

LEV4REC: A Low-Code Environment to Support the Development of Recommender Systems

Claudio Di Sipio (✉ claudio.disipio@graduate.univaq.it)

University of L'Aquila

Juri Rocco

University of L'Aquila

Davide Ruscio

University of L'Aquila

Phuong T. Nguyen

University of L'Aquila

Research Article

Keywords: Recommender systems, Low-code platforms, Model-Driven Engineering

Posted Date: April 11th, 2022

DOI: <https://doi.org/10.21203/rs.3.rs-1537563/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

LEV4REC: A Low-Code Environment to Support the Development of Recommender Systems

Claudio Di Sipio · Juri Di Rocco ·
Davide Di Ruscio · Phuong T. Nguyen

the date of receipt and acceptance should be inserted later

Abstract Recommender systems (RSs) are complex software systems that suggest relevant items of interest given a **specific application domain** to users. The development of RSs encompasses the execution of different steps, including **data preprocessing**, **choice of appropriate algorithms**, **item delivery**, to name a few. Though RSs can alleviate the curse of information overload, existing approaches resemble black-box systems, where the end-user is not supposed to customize the overall process. To fill the gap, we proposed LEV4REC, an initial MDE-based prototype for supporting the mentioned activities needed to conceive an RS from the design phase to the actual deployment of the system, including the parameters fine-tuning.

As a first step, the **user defines a coarse-grain model** that allows the configuration of the desired RS, which then can be finalized by fine-tuning different parameters. LEV4REC eventually generates the source code of the RS, being ready for actual deployment. LEV4REC is provided as a plugin extension for two different IDEs and an initial web interface. To study the capabilities of the approach, we utilized LEV4REC in curating two existing RSs, which have been designed on top of two different algorithms, i.e., **collaborative filtering** and **feed-forward neural network**.

Experimental results show that our proposed tool can create systems that can provide suitable recommendations, thereby conforming to their original design. LEV4REC is capable of enabling developers to refine the produced system by experimenting with different algorithms, experimental settings, and evaluation metrics.

Keywords Recommender systems, Low-code platforms, Model-Driven Engineering

Claudio Di Sipio, Juri Di Rocco, Davide Di Ruscio, Phuong T. Nguyen
DISIM - University of L'Aquila (Italy)
E-mail: firstname.lastname@univaq.it

1 Introduction

To cope with their everyday programming tasks, developers access and browse various sources, searching for items relevant to their need [11]. For instance, developers can look for API calls/code snippets from open source software (OSS) forges, e.g., GitHub, Maven, or discussions from Stack Overflow to complete the development task at hand. Given a wide range of choices, the problem is not a lack, but instead, an overload of information [40]. Generally, developers are overwhelmed by many techniques and algorithms that might be too complex to comprehend fully. In fact, the development of new software projects relies heavily on the reuse of existing components, artifacts, and frameworks that offer invaluable support to newcomer developers. In this context, recommender systems (RSs) [45,24] come in handy as they help users narrow down the search scope and approach suitable items [47]. So far, several RSs have been conceptualized to extract knowledge from existing repositories to provide tailored recommendations to developers, such as code snippets [42], tags/topics [17], documentation [43], allowing them to complete their current development tasks. Nonetheless, the development of custom RSs is a challenging task that necessitates deep knowledge of different technologies and tools, e.g., the recommendation algorithms to be adopted, the evaluation protocols to be employed, as well as the metrics to be considered to assess the final outcomes.

Over the last decade, several frameworks have been proposed by academia and industry [21,23], attempting to mitigate the abundance of possible choices and reduce the burden related to the development and evaluation of RSs. However, even though such frameworks represent relevant facilitators for reproducing performed experiments, they still require development activities with the programming languages the selected frameworks are written. Thus, despite the availability of an impressive number of recommendations algorithms and evaluation techniques, their wide adoption is still an issue for users who do not have enough expertise to develop their own RSs.

