They might rely on a snapshot of data collected from online systems. In this setting, algorithms designed are typically for batch processing, not for real-time responding to user interactions. Online environment however involves a constantly

running system that deals with user requests and interactions in real-time whenever a user visits. Two aspects of a research can go from offline to online: *design* and *evaluation*. In the design aspect, it means designing for the user-system interaction process, going beyond historical statistical assumptions, being explicit in modeling user state changes with the environment. In the evaluation aspect, it means having the goal of user experience in mind, evaluating interfaces, model and algorithm design in front of people, answering questions of how these manipulations affect people in both lower-level perception, higher-level cognition and more broadly user personal development and social welfare.

Why do we want to go from offline to online in recommender systems? The reasons of going from offline to online go into three perspectives. One is from the perspective of studying the theories of human psychology and behavior. Recommender systems are designed for people to use to accomplish certain goals or tasks. The research of recommender systems largely involves people using the system and how we design computational models or algorithms to describe and support these people's tasks. From this perspective, a recommender system is an artifact, stimuli or a way of manipulation through which we study people. In this case, we have to put our system in front of people for them to use.

Another perspective starts from the limitation of machine learning theories. Simply treating recommendation problems as statistical learning problems may not be the right approach because there are mis-alignments between modeling assumption and the reality. The environment is dynamic which constantly involves distribution shifting. Recommender systems make real-time decisions in dynamic uncertain environments. Algorithms designed to learn from dynamic environments needs to be tested in user-facing systems to gain better ecological validity.

Lastly, a recommender system at its core is a decision-making support tool or information navigational support tool. The ultimate goal is for user experience, satisfaction and adoption of the technology. In the long run, we can better understand how recommender systems are affecting people's life, *e.g.*, answer the question of whether the algorithm or design is improving or hurting the well-being of people.

How do we go from from offline to online in recommender systems? Going from offline to online has benefits but also challenges. For example, when we think about

designing and evaluating a design in front of people, there are potentially two apparent obstacles. One obstacle is the access to real users. Another one is the cost of system implementation. In this work, we propose a generic recommender server framework to reduce the cost of implementation by extracting out the common components of recommender systems. Recommender system researchers or designers can now focus on one part and then re-use other parts. The barrier of access to users can be potentially overcome by designing user studies and recruiting people to try out new techniques (*e.g.*, from crowd-sourcing platforms, which however has limitation of generalization because the participants might not be real users of the system). We also envision that this server can become a open service that gathers and allocates application users as resources. What we hope to achieve in the long-run is a recommender system research platform where researchers and practitioners can easily plug-in their own design and get evaluation out of it from various domains of applications if these applications are hosted in the service. This work of designing a generic server is a necessary step in moving towards that direction

7.2 Related Work

There are many softwares that have been built to support building recommender systems. We categorize these softwares into four categories: *command-line tools, libraries with programming APIs, (distributed) systems* and *servers (or services)*, as shown in Table 7.1.

Some softwares implement specific models or algorithms, *e.g.*, Apex SVDFeature [21], SLIMx [8], MyMediaLite [135], LibFM [110], LibSVM [136] etc. which we categorize as command-line tools.

Some softwares provide with implementations for multiple models or algorithms and provide their capabilities through programming language APIs, *e.g.*, Lenskit for collaborative filtering based algorithms [89], scikit-learn [90] and SparkML [91] for various kinds of machine learning models. These also include many python, Java or R packages for building recommender systems. Researchers have developed general libraries to build models with complicated structures, *e.g.*, the recently popular platform TensorFlow [92]. Stan [93] also enables flexible specification of model structure and was

developed earlier than TensorFlow although it is more oriented towards offline data analysis.

Some softwares implemented generic modeling techniques or algorithms to run in clusters of machines. It could be specific models, *e.g.*, *xgboost* [108], *difacto* [137] etc. It could be general computation operators, *e.g.*, TensorFlow [92]. We categorize these softwares as *(distributed) systems*. *Prediction.io* [94] is a quite different distributed system because it is service-oriented. It has production serving and maintaining components and integrates a variety of models and algorithms through SparkML [91]. We categorize it as a predictive or recommendation server if it is used in recommender systems.

