Orchestration of ML-Based Recommendation Systems

Vivek Kumar Singh
Dept. of CSE,
Global Academy of Technology,
Bengaluru-98
viveklp99@gmail.com

Shruthi E Karnam
Dept. of CSE,
Global Academy of Technology,
Bengaluru-98
shrutiek523@gmail.com

Bhagyashri R Hanji
Dept. of CSE,
Global Academy of Technology,
Bengaluru-98
bhagyashri@gat.ac.in

Abstract:
Many e-commerce websites use recommendation systems to recommend products to users to boost sales and user experience. These recommendations do not always come from the same recommendation engine. Websites can use multiple recommender models that use different machine learning algorithms and neural networks to compute these recommendations. There arises a need for a machine learning pipeline that will help orchestrate all the steps required to compute and display recommendations.
The pipeline handles training a model using content-based approach, storing it with required metadata, loading it, precomputing recommendations, collecting user metrics, analysing the metrics and retraining the models with updated hyperparameters if required. Without a pipeline to automate and streamline the process, much of the work must be done manually.
Keywords:
Recommendation systems, Collaborative filtering, Content-based filtering, Hybrid recommendation systems, Cold start, Scalability, Sparsity, Synonyms, Shilling attacks, Grey sheep, Black sheep, Longtail, Pipeline

Introduction:
Machine learning is commonly used to build recommendation engines that recommend movies, create playlists, recommend products and advertisements on sites like Instagram, Netflix, Spotify and Amazon. An e-commerce website usually has a product recommendation system that recommends products to customers. The aim of these recommenders is to expose users to niche and diverse products that are typically not available in brick and mortar stores.
Without recommender systems, users would have to go through tons of products in a category manually and decide what to buy. Although having more options may seem appealing to a user, it has been proven to yield a very low purchase rate and usually leaves people undecided and confused.
Recommender systems give users a starting point in their search for a product that suits their needs. The most common approaches to building a recommender system are collaborative filtering and content based filtering. Content based filtering uses keywords to describe an item and recommends products with similar description and features. This helps us overcome some of the shortcomings such as cold start, scalability, sparsity of user data, shilling attacks, grey sheep, black sheep and long tail.
Websites that use recommendation systems often have a monolithic code that performs data acquisition, model training, deployment, computing recommendations and collecting and analyzing metrics. This requires a lot of maintenance and human supervision. Machine Learning Pipelines are used to automate machine learning workflows. It splits up the ML workflows into independent, reusable and modular parts that can be pipelined to make the system more efficient and to reduce redundant work.

Literature Survey:

Jieun Son and Seoung Bum Kim in their paper "Content-based filtering for recommendation systems using multi attribute networks" [1] discuss collaborative filtering techniques and their many advantages and drawbacks. In their study they have proposed a type of CBF that uses a multi attribute network (MN), which comprises entire attribute information for different items. In the network analysis, they measure the similarities between directly and indirectly linked items. This approach has shown to address the sparsity problem and the over-specialization problem that frequently affect recommender systems. Furthermore, their proposed method depends only on ratings data obtained from a user's own past information, and so it is not affected by the cold start problem that is prevalent in CF.
Schröder, Gunnar, Maik Thiele, and Wolfgang Lehner in their research titled "Setting goals and choosing metrics for recommender system evaluations" [2] have developed a framework for the evaluation of recommender systems. Evaluation metrics for recommender systems can be divided into four major classes: 1) Predictive accuracy metrics,

2) Classification accuracy metrics, 3) Rank accuracy metrics and 4) Non-accuracy metrics. Predictive accuracy or rating prediction metrics embark on the question of how close the ratings estimated by a recommender are to the true user ratings. Classification accuracy metrics try to assess the successful decision-making capacity (SDMC) of recommendation algorithms. A rank accuracy or ranking prediction metric measures the ability of a recommender to estimate the correct order of items concerning the user's preference, which is called the measurement of rank correlation in statistics.

Milano, Silvia, Mariarosaria Taddeo, and Luciano Floridi in their article "Recommender systems and their ethical challenges." [3] identify the six areas of ethical concern while collecting user information, and map them onto a proposed taxonomy of different kinds of ethical impact. They stress on the importance of identifying the ethical issues, they need to be understanding and addressing them by evaluating the design, deployment and use of the recommender systems, and the trade-offs between the different interests at stake. A failure to do so may lead to opportunity costs as well as problems that could otherwise be mitigated or avoided altogether, and, in turn, to public distrust and backlash against the use of RS in general.