Recently, the Elliot framework [6] has been proposed to support the configuration of RSs, allowing developers to specify their design requirements using the YAML file format. Elliot aims at dealing with the increasing number of recommendation algorithms, evaluation protocols, metrics, and tasks, by offering an environment to create evaluation benchmarks or tune hyperparameters. While the tool works in practice, according to our investigation, there is still the need to assist developers in tailoring an RS's design. In this respect, we aim to support developers by providing them with a low-code environment to specify the characteristics of the desired RS, and guiding the users throughout the whole development process up to the automated generation and deployment of the final source code. To this end, we proposed an initial prototype, named LEV4REC, to assist developers in designing, configuring, and delivering recommender systems by taking inspiration from the low-code paradigm [15]. Given an initial configuration specified by the user, the system can realize the designed components by generating the corresponding source code

implementation. LEV4REC provides users with an environment *(i)* to select the **components** that need to be used for the wanted recommendation system; *(ii)* to **configure** the chosen modules through a dedicated modeling environment. The two specifications conform to a dedicated metamodel defined using the Eclipse Modeling Framework (EMF) [25]. In addition, a generator module built on top of Acceleo¹ can generate Python source code implementing the specified RS automatically. In such a way, LEV4REC allows RS developers to define, fine-tune, and test their recommender systems. To validate our approach, we deployed LEV4REC to build two real-world RSs. From an empirical evaluation of various datasets, we see that our conceived environment can adequately define and implement the critical components of the considered systems, allowing them to follow their original design and implementation.

With respect to our previous work [15], we enhance LEV4REC by enabling the generation of Web-based RSs as well their usage into two different IDEs, i.e., VS code² and Eclipse.³ We evaluate the tool by introducing another RS that relies on a completely different technique, i.e., neural network model. This aims to demonstrate the tool’s capability of generating different systems using the elicited concepts.

The **main contributions** of our work are given below:

- A low-code environment, called LEV4REC, to support the creation, configuration, and deployment of RSs;
- An empirical evaluation on real-world datasets to showcase the technical advantages of our proposed approach by resembling the key components of two existing RSs;
- Two different IDE integrations as well as a Web service implementation;
- A complete replication package and the generated results have been made available to foster future research.⁴

The paper is organized as follows. Section 2 introduces the motivations and background for our work. The LEV4REC architecture and its constituent components are detailed in Section 3. We present two different case studies in Section 4. An empirical evaluation of LEV4REC is presented in Section 5 while the threats to the validity of our findings are discussed in Section 7. We review related work in Section 8, and conclude the paper in Section 9.

2 Motivations and Background

This section identifies the challenges and the methodologies that have been used to conceive the proposed approach. In particular, we elicit the RS essential building blocks and cope with the related issues in Section 2.1. Afterward,

¹ <https://www.eclipse.org/acceleo/>

² <https://code.visualstudio.com/>

³ <https://www.eclipse.org/>

⁴ <https://github.com/MDEGroup/LEV4REC-Tool>

Section 2.2 presents an overview of techniques and frameworks adopted to support the development of RSs.

2.1 Challenges in developing RSs

To build a recommender system from scratch, developers need to follow a number of steps [49,14] as shown in Fig. 1. First, based on their context, developers select suitable *Raw data sources*, e.g., source code of OSS projects required to recommend useful code snippets. Then, the *Filtering data* phase is conducted to clean data and retain the most useful/relevant items, exploiting various preprocessing techniques, e.g., parsing, and keyword searching. In the *Algorithm selection* phase, recommendation techniques suitable for solving the desired tasks are chosen to work on the filtered data. By the *Outcomes evaluation* phase, developers evaluate the resulting system using quality metrics [47]. Depending on the outcomes, some *System tuning* steps may be necessary for fine-tuning the framework and improving its performance.

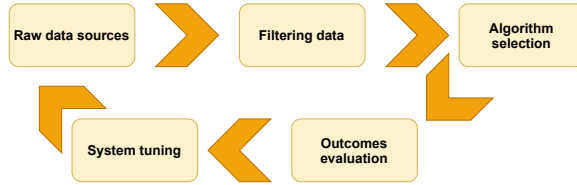


Fig. 1 Key operations in the development of an RS.

Properly conducting the steps mentioned above to conceive and evaluate a recommender system from scratch is a daunting task [6], especially for developers that are not RS experts. In most cases, they may adopt off-the-shelf tools to realize the required recommendations [32]. While this eases the development process, it may prevent them from personalizing and optimizing the design. Therefore, we identify the following main development challenges when it comes to the conceptualization and evaluation of an RS [44]:

- *DC1: Design and specification of the recommendation process:* The first step is to identify the commonalities among different RSs. Satisfying these requirements at the beginning of the process is a contributing factor to realize the envisioned system.
- *DC2: Collecting data for training RS:* The selection of features and preprocessing techniques are among the contributing factors to the success of an RS [1]. In particular, data curation is crucial for the performance, e.g., the size of training data and their properties such as rating frequency distribution, rating space, to name a few.