The generic server designed in this work is novel because of the following reasons (particularly compared with *Prediction.io*).

- Recommendation is different from machine learning, particularly retrieving and ranking candidates are conceptually different, additional functionalities to provide than building predictive models.
- In our server design, there is no hard separation between the event logging and the serving of predictions or recommendations in the architecture level, which is *Prediction.io*' design. This is because data processing especially feature extraction processes (described in details later) can be determined by specific recommendation models in the engine. This design enables the following two key benefits.
  - The online and offline loop is closed and there is not a hard separation between offline model building and online model serving, which helps achieve the goal of easily going from offline to online.
  - Real-world applications of recommender systems require a data processing and feature extraction framework that can flexibly take in different types of data sources and utilize them in the model. Our design better addresses this requirement as elaborated in details later.

In the following section, I describe in details the design goals, principles and specific implementations of the generic server, which we open sourced on GitHub [1] and name as *Samantha*.

[1] https://github.com/grouplens/samantha

Table 7.1: A high-level comparison of the available softwares for building recommender systems.

Software	General Domain	Software Type
Lenskit	recommendation	general tool/library
MyMediaLite	recommendation	general tool
SLIMx	recommendation	specific tool
libsvm **libFM** **Apex SVDFeature**	machine learning	specific tool
scikit-learn **Weka**	machine learning	general library
dmlc xgboost **dmlc difacto**	machine learning	distributed system
dmlc mxnet **theano** **caffee** **tensorflow**	deep(architectural/graphical) machine learning	general library distributed system
SparkML (Mahout)	machine learning	general library distributed system
Prediction.io	machine learning	server distributed system
Samantha	recommendation machine learning	general library server distributed system

7.3 The Generic Server Design

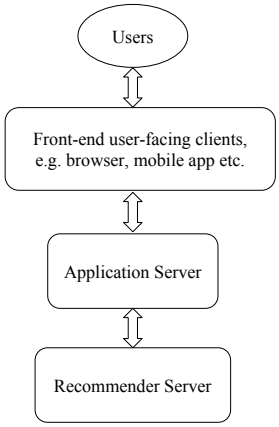

Figure 7.1: The environment that the generic recommender server is designed for.

Samantha, as a generic recommender server, is designed with the environment illustrated in Figure 7.1 in mind. It has four major parties: users, front-end user-facing clients, application servers and recommender servers. The recommender server only communicates with an application server, but not with the user-facing clients because of security reasons. Typically they communicate with each other through the HTTP

(HTTPS) protocol, *i.e.*, a recommender server is a web server similar to an application server, but specially focused on recommendation tasks. Going from offline to online means that the recommender server shipped with recommendation models and algorithms needs to respond to real-time requests and interaction feedbacks from users.

To enable this real-time recommendation serving capability, the following functionalities are necessary (this is what it takes to go from offline to online in summary for a recommender system):

- *data management*, including data storage and processing pipeline in response to user interactions or application content management
- *model management*, including online updating, building, loading, dumping and serving of the models
- *standard models and algorithms*, *e.g.*, collaborative filtering algorithms, machine learning regression or classification techniques
- *experimenting support*, *e.g.*, A/B testing, random or hashed, persistent assignment of users into different experimental or control conditions
- *real-time feedback loops*, for online machine learning and evaluation
- *extensibility for new model and algorithm design*, *e.g.*, the server can provide with parameter abstraction for model designers to freely design new computational algorithms on top of parameters without worry about where these parameters are stored and how to scale them up to large models; the server can also support parameter estimation by providing general optimization techniques or classic solvers, *e.g.*, stochastic gradient descent, to optimize for flexible objectives that come out from the model design process
- *flexible model dependency*, *e.g.*, model ensemble through boosting, bagging or stacking (*i.e.*, multiple levels of model dependency)
- *compatibility with other state-of-the-art systems*, *i.e.*, enable plugging in other implementations of recommendation or machine learning algorithms, general libraries

In this section, I describe how Samantha is designed to support all of these functionalities in a reusable, extensible and potentially scalable way so that researchers can focus on the most relevant part but still have a fully-functional system to go from offline to the online environment.