Van Meteren, Robin, and Maarten Van Someren in their paper "Using content-based filtering for recommendation." [4] discuss the reasons to use content-based filtering and their drawbacks in comparison with collaborative filtering. The test results indicate that on average, slightly more than one out of two of the suggestions that PRES makes is relevant. The results are negatively influenced by the fact that the same concept can usually be described with several terms and many terms have more than one meaning. This makes the user profile less accurate. The effectiveness of PRES can therefore be further improved if content-based and collaborative filtering would be combined.

Proposed Work:

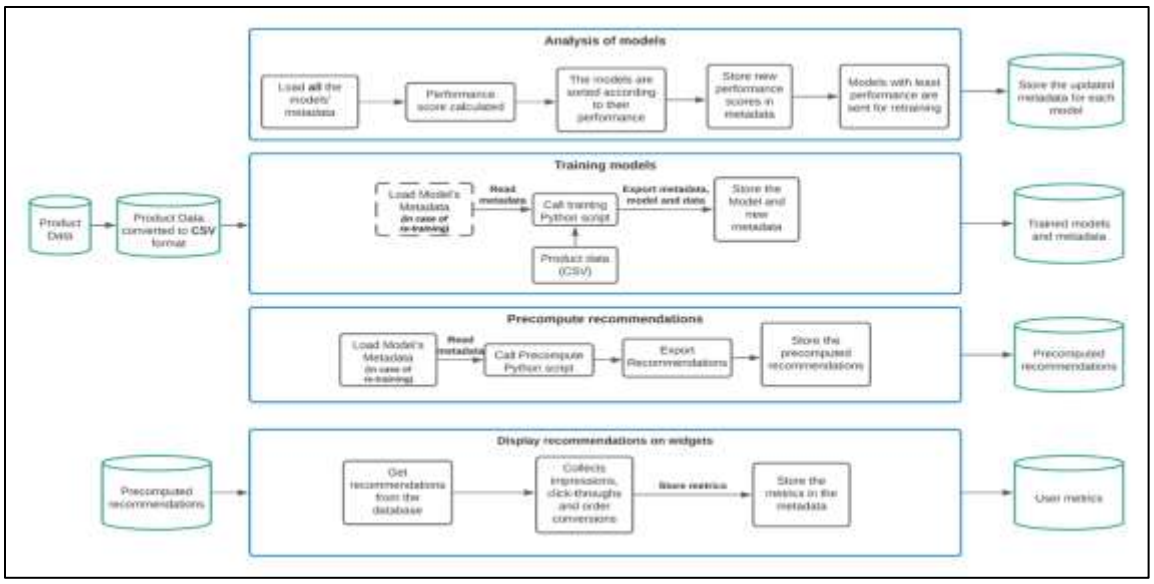


Fig 1. System Architecture

The system we are proposing will be using 2 content-based recommendation systems which will recommend products based on their description. The first recommendation system uses TFIDF algorithm and Cosine similarity to find similar products based on the frequency of the words used in the product descriptions. The second recommendation system uses K-Means to cluster similar products together and recommends products within the same cluster.

Our main focus, however, is on the pipeline which manages the various product recommendation models. The pipeline trains the models, stores them with the required metadata, loads them to pre compute recommendations, displays the recommendations on widgets, implicitly collects metrics about the performance of these models from the user and periodically analyses the metrics to improve the performance of the models. The metrics are stored and analyzed periodically. The models are retrained with new hyper parameters if required with a new version number.

The older versions of the models are stored so that the system can revert to them in case the new models fail. This pipeline helps with the automation of the machine learning workflow. The pipeline needs minimum to no human interaction making the whole system highly scalable. The pipeline orchestrates the entire machine learning workflow and is self-sustaining.
The Cron job triggers the operations on the pipeline on a periodic(daily/weekly/monthly) basis. The first module analyses the performance of the models. The metadata of the models stored in a .json file are loaded. Using the metrics stored in the file, the performance score is calculated for each model. The models are then sorted according to their performance. The models that have the least performance are sent for retraining. The metadata file is updated with the performance score.
The second module takes care of training the models. The product data is stored in PostgreSQL. The training data is usually a subset of the product data. It is present in the form of a .csv file. The model's metadata is also loaded as it contains the location of the training scripts. The training scripts (python scripts) are executed and the model is retrained. The new model is also stored with its metadata for future use.
The module handles precomputing the recommendations. The model's metadata is loaded as it contains the location of the python precomputing script. The precomputing script is run and the database (MongoDB) is populated with the product Id's and their recommendations.
When the user logs in and searches for a product, the recommendations for that product ID is displayed on the screen after fetching them from the MongoDB database. Metrics such as impressions, order conversions and click-throughs are recorded and updated in the model's database.
The browser will fetch related product recommendations from the server via an API. The API provides 'n' number of recommendations based on the number of widgets on the webpage. Based on the user interaction with the webpage, user metrics are collected and stored.
Some metrics collected are:
How many times did the user click on the recommendation?
How many times did it result in an order (order conversion)?
How many times did the user see the recommendation (impressions)?