- *DC3: Managing the composition of miscellaneous components:* Producing recommendations involves several building blocks, which need to be executed following a well-defined process. Furthermore, each module requires a specific input format to produce intermediary outcomes which are used by the subsequent components.
- *DC4: Development of source code components:* Once the system has been designed, the next step is to implement the conceived components by writing the corresponding source code. Such an activity is challenging since it necessitates rigorous implementations to realize the desired functionalities and satisfy the system requirements.
- *DC5: Outcomes evaluation and system tuning:* The evaluation of an RS involves performing different steps, among others, the selection of testing data and methodology, as well as identifying suitable quality metrics. In fact, conducting these steps is time consuming and requires domain-specific knowledge [14].

In this respect, we see an urgent need to conceptualize a general-purpose environment to foster the configuration of an RS from the design to the deployment phase. LEV4REC has been conceived as a means to address the challenges mentioned above by employing software engineering and model-driven engineering principles and tools. In particular, to tackle *DC1* and *DC3*, we rely on different generated specifications, i.e., **system and feature models**, respectively. Concerning the *DC2* challenge, we define a system specification that allows us to define a dataset’s structure and select suitable preprocessing techniques. It is worth noting that we do not cover the creation of the dataset since this can be done with existing domain-specific languages or tools, e.g., CROSSFLOW [33] and Perceval [19]. To deal with *DC4*, we specify a code generator that automatically produces the Python code needed for each component of the designed system. Though we limit ourselves to this programming language, the developed generator can support other technologies such as Java and C++. Developers can generate a modified version of the resulting RS by editing the two mentioned abstractions, i.e., the feature model and the system configuration. In such a way, our environment can cover the specification and the deployment of RSs once evaluated as described in *DC5*.

2.2 Supporting technologies

Feature-Oriented Software Development (FOSD) [53] is a design paradigm for implementing software systems based on a predefined set of features, e.g., characteristics or requirements. In particular, FOSD aims to modularize a software system by identifying and combining feature modules. *Feature model* [54] is a core concept in FOSD consisting of a set of organized features and constraints between them, e.g., if a specific feature is selected while some others cannot be. Such a model is defined by a *feature tree* that is a hierarchical representation of features. These features are composed in different ways, e.g., they may be mandatory, optional, or mutually exclusive depending on the ones

already selected. Several valid *feature configurations* can be chosen to satisfy the constraints defined at the level of the feature model, depending on the input features.

Possible applications of feature models to configure recommender systems and machine learning applications have been analyzed in [22]. The study evaluates the feasibility of feature models under three dimensions, i.e., interactive configuration, reconfiguration, and modeling processes. Besides the analyzed applications, this study highlights other scenarios, e.g., modeling user interactions, intelligent grouping features, and constraints, to name a few. In recent work, a dedicated feature model has been used to elicit essential components for recommender systems in MDE [3]. To understand which modeling tasks are susceptible to recommendations, the authors conduct a systematic mapping study by analyzing 66 papers to foster future research in this domain.

Concerning the development of recommender systems, several frameworks and libraries have been released in recent years. The RankSys tool [56] supports the rapid prototyping and testing of RSs based on the collaborative-filtering technique. Similarly, the Surprise library [29] offers a collection of algorithms specifically designed for rating prediction. Furthermore, it also provides utilities to evaluate the performance of each implemented algorithm. Besides traditional techniques, Machine Learning (ML) models have been widely used to build recommender systems [51]. Among others, ML frameworks such as Scikit-learn,⁵ PyTorch,⁶ TensorFlow,⁷ or Keras⁸ provide a rich set of functionalities, which enables developers to flexibly customize their implementation by fine-tuning hyperparameters.

Low-Code Development Platforms (LCDP) have been recently introduced to simplify the development of fully functional software systems employing advanced graphical user interfaces and visual abstractions requiring minimal or no procedural code [46]. The main goal of LCDPs is to permit users to build their software systems even if they do not have advanced programming skills [59, 48].

3 Proposed approach

This section presents LEV4REC, an extensible environment for supporting the development of recommender systems. **The proposed approach relies on the assumption that it is possible to identify typical components building up RSs and recurrent stages underpinning their evaluation** [6]. In this respect, by adhering to the principles and methods supporting the development of recommender systems [10, 16], we identified a set of features characterizing any RS, and proposed the tool-supported process shown in Fig. 2.