7.3.1 Recommender Components and Extensibility

In the server Samantha, *Recommender Engine* is referred to as a complete specification of how a recommender works. It supports multiple recommender engines, which is designed to scope applications. They are functionally similar to running multiple recommender servers. A recommender engine consists of eight types of components each of which can have multiple ones: *indexer*, *retriever*, *predictor*, *ranker*, *recommender*, *router*, *evaluator*, *scheduler*.

indexer. This type of component deals with data indexing in real-time when receiving data from applications. Since it knows how the data is stored, it is also responsible for outputting data from the behind storage for further batched data processing. By default, Samantha provides with multiple types of indexer implementations corresponding to different data storage back-end systems. Usually, one indexer is configured for one data type, similar to a database table, although one indexer type is general enough to take in any data types with different data fields. One can use one indexer for each data type the application sends in because it eases the integration between the recommender server and the application server. This will be elaborated in details later.

retriever. This type of component is responsible for retrieving candidates for a recommender engine. Recommender systems research typically focus on predictive model and algorithm innovation ignoring the problem of retrieving in an actual recommender system. Partially, it's because there are not many online experiments reported in details in academic and industrial research as to go into this aspect. However, when item space is big and especially when the predictive model involves complicated computation, an initial candidate generation process is a necessary component for a recommender engine. Retriever can be simple and straightforward, *e.g.*, retrieving all of the items or retrieving the top popular items. It can be complicated and critical to have good recommendations, too. For example, we can build simple machine learning models in a retriever to score all items and pick the top to generate candidates. Another possible approach is

to build fast associative models as an item-based k-nearest neighbor algorithm does, in which candidates become those most similar items to a user's previously liked items. We can also blend multiple retrievers with different priorities, *e.g.*, first using any results produced by a personalized retriever and resort to non-personalized one when necessary.

predictor. This type of component is the core part of operationalizing machine learning theories, roughly falling into the supervised learning domain. This is mainly a wrapper for a machine learning model implementation, which is essential for enabling Samantha to integrate with other machine learning libraries.

ranker. This type of component takes in an initial candidate item set and rank them based on certain criterion which could just be the output score of a predictor component.

recommender. In Samantha, a standard recommender is just the combination of a retriever and ranker, *i.e.*, a retriever retrieves initial candidates from the storage or an initial model and feeds them into ranker to generate the ranked list.

evaluator. There are two standard types of evaluators. Prediction evaluator provides the ability to compute prediction metrics for any predictor component. Recommendation evaluator evaluates any recommender component by computing top N recommendation metrics.

router. This type of component implements how to find the right recommender or predictor for a request, which is the foundation of A/B testing or between-subjects field evaluation framework.

scheduler. This type of component supports calling model management interfaces regularly with a predefined schedule, *e.g.*, for re-training a machine learning model.

Extending the capabilities of the generic server involves providing specific type of implementation for any type of recommender component that is relevant. For example, if the research is about designing new types of predictive models, it then involves implementing a predictor type after which the specification of the predictors in a recommender engine can be replaced with the new implementation but reuse the default available types of other components.

7.3.2 Server Interface, Architecture and Scalability

Samantha can be summarized in simple words as a HTTP server embedded with a recommendation framework. Before describing its interfaces and architecture, we first introduce the design of the Data Access Object (DAO) interface. Whenever the server interacts with outside data sources, it goes through a DAO implementation. All the implementations of the DAO interface summarizes the server's understanding on the possible data formats that the server can take in. When a request wants to tell the server to use a piece of data, it needs to tell the server which type of DAO is being used and how to construct that DAO with additional parameters passed in in the request.

Samantha's capabilities are provided to applications through three types of HTTP interfaces: data indexing, model management, recommendation (prediction) serving. This is the very front boundary of the server, through which the application server interacts with. Figure 7.2 illustrates the processing flow of Samantha receiving different types of requests.

Model management with parameters of component type, component name, model name, model operation. When model management requests come, Samantha first recognizes which type of component it is, *e.g.*, retriever or predictor and then finds the specific component (there could be multiple components with the same type, *e.g.*, multiple predictors). After this, the identified component will be constructed and do the actual work of managing its models. The request will also pass in DAO information if the model management task involves reading and processing a piece of data, *e.g.*, training a machine learning model with a given data source.