Periodically, these recommenders must be tuned. A cron job is used to invoke the orchestrator, which first analyses the models by calculating the performance score of each model based on the metrics that were collected. The performance score is calculated as follows:

1. Calculate click-through rate.

$$click\ through\ rate\ = \frac{click\ through\ count}{impression\ count}$$

2. Calculate order conversion rate.

$$order\ conversion\ rate\ = \frac{order\ conversion\ count}{click\ through\ count}$$

3. Calculate the score.

$$score = (click\ through\ rate * weight_1) + (order\ conversion\ rate\ * weight_2)$$

$weight_1$ and $weight_2$ are used to normalize the score.

4. Repeat for each model.

The performance score is used to compare models against each other to determine which models must be retrained and which models can be used without any tuning. Models are then tuned and retrained with the custom tuning handler that the developer/user has to implement.
The retrained models are then passed to the orchestrator so that the new recommendations can be computed again.

Results:
When the user clicks on a product, similar products are fetched based on the recommendations in the database. A recommendation service takes the product ID and the number of recommendations required and queries the database for that product ID. The user metrics are collected and stored as a part of the metadata for the model. Some of the metrics collected are impressions (how many times did the users see a product from this model), click-through(how

many times a user clicked on a recommendation) and order conversion (how many times the user ordered a recommended product).
The metrics stored in the metadata are used to calculate each model's performance score. The performance score is calculated using the click through rate and the order conversion rate. The model with the lowest performance score is sent to be retrained. The retraining script calls a python script and loads the training dataset based on the file storage location present in the metadata. Once the training is completed, the pipeline exports the model and the data, creating a new model and metadata for the model.
The python scripts for precomputing the recommendations are called by the orchestrator. The script loads the model and the required data along with a list of product IDs. It generates the recommendations for each product ID and stores them in a .json file. The node.js script reads the .json file and populates the database with the recommendations computed by the model.

This section describes the screens of the "Orchestration of ML based Recommender Systems". The snapshots are shown below for each module.

Snapshot1: Module 1 - Raw data returned from the database

The DAO layers return raw data from the database based on the query parameters and attributes. The data is stored in PostgreSQL and MongoDB. Both databases return data in the same format. The picture below shows product details of the recommended products being returned based on product ID.

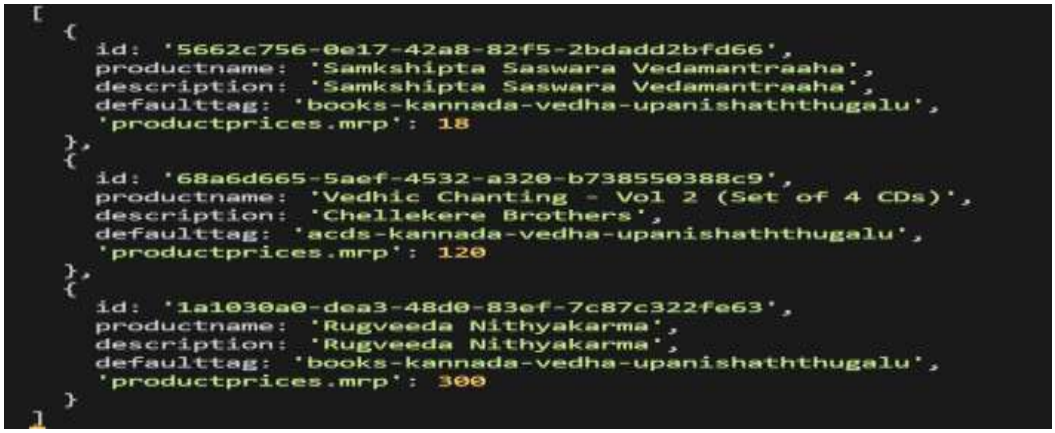


Fig 2. Snapshot of raw data returned from the database

Snapshot 2: Module 2 – Handler for models

The handler for the trained ML models is used to precompute the recommendations and store them in the database and also re-train them when needed. The below screenshot shows how the precomputations are stored in MongoDB. The product IDs act like an index and the recommendations are grouped by the model that generated them.