⁵ <https://scikit-learn.org/>

⁶ <https://pytorch.org/>

⁷ <https://www.tensorflow.org/>

⁸ <https://keras.io/>

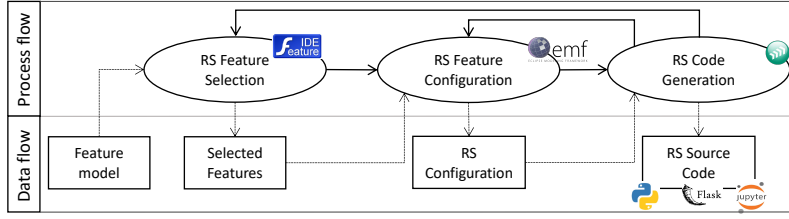


Fig. 2 Overview of the proposed approach.

According to the proposed methodology, an **RS developer (RSD)** starts with the *RS Feature Selection* phase to **select the features of the RS under development**, e.g., the recommendation **algorithm** to be employed, the **pre-processing phases** to be operated on the selected dataset, and the **evaluation** procedure to be performed. A dedicated set of constraints supports such a selection. For instance, if the designer decides to use a supervised dataset, the environment automatically disables the possibility of selecting algorithms for unsupervised learning. After the selection of the wanted features, the environment supports the subsequent *RS Configuration* phase. For instance, if in *RS Feature Selection* the user has selected cross-validation as an evaluation technique to be employed, during the *RS Feature Configuration* phase, the wanted number of folds needs to be specified. The final step of the process consists of the *RS Code Generation* activity. It takes as input the complete *RS Configuration* and generates the source code of the specified RS. Python, Flask,⁹ and Jupyter¹⁰ are the target technologies that are currently supported. The proposed approach facilitates the possibility of testing different features and configurations as shown in Fig. 2 by the arrows from *RS Code Generation* back to *RS Feature Selection* and *RS Feature Configuration*. We detail each phase of the proposed process in the following subsections.

3.1 RS Feature Selection

By carefully review existing technologies, we defined an agnostic feature model to specify different types of recommender systems according to the developer's needs. As shown in Fig. 3, the specified model has two main elements defined as follows.¹¹

- **Recommender component:** This feature includes all the necessary building blocks to construct an RS, e.g., preprocessing techniques, datasets, algorithms, and evaluation utilities, to name a few. Different subfeatures for

⁹ <https://flask.palletsprojects.com/>

¹⁰ <https://jupyter-notebook.readthedocs.io/en/stable/notebook.html>

¹¹ For the sake of clarity, only some of the features are shown. Folded features are annotated in Fig. 3 with an integer representing the number of the sub-features that are hidden. The interested reader can refer to the complete feature model, which is available at <https://tinyurl.com/5yxawyfe>.

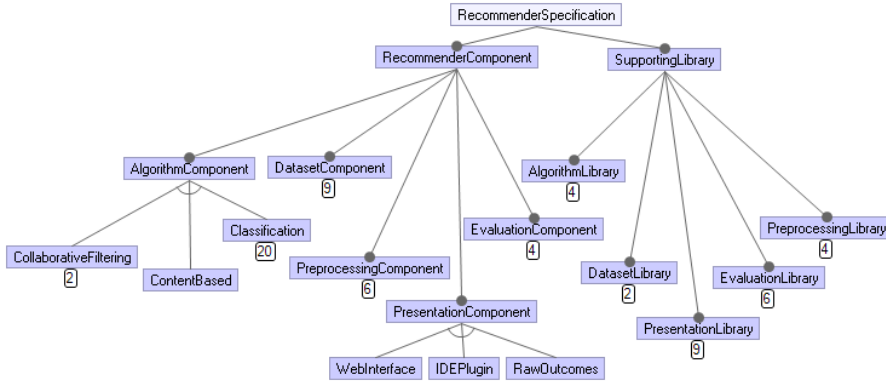


Fig. 3 Overview of the proposed **feature model**.

each principal component have been defined to cover the desired RS functionalities. For instance, depending on the initial requirements considered during the recommendation process, it is possible to select a collaborative-filtering algorithm on top of either the user-based or item-based technique [31].

- **Supporting library:** Alongside the system design, the developer needs to specify the software libraries that can be adopted to support the specified recommender components. For instance, if the user selects the *Content-Based* algorithm component, the libraries that do not support this algorithm are disabled by leaving only *Surprise* as a possible selection. The feature sub-trees also deal with potential incompatibilities among different libraries, for instance, when the developer cannot apply a data splitting technique provided by the Scikit-learn library on a Surprise algorithm.