Data indexing with the parameters of indexer name, DAO information. If data are sent in for indexing, the server finds the specific indexer based on the indicated name in the request and ask the indexer to index data into the back-end storage system. Indexers support subscribers which means the application can ask to pass those data to other components at the same time in addition to being indexed so that other components can update themselves in real-time, *e.g.*, updating a machine learning model in a predictor according to an online optimization algorithm.

Recommendation and prediction serving with the parameters of user identification, context information. If the application server is requesting recommendations or predictions, Samantha first asks a router to identify a recommender or predictor. Then the

identified recommender or predictor is responsible for generating recommendation or prediction results. Before returning results, a wrapping process writes relevant information on the working recommender or predictor into the response in order to let the application know who generates the results (together with logging by the application, this enables analysis and comparison among different predictors or recommenders).

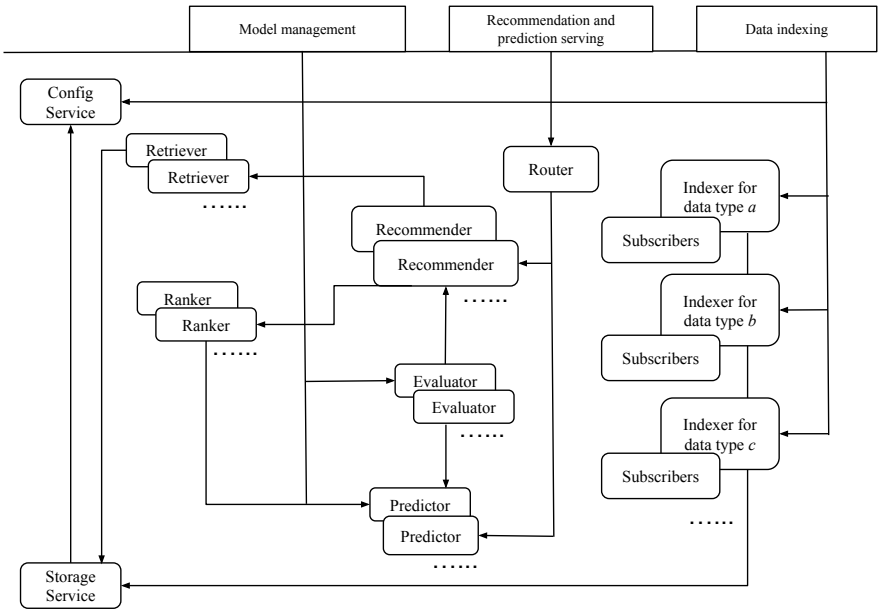

Figure 7.2: Samantha Data and Request Processing Flows. The directions of the arrows represent data flow and component dependencies.

Data processing pipeline

The data processing pipeline in Samantha consists of two concepts: *data expanders* and *feature extractors*. Data expanders are motivated by the observation that a complex model building requires data that goes beyond what one data type can offer or the raw content, interaction data sent in by the application requires future processing before being used for building models. Feature extractors are motivated by the observation that machine learning models require design matrix in which each data point is represented as a numerical vector containing the values of variables to be modeled, each of which is associated with certain parameters to estimate (*e.g.*, simple scalar coefficient or vector

embedding) depending the model design.

There are four common types of expanders: data joining, predictor based expander, filtering, (grouping and then) merging. We illustrate these expanders with an example. Imagine an application sends in a data point which is a tuple of user ID, item ID and rating. In order to build a potentially complex rating prediction model, we need to expand this data point according to how the rating prediction model is designed. For example, this model might relies on another model that tries to estimate how much the user likes tags of the item. We first need an expander that can communicate with the data storage service (*e.g.*, a relational database) for the tags of the item. This is a generic data expander because it is performing key-based query in a data service and join the search results with the data point. The next expander we need is a predictor based expander that takes in this data point and expands it with another model's predictions (assuming the user-tag preference model has been built in a predictor). If we want to exclude ratings that are out of the range [0.5, 5.0] because those might be invalid ratings or ratings designed for other purposes by the application, we need another filtering expander that filters the data point according to the criteria. If the rating prediction model actually needs all the user's rating history (*e.g.*, SVD++ [15]), having access to such individual ratings after querying the data storage service, we need a merging expander that combines all the previously rated items of the user into one data field and join it with the current data point. Grouping expanders might be needed when we are training a SVD++ prediction model (through model management requests) while the rating data provided by the DAO are individual ratings not grouped by users yet.