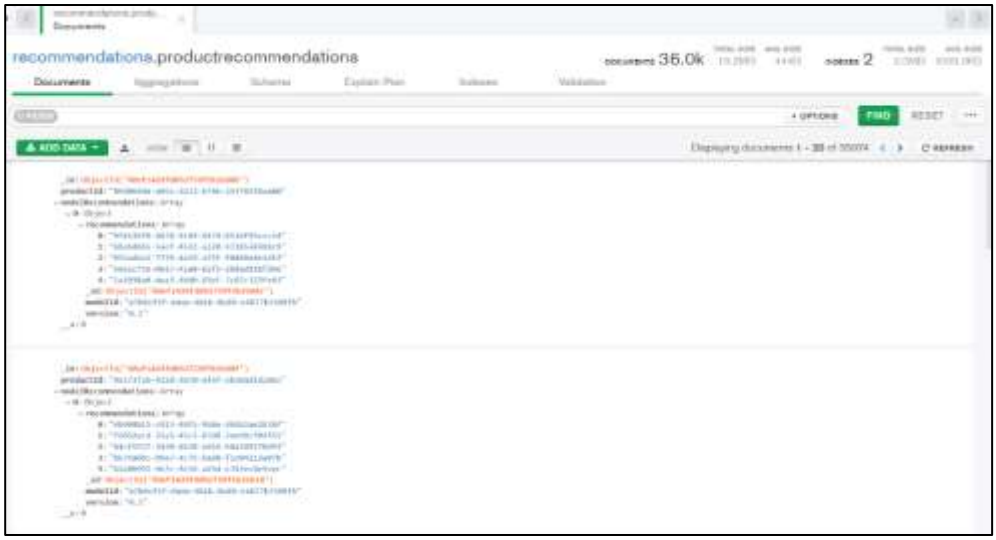


Fig 3. Snapshot of Database storing recommendations

Snapshot 3: Module 3 – Fetch Recommendation

The pre computed are stored in MongoDB.
When the user clicks on a product, similar products are fetched based on the recommendations in the database. A recommendation service takes the product ID and the number of recommendations required and queries the database for that product ID.
The below snapshot shows the recommendations for Murugan Songs, based on similar description. The similar products include Instrumental music, saxophone CDs, classical veena.

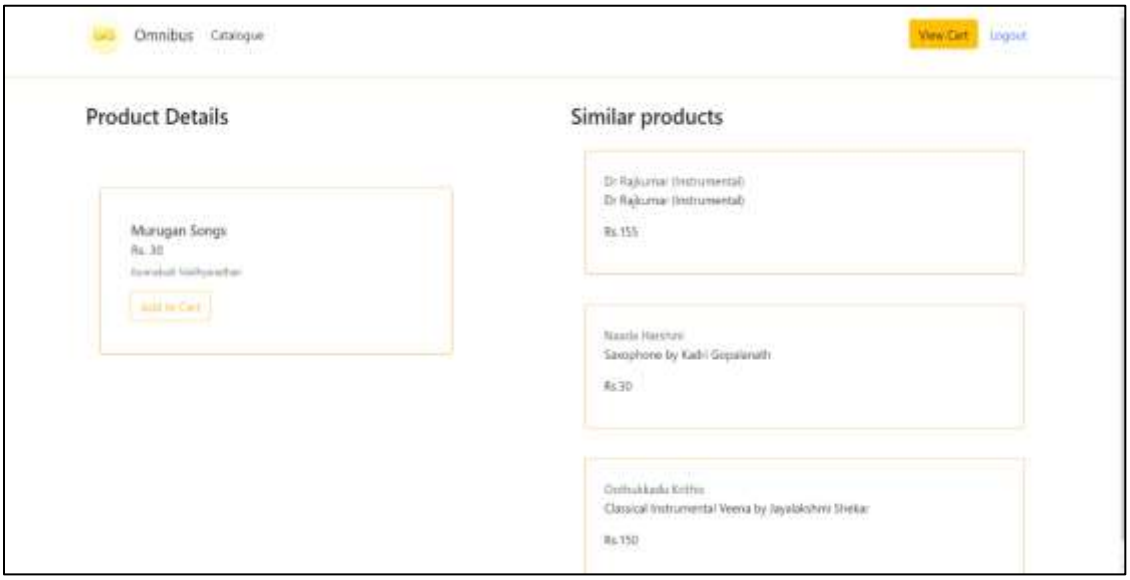


Fig 4. Snapshot of recommendations fetched from the backend

Snapshot 4: Module 4 – Store metrics

The user metrics are collected and stored as a part of the metadata for the model. The below snapshot shows the metrics collected such as impressions (how many times did the users see a product from this model), click-