As previously mentioned, the whole feature selection phase is restricted through a set of defined constraints that can either include or exclude certain features depending on the current selection. Table 1 reports and explains the set of the constraints embedded in the LEV4REC feature model. For instance, a *Supervised algorithm* cannot be selected if the approach relies on a *Unsupervised Dataset* to produce recommendations. Similarly, we impose constraints that act on the software libraries employed to generate the actual RS code. In particular, the *Surprise* feature dictates that the *Surprise preprocessing* and *Surprise Split* features must be adopted to work appropriately by excluding other libraries, e.g., Scikit-learn.

3.2 RS Feature Configuration

Once the feature model has been defined, the next step is the generation of an initial RS configuration conforming to the metamodel shown in Fig. 4. The metamodel defines all the constructs that can be used to define a complete RS

Table 1 Constraint-supported RS specification.

Constraint	Description
$\neg \text{Scikit-learn} \implies \neg \text{Seaborn}$	The <i>Scikit-learn</i> library is required to run the <i>Seaborn</i> graphical utilities
$\neg \text{TextualData} \implies \neg \text{NLP}$	As the <i>NLP</i> pipeline acts on textual data, it requires such a format to be enabled
$\text{CollaborativeFiltering} \implies \neg \text{Scikit-learn} \wedge \neg \text{PyTorch} \wedge \neg \text{TensorFlow}$	Selecting <i>Collaborative filtering</i> technique automatically excludes software libraries that do not support it, e.g., <i>Scikit-learn</i> , <i>TensorFlow</i> , or <i>PyTorch</i>
$\text{IDEPlugin} \implies \text{IDELibrary}$	If the <i>IDEPlugin</i> feature is selected then <i>IDELibrary</i> must be set as the presentation layer generator
$\text{RandomSplitting} \implies \text{SKRandomSplit} \vee \text{SurpriseRandomSplit}$	If <i>CrossFold</i> uses a random splitting rule, RSD can select one of the supported libraries, i.e., <i>SKRandomSplit</i> or <i>SurpriseRandomSplit</i>
$\text{Sklearn} \implies \text{SklearnPreprocessing} \wedge \text{SklearnSplit}$	To preserve consistency, the system automatically selects <i>SklearnPreprocessing</i> and <i>SklearnSplit</i> for the corresponding library
$\text{SplittingKfold} \implies \text{SurpriseCrossFold} \vee \text{SKCrossFold}$	Similar to the <i>RandomSplitting</i> feature, RSD can select two different <i>CrossFold</i> implementations depending on the chosen library
$\text{SupervisedDataset} \implies \neg \text{UnsupervisedAlgorithm}$	The system selects automatically an <i>UnsupervisedAlgorithm</i> if RSD specifies a <i>SupervisedDataset</i>
$\text{Surprise} \implies \text{SurpriseSplit} \wedge \text{SurprisePreprocessing}$	If <i>Surprise</i> is selected as <i>AlgorithmLibrary</i> , then the preprocessing and splitting policies must belong to this library
$\text{UnsupervisedDataset} \implies \neg \text{SupervisedAlgorithm}$	If <i>Unsupervised Dataset</i> is selected then the developer cannot select supervised algorithms
$\text{WebInterface} \implies \text{WebLibrary}$	<i>WebLibrary</i> must be used as the presentation layer if the feature <i>WebInterface</i> is set

configuration, which is then used to generate the source code of the designed RS. The root element named **RModel** is composed of several abstract components depicted in light grey, i.e., **Dataset**, **Recommender system**, **Validation Technique**, and **PresentationLayer**. Each entity can be specialized with concrete elements needed to implement different functionalities. The metaclasses extending such abstract elements are described in the following.

At the beginning of the design process, the user must specify the type of data that will be used to feed the recommender system. Figure 5 depicts **Dataset** specializations which come in handy by providing the information needed to represent a dataset, i.e., independent **variables**, **preprocessing** techniques, e.g., Natural Language Preprocessing, normalization, etc., and the corresponding **datasetManipulationLibrary** that is used to manipulate the datasets at hand, e.g., *pandas* [37] and *numpy* [55]. Since recommender systems rely on heterogenous data, LEV4REC provides **Data Source** concepts designed to collect and organize information coming from different sources, i.e., **Code repository**, **Communication Channel**, and **Bug Tracking**. Furthermore, the **DataStructure** metaclass is used to dictate the dataset features, e.g., the format, the size, to name a few. For instance, data extracted from a given GitHub repository could be organized in a **Graph** as well as a **Matrix** depending on the nature of the recommender system.