Even if we have all data fields available in the data point, a statistical model can not use it yet because it needs a numerical vector representation. This requires mapping a data field into an index space with which a corresponding parameter space is created. For example, user ID needs to be converted to an index that refers to its bias parameter in a vector parameter space or its latent factor parameters in a matrix parameter space for a standard matrix factorization model [18]. This is exactly what feature extractors are designed for. Each feature extractor takes in a data point and convert the relevant data field to be an index and a value (named collectively as a *feature*). For categorical

fields like user IDs, the value here is usually one, but for numerical variables (*e.g.*, predicted tag preference), the value here is the predicted preference score. After processing through a list of feature extractors, we have a numerical vector representation of a data point which can further be used by statistical models now.

Feature extractors rely on the capability of converting any string into the index of a densely organized variable or parameter space, which is referred as *index space*.

Why is this scalable?

The scalability of the server can be explained in three aspects. First, model size can go beyond what a single machine can hold utilizing distributed model training through the parameter server paradigm. Second, data sets can go beyond what a single machine supports and utilizing distributed storage system. Third, the serving of recommendation and predictions can be duplicated across multiple server clusters responding to potentially a large number of requests at the same time.

7.3.3 Using the Server

To use the server framework, what researchers and practitioners need to do now become

1. configure a recommender engine (explained below) to specify how each part should run

2. if the currently implemented state-of-the-art models or algorithms can not be used or the research itself is about new models or algorithms, design and develop new models or algorithms following the server framework

3. send in data generated by application content management or users interacting with the application front-end clients

4. ask for recommendations or predictions in real-time

A minimum recommender engine requires specifications on what data will be stored and how they are stored when data comes in in real-time, how an initial set of candidates will be retrieved and how to rank the candidates based on the prediction of a user

preference model. Assuming a standard matrix factorization model predicting user-item ratings, we illustrate below how to set up an engine in the recommender server to respond to application users in real-time.

indexers: One indexer that supports indexing user-item rating data into the server

- name: UserItemRatingIndexer
- implementation: database based indexer (*e.g.*, MySQL database based indexer)
- data fields: user ID, item ID, rating, time-stamp

retrievers: One retriever that retrieves all available items in the database store of the item data (note that the retrieved item list needs an expander to set the current user ID in the request)

- name: AllAvailableItemsRetriever
- implementation: database based retriever (*e.g.*, MySQL database based retriever)
- table: ItemData
- retrieved fields: item ID
- expanders: set the user ID data field in the request into the retrieved list of item data

predictors: One predictor that builds and maintains a matrix factorization model [18] with user, item bias terms and user, item latent factors.

- name: MatrixFactorizationRatingPredictor
- implementation: matrix factorization model based predictor
- feature extractors:
  - user bias extractor: convert user ID into the user index in the bias parameter space named as UserBias
  - item bias extractor: convert item ID into the item index in the bias parameter space named as ItemBias

  - user factor extractor: convert user ID into the user index in the latent factor parameter space named as UserFactor
  - item factor extractor: convert item ID into the item index in the latent factor parameter space named as ItemFactor

- model loss: L2-norm loss (regression)
- learning method: stochastic gradient descent

rankers: One ranker that ranks based on the predicted rating of the above rating predictor

- name: RatingPredictorBasedRanker
- implementation: a predictor based ranker which first makes predictions on a list of retrieved candidates and rank the list according to the predictions
- predictor name: MatrixFactorizationRatingPredictor

recommenders: One recommender that retrieves a candidate list and then ranks it based on the above ranker

- name: StandardRecommender
- implementation: a recommender that first retrieves a list of candidates with a retriever and ranks the candidates with a ranker
- retriever name: AllAvailableItemsRetriever
- ranker name: RatingPredictorBasedRanker

schedulers: One scheduler that trains the rating prediction model every day by taking in data from the indexed user-item ratings (note that the indexer supports both indexing, *i.e.*, data in, and outputting, *i.e.*, data out). Without this scheduler, the application server can still regularly sends in model management requests (described above section) to update the model.