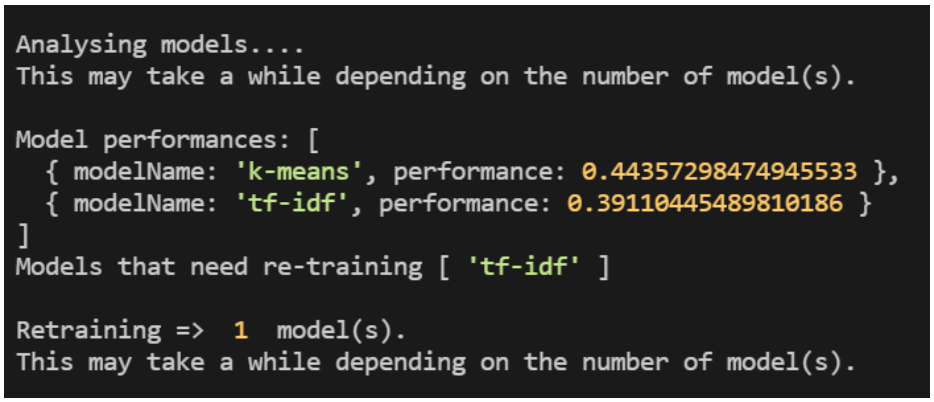
through(how many times a user clicked on a recommendation) and order conversion (how many times the user ordered a recommended product).

```
{
  "modelName": "k-means",
  "training": {
    "separator": " , ",
    "database": { "whereClause": {} },
    "storage": {
      "local": "/home/vivek/Desktop/Projects/FYP/build/ml-node/utils/tf-idf-data.csv"
    }
  },
  "modelId": "e7b9cf5f-da1e-4b1b-8c88-c6077b7499fb",
  "modelVersion": "0.1",
  "metrics": {
    "impressionCount": 54,
    "clickThroughCount": 17,
    "orderConversionCount": 9
  },
  "performanceScore": 0.39110445489810186
}
```


Fig 5. Snapshot of metrics stored in the metadata

Snapshot 5: Module 5 – Analysis of metrics

The metrics stored in the metadata are used to calculate each model's performance score as shown below in Fig 7.5. The performance score is calculated using the click through rate and the order conversion rate. The model with the lowest performance score will be sent for retraining as shown.

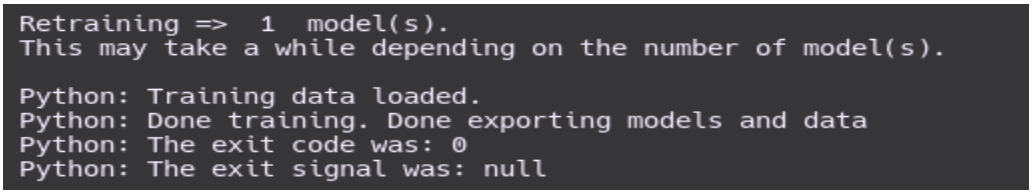


```
Analysing models....
This may take a while depending on the number of model(s).

Model performances: [
  { modelName: 'k-means', performance: 0.44357298474945533 },
  { modelName: 'tf-idf', performance: 0.39110445489810186 }
]
Models that need re-training [ 'tf-idf' ]

Retraining =>  1  model(s).
This may take a while depending on the number of model(s).
```


Fig 6. Snapshot of analysis of metrics

Snapshot 6: Module 6 – Model tuning

The models that need to be retrained are trained again with the training dataset and parameters. The below snapshot in Fig 7.6 shows the various stages of retraining. The retraining script calls a python script and loads the training dataset based on the file storage location present in the metadata. Once the training is completed, the pipeline exports the model and the data, creating a new model and metadata for the model.



```
Retraining =>  1  model(s).
This may take a while depending on the number of model(s).

Python: Training data loaded.
Python: Done training. Done exporting models and data
Python: The exit code was: 0
Python: The exit signal was: null
```


Fig 7. Snapshot of retraining models

Snapshot 7: Module 7 – Machine Learning models

The trained machine learning models are used to precompute recommendations. The python scripts for precomputing the recommendations are called. The script loads the model and the required data along with a list of product IDs. It generates the recommendations for each product ID and stores them in .json file. The node.js script reads the .json file and populates the database with the recommendations computed by the model.