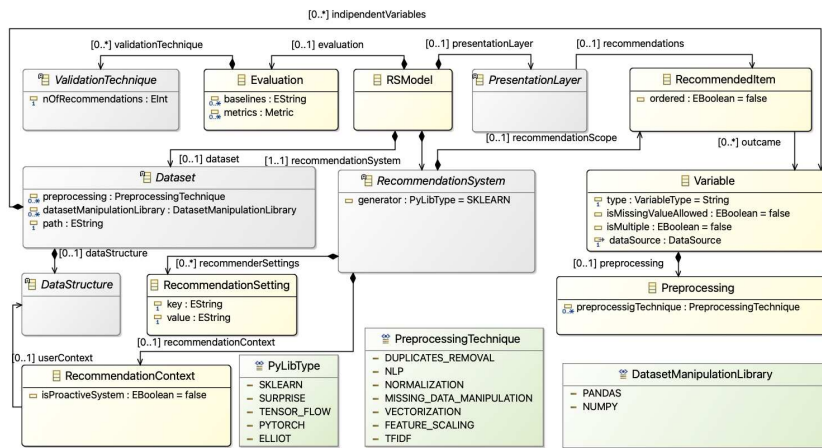


Fig. 4 The LEV4REC configuration metamodel.

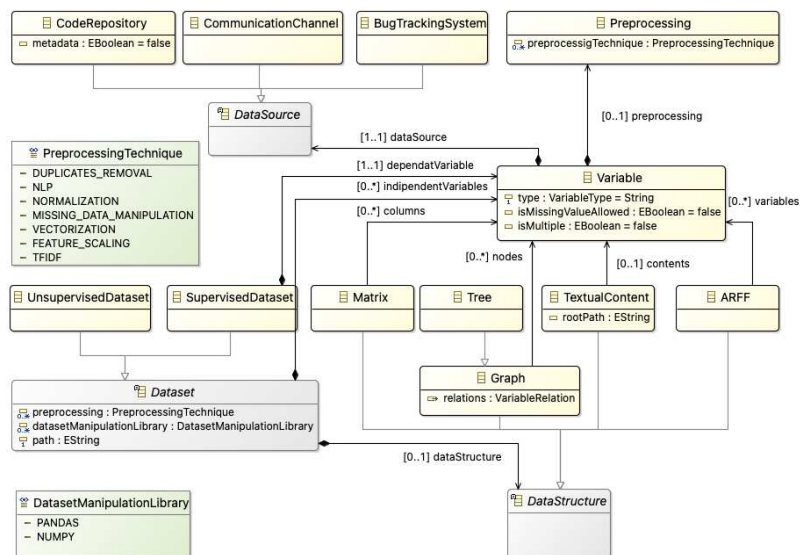


Fig. 5 The LEV4REC Dataset and DataStructure metaclasses.

Once the dataset has been specified, the user can configure the underpinning recommendation algorithm and its internal parameters by using the `RecommendationSystem` concepts shown in Fig. 6. Depending on the `RecommendationContext`, several algorithms can be chosen, e.g., `MachineLearningBasedRS`, `FilteringRS`, to name a few. The user can also implement a `CustomRecommender` if none of the implemented systems can offer the desired functionalities. Afterward, the elicited algorithm can be decorated with different

parameters using **RecommendationSetting** and several enumerations entities represented by the green boxes. For instance, if a filtering algorithm is selected, it is possible to specify the similarity function and the type of the algorithm among the available ones.