- name: TrainingRatingModelScheduler

- implementation: a scheduler that mimics a model management request
- schedule: every mid-night
- request context: all available data in the indexer UserItemRatingIndexer

7.4 Case Studies

In this section I describe three case studies that demonstrate a) this server framework can easily integrate with state-of-the-art machine learning libraries and systems; b) this server framework enables complex online experiments with multi-level model dependencies or many important factors to model.

7.4.1 Extension and Integration

Integrating the generic server with other libraries or systems involves implementing a recommender component. This is exactly the same as writing its own implementation of a recommender component within the server. For example, the server provides with the implementation of a type of predictor based on the modeling technique SVDFeature [21]. When the component involves complex learning models or algorithms, the server provides support to further simplify by defining interfaces for the new component to implement. The interfaces for any recommender component that involves complex statistical machine learning models are as follows:

- *LearningInstance*: A feature list (or numerical vector) representation of a data point
- *LearningData*: An iterable set of LearningInstance representing a data set
- *LearningModel*: A complete representation of a model being able to extract features for a data point to get LearningInstance representation and make predictions on a LearningInstance
- *LearningMethod*: Train or update a LearningModel with a LearningData

TensorFlow

TensorFlow [92] is one of the most widely used machine learning libraries and systems nowadays. The powerful capabilities of TensorFlow enable not only flexible incorporation of data available in real-world applications, but also the flexible design of the modeling structure. We show here that the server framework we designed can easily integrate TensorFlow so that any recommendation models that are designed in TensorFlow with any type of computational graphs can run in the generic server leveraging the data processing pipeline, model management interfaces in Samantha.

Correspondingly, the integration takes the following implementation following the server framework (note that this integration is agnostic to the computational graphs):

- LearningInstance: A dictionary of tensors (called feed dictionary in TensorFlow) where the key is the name of the tensor defined in the computational graph and the value is the tensor representation of a data point
- LearningModel: Extract features by utilizing feature extractor interfaces and convert the list of indices and values into a TensorFlow specific LearningInstance; make predictions by feeding in the dictionary of tensors and running a specified operation in the computational graph
- LearningMethod: Iterate over a LearningData that outputs TensorFlow specific LearningInstance, feed in the dictionary of tensors and run a specified model updating operation in the computational graph (*e.g.*, a minimizer over a loss in the graph)
- predictor: When responding to prediction requests, this TensorFlow specific predictor asks the TensorFlow specific LearningModel to make predictions on the input data points. When responding to model management requests, it delegates the server to interpret a specific DAO and asks the TensorFlow specific LearningMethod to update or train the model with the DAO based LearningData representation.

Imagine that we want to run a TensorFlow graph defining a SVD++ [34] model. For simplicity of illustration, we dropped the bias terms here which then gives the following

prediction function, where U, Q and V are user, implicit action and item latent factor matrix and I is the set of items rated by user u.

$$f(u, I, a) = (U_u + \frac{1}{\sqrt[2]{|I|}} \sum_{i \in I} Q_i)^T V_a \tag{7.1}$$

The simplest definition of the a TensorFlow graph would be taking in four tensors (ignoring the bias terms for simplicity of illustration): a tensor (ImplicitFactor) with items rated by a user (note these items need to be indices referring to a matrix parameter space which suggests that the item IDs in the data sources should go through the feature extracting process in the data processing pipeline mentioned above), a tensor with the user (UserFactor, similarly an index referring to a matrix parameter space), a tensor with the target item (ItemFactor, similarly an index) to make prediction on and a tensor (Rating) with the actual rating given by the user on the item. To move this TensorFlow graph from offline to online in the generic server, we replace the predictor in section 7.3.3 with the following (note that it assumes the input data point has a data field with all the rated items by the user which can be easily achieved through the data expanders in the data processing pipeline of Samantha).