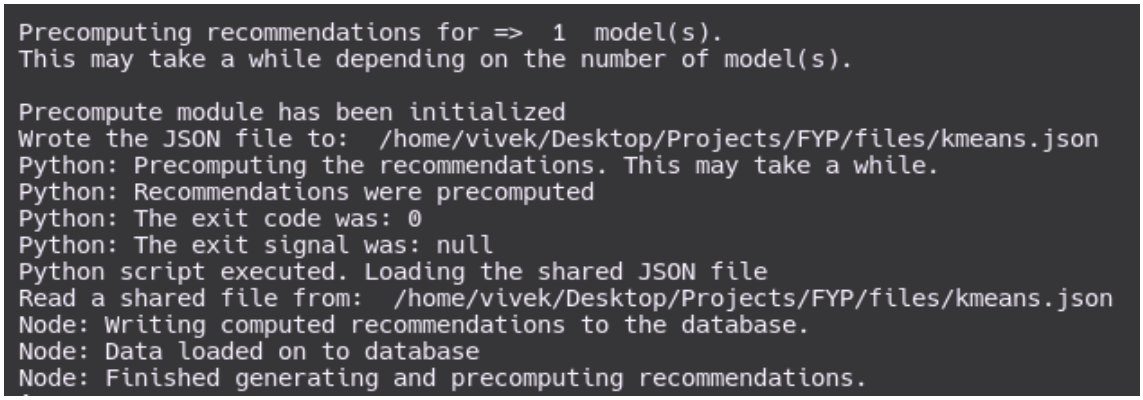


Fig 8. Snapshot of precomputing recommendations

Conclusion:

In conclusion, our system introduces automation and streamlining to the whole flow of machine learning workflow that is needed for product recommendations. This orchestration of this whole process makes the system more self-sustainable and better at scaling with minimal human interaction. It also gives the system the capability to improve its recommendations over time. The existing systems mainly focus on making good product recommendations while neglecting the importance of the ML pipeline. This leads to them having to deal with the problems that arise from a monolithic piece of code that handles the entire workflow. Such monolithic systems cannot be scaled easily and are a nightmare to maintain. Our work "Orchestration of ML-Based Recommendation Systems" introduces the automation of the workflow for product recommendations and evaluation of the metrics.

Major contributions:

We have provided a streamlined, structured and automated ML workflow that orchestrates many recommender systems into a single pipeline. This pipeline also allows the models to improve their recommendations over time without a developer having to run separate accuracy tests on the models. On a model specific level, we have used product descriptions to provide content-based recommendations.

Future work:

In the future, more advanced machine learning models can be used to deal with products or product specifications in regional languages. This would help in providing more accurate recommendations to the end user. The retraining of models and pre computation of recommendations can be offloaded to a compute instance on the different machine or server.

Live data streams can be used for training the models instead of static data sets. When new products come in, the models will not have to be trained all over again for the whole dataset but for only a small subset of new products.

User data and reviews can be used to make the recommendations more personalized. Data visualization about the performance of the models can be used to understand the business's consumer demographics better to improve the companies order conversion.

References

[1] Sivapalan, Sanjeevan, Alireza Sadeghian, Hossein Rahnama, and Asad M. Madni. "Recommender systems in e-commerce." In 2014 World Automation Congress (WAC), pp. 179-184. IEEE, 2014.
[2] Son, Jieun, and Seoung Bum Kim. "Content-based filtering for recommendation systems using multi attribute networks." Expert Systems with Applications 89 (2017): 404-412.
[3] Cai, Yi, Ho-fung Leung, Qing Li, Huaqing Min, Jie Tang, and Juanzi Li. "Typicality-based collaborative filtering recommendation." IEEE Transactions on Knowledge and Data Engineering 26, no. 3 (2013): 766-779.
[4] Schröder, Gunnar, Maik Thiele, and Wolfgang Lehner. "Setting goals and choosing metrics for recommender system evaluations." In UCERSTI2 workshop at the 5th ACM conference on recommender systems, Chicago, USA, vol. 23, p. 53. 2011.
[5] Souali, Kamal, Abdellatif El Afia, and Rdouan Faizi. "An automatic ethical-based recommender system for e-commerce." In 2011 International Conference on Multimedia Computing and Systems, pp. 1-4. IEEE, 2011.