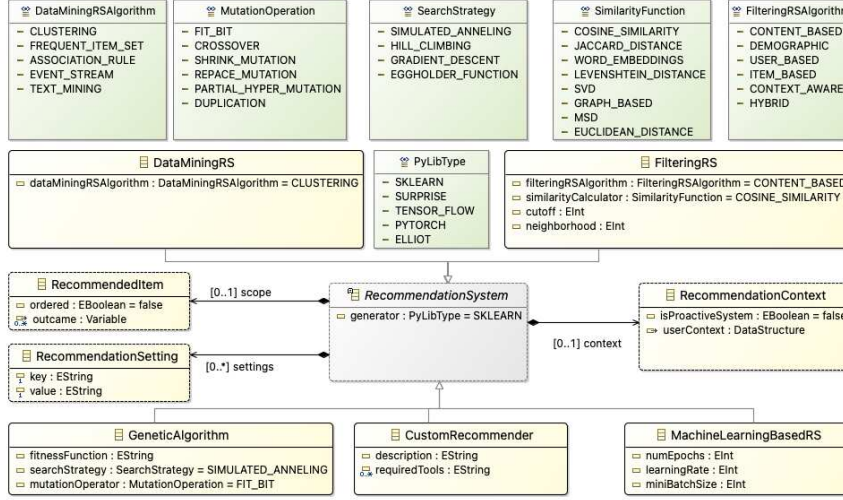


Fig. 6 The LEV4REC RecommenderSystem metaclass.

The designed algorithm eventually produces a list of **RecommendedItem** that can be used by the next components, i.e., **ValidationTechnique** and **PresenationLayer**, to evaluate the system and deliver the outcomes to the user, respectively. The former specifies two main types of **ValidationTechnique** that could be used to assess the selected recommender system, i.e., **Automated-Validation** and **UserStudy**. Figure 7 depicts the key concepts related to the **ValidationTechnique** metaclass. Similar to the algorithm’s configuration, the user can set different parameters such as metrics, the type of analysis to be conducted, and the programming library to generate the corresponding code. At its current status, LEV4REC supports the generation of automatic techniques, while user studies are not yet implemented. Nevertheless, we can rely on existing works that come in handy to specify such kind of evaluation [50, 9].

Figure 8 depicts the **PresentationLayer** metaclass that is devoted to retrieving produced recommendations from the used IDE e.g., VSCode and Eclipse. **PresentationLayer** relies on a **WebService** that allows two main user interactions, i.e., proactive and reactive. The former continuously monitors the user’s context and performs some actions according to certain conditions. Contrariwise, the latter reacts to a specific **UserEvent** that directly triggers the action. Both interactions are managed by **RecommendationHandler** that also rules **RecommendationUsage**, i.e., how the user makes use of the returned

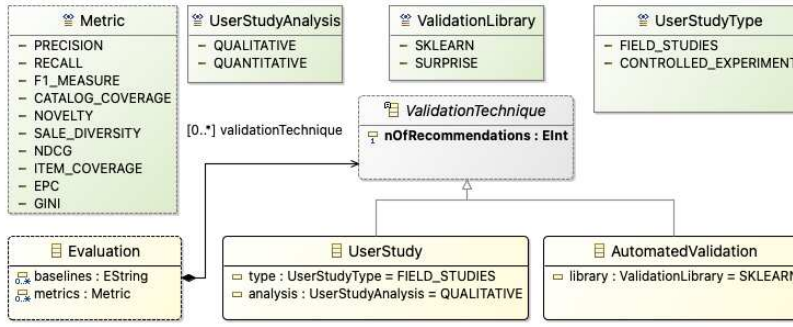


Fig. 7 The LEV4REC ValidationTechnique metaclass.

items. In this respect, transformative usage acts directly on the retrieved recommendations by modifying them. For instance, if the system suggests code snippets, they can be adapted according to the actual context. Meanwhile, the user can visualize the delivered items without the possibility of modifying them when the `RecommendationUsageType` instance is set to visualization. According to the type of the selected interaction, the system provides `GUIElement` components capable of handling different behaviors in a specific graphical interface.

3.3 RS Code Generation

The final step of the process shown in Fig. 2 is the generation of the actual source code for the selected and configured RS components. Starting from an input *RS Configuration*, LEV4REC produces the corresponding source code implementing the designed RS by relying on template engines that have been successfully employed to support this task [7].

The code generation phase has been developed by using Acceleo.¹² It is a well-known technology in model-driven engineering, and it supports the development of code generators in terms of dedicated templates.¹³ Acceleo templates identify repetitive and static parts of the system being generated and embody specific queries on the source models to fill the dynamic elements. The rationale behind the selection of Acceleo is twofold: (i) it provides a well-defined syntax to specify the templates, and (ii) such a generation technique can deliver output in different programming languages apart from Python.