predictors: one predictor that is based on the TensorFlow predictor type

- name: TensorFlowSVDPlusPlusRatingPredictor
- implementation: the above mentioned TensorFlow specific predictor type
- graph: a file path pointing to the definition file of the TensorFlow graph
- feature extractors:
  - user factor extractor: convert user ID into the user index in the user latent factor parameter space named as UserFactor
  - item factor extractor: convert item ID into the item index in the item latent factor parameter space named as ItemFactor
  - implicit action extractor: convert a list of item IDs into the the item indices in the implicit action latent factor parameter space named as ImplicitFactor
  - rating value extractor: output the rating value as it is named as Rating

xgboost

xgboost [108] is also one of the widely used machine learning systems in various domains including recommender systems, *e.g.*, GBDT has been used in Yahoo! News recommendation [138]. Similarly, we show that xgboost can be easily integrated into the server framework and accelerate the process of moving a xgboost-based recommendation system from offline to online.

- LearningInstance: A labeled feature map from the feature index to feature value as required by xgboost
- LearningModel: Extract features by utilizing feature extractor interfaces and convert the list of indices and values into a xgboost specific LearningInstance; make predictions by getting results from a xgboost Booster
- LearningMethod: Iterate over a LearningData that outputs xgboost specific LearningInstance to create a xgboost specific data iterator and ask the xgboost library to train the xgboost LearningModel with the data iterator.
- predictor: When responding to prediction requests, this xgboost specific predictor asks the xgboost specific LearningModel to make predictions on the input data points. When responding to model management requests, it delegates the server to interpret a specific DAO and asks the xgboost specific LearningMethod to train the model with the DAO based LearningData representation.

7.4.2 Online Recommender Blending

A challenge of running recommendation models and algorithms in an online setting where application users can interact with the system any time lies at the continuous management of data and models, and dealing with complex multi-level model dependency. Samantha has been used by MovieLens to provide personalized movie recommendations [139]. Here I describe how one of the recommenders was implemented in Samantha which is based on the techniques of matrix factorization and LinearUCB [35].

The recommender has two levels of model dependency. In the first level, there are two matrix factorization models predicting rating ($f_1(u, a)$ in Equation 7.2) and

action probability ($f_2(u, a)$) respectively (where action is binary representing whether a displayed movie was interacted by the user or not). Conceptually, these two models are estimating how much a user might like a movie (predicted rating) and how likely a user might interact with a movie (predicted action probability). In the second level, there is one LinearUCB model that combines the predicted rating and action probability of a user-item pair to maximize the online interactions from users as a reward function by estimating the best set of weights for the two predictions. The label value of $r^*(u, a)$ also takes the value of either zero (when a movie is displayed but there is no action) or one (when a movie is displayed and also acted upon by the user).

$$r^*(u, a) = \beta_1 \cdot f_1(u, a) + \beta_2 \cdot f_2(u, a) \tag{7.2}$$

With already implemented predictor types supporting the matrix factorization and LinearUCB models, moving this recommender from offline to online environment does not need any additional implementation by using the following specifications in the recommender engine (continuing the minimum recommender engine example in Section 7.3.3).

indexers: One indexer that supports indexing user-item action data into the server

- name: UserItemActionIndexer
- implementation: database based indexer (*e.g.*, MySQL database based indexer)
- data fields: user ID, item ID, action, time-stamp

predictors: One predictor that builds and maintains a matrix factorization model [18] to predict action probability.

- name: MatrixFactorizationActionPredictor
- implementation: matrix factorization model based predictor
- feature extractors: the same as MatrixFactorizationRatingPredictor
- model loss: logistic loss (binary classification)
- learning method: stochastic gradient descent

Another predictor that online estimates a LinearUCB model to dynamically combine rating and action prediction models.

- name: LinearUCBActionPredictor
- implementation: LinearUCB model based predictor
- data expanders:
  - a predictor based expander that expands with the predicted rating of MatrixFactorizationRatingPredictor
  - a predictor based expander that expands with the predicted action probability of MatrixFactorizationActionPredictor
- feature extractors:
  - predicted rating value extractor: convert the predicted rating into a representation with both the blending weight index in the LinearUCB model and the predicted value
  - predicted action probability value extractor: convert the predicted action probability into a representation with both the blending weight index in the LinearUCB model and the predicted value
- learning method: LinearUCB specific learning method

schedulers: One scheduler that trains the action prediction model every day by taking in data from the indexed user-item actions.

- name: TrainingActionModelScheduler
- implementation: a scheduler that mimics a model management request
- schedule: every mid-night
- request context: all available data in the indexer UserItemActionIndexer

7.4.3 System-Level Cold-Start

One goal of Samantha is to support researchers who are interested in answering questions about how a predictive system or recommendation system affect user experience or user tasks. In this case study, we demonstrate how to set up a recommender server, with the minimum amount of implementation but without losing the flexibility of incorporating important factors in the domain, to study a new system that recommends emails to a user of an email client based on the user's calendar schedule [140]. We assume application developers have access to emails in a user's "inbox" or "sent" folder and the user's calendar schedule in the upcoming week after getting permission from the user (*i.e.*, acting as a agent for the user). Imagine that the application is a plugin for the email client that can proactively recommend potentially useful emails to the user based on the user's next upcoming meeting.

One can imagine a condition where the application provides an interface that enables real-time feedback from users so that users can tell the system whether the displayed emails are useful to the user's upcoming meeting or not (if the users want to) and the system interactively evolves by learning from this feedback. Alternatively, the system can directly learns from implicit user interactions, *e.g.*, treating hovering on or clicking the recommended email as positive signals. A particular reason that we want the system to learn in real-time is that the system does not have any usefulness feedback (neither explicit usefulness judgments or implicit interaction). The specification below following the recommender server framework we designed illustrates that it only takes minimum customization in the feature extraction process to have a production-ready recommender system for the project. All we need is a predictor specification similarly based on the LinearUCB model (other predictor types can be used as long as the predictor support online model learning or updating). Note that it assumes the input data sent by the email client (proxied by a secured application server) has four data fields for each data point: people involved in the email and meeting, content of the email and the meeting, *i.e.*, the application is asking (in the phase of requesting predictions) and telling (in the phase of training the model) the recommender how much useful a list of emails might be to a meeting. The researcher might hypothesize the content similarity between the email and meeting and the overlap of the sets of people involved are potentially important factors.

predictors: One predictor that online estimates a LinearUCB model to dynamically combine multiple factors that might be important for predicting the usefulness of emails to a meeting.

- name: LinearUCBEmailMeetingUsefulnessPredictor
- implementation: LinearUCB model based predictor type
- feature extractors:
  - email-meeting content similarity extractor: compute the similarity between the content of the email and meeting, set the extracted feature to be the parameter index of this factor and the computed similarity value
  - email-meeting people overlap extractor: compute the overlap between the sets of people involved in the email and meeting, set the extracted feature to be the parameter index of this factor and the computed overlap value

If the research develops customized complex modeling technique for predicting the usefulness of an email to a meeting, *e.g.*, implementing natively in the server or designing computational graphs in TensorFlow, the process of having this new technique take effects in front of users (or recruited participants) to get user feedback or user-centric evaluation can still be accelerated by using some of the components of the recommender server.

7.5 Discussion

In this work, we propose that recommender system research and practice can benefit from going from offline to online environments because it better connects the user-centric research and offline algorithmic and modeling research in recommender systems. In the long run, it can help answer the important questions of how recommendation technologies affect user perception, experience, satisfaction and people's life. We demonstrate that we can support this type of work by designing generic server frameworks. We provide an example framework along with case studies to show that this framework can support real world applications and potentially fulfill the goal of accelerating going from offline to online.

We recognize that not all research can benefit from following this approach. For example, computational questions for specific recommendation algorithms might be well answered with offline data experiments. Similarly, questions regarding user factors that are independent of algorithmic manipulation can be well answered by designed experimental tasks for users to accomplish without involving field usage of a system. For either type of work, following this server framework might incur significant overhead because it introduces constraints or additional work (*e.g.*, working with and maintaining a server environment). Overall, this work makes contribution by supporting an important thread of research in recommender systems where complex modeling and algorithms are combined with user-centric design and evaluation.