¹² <https://www.eclipse.org/acceleo/>

¹³ https://wiki.eclipse.org/Acceleo/User_Guide#Templates

Listing 1 An explanatory Acceleo template.

```

[template public generateFilteringRS(algo : FilteringRS)]
[if (algo.filteringRSAlgorithm = FilteringRSAlgorithm::USER_BASED)]
is_user_based=True
[/if]
[if (algo.filteringRSAlgorithm= FilteringRSAlgorithm::ITEM_BASED)]
is_user_based=False
[/if]
neighborhood=[algo.neighborhood/]
[if (algo.similarityCalculator = SimilarityFunction::
COSINE_SIMILARITY)]
sim_funct='cosine'
[/if]
[if (algo.similarityCalculator = SimilarityFunction::MSD)]
sim_funct='msd'
[/if]
sim_settings = { 'name': sim_funct, 'user_based': is_user_based }
algo = KNNWithMeans(k=neighborhood, sim_options=sim_settings)
[/template]

```

To support the generation of a recommender system, we devise a hierarchical structure in which we provide different templates for each metaclass described in Section 3.2. In such a way, the system can generate a custom instance of each component that can be even fine-tuned to experiment with different configurations. As an example, Listing 1 depicts the Acceleo template to generate the source code to adopt a recommendation algorithm based on the collaborative filtering technique. The template uses the key element specified in the model to generate source code. In particular, this component automates the setting of parameters and the evaluation of results. In particular, the *algo* entity represents an instance of the **FilteringRS** metaclass in Fig.6. According to the features selected by the RS developer, this component automates the setting of the parameters to produce the corresponding source code. For instance, the *is_user_based* variable can be *True* or *False* depending on the chosen filtering technique, i.e., either user-based or item-based. Similarly, the RS developer can choose a different similarity function by setting the corresponding attribute in the LEV4REC model.

3.4 IDE Integration

As depicted in Fig. 4, the **PresentationLayer** concept is specialized in different sub-concepts that define the way where the recommendations are provided to the users. For example, by using **WebServices** instances, developers dictate that the recommender system should be deployed as a Web service providing recommendations by means of a REST API.¹⁴ In our implementation, we generate Web services in Flask, a popular Python framework used for developing Web applications.

¹⁴ <https://datatracker.ietf.org/doc/html/rfc7231>

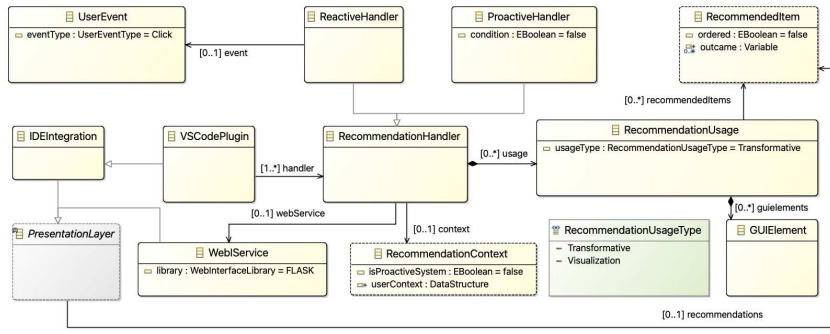



Fig. 8 The LEV4REC `PresentationLayer` metaclass.

An example of how generated services can be used to integrate the recommender system into different IDEs is shown in Fig. 9. In particular, the left part shows the Swagger UI interface generated from the OpenAPI specification¹⁵ of the developed and deployed RS, while the right part of Fig. 9 depicts two IDE integrations, i.e., Eclipse and VS Code, that query the services to provide recommendations to the users.

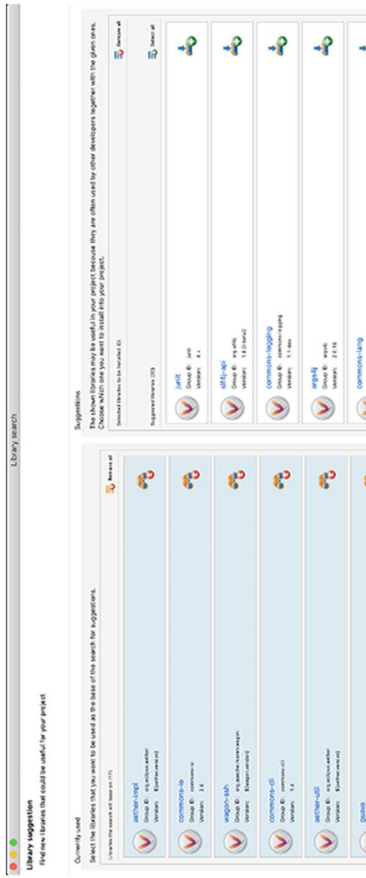
¹⁵ <https://swagger.io/specification/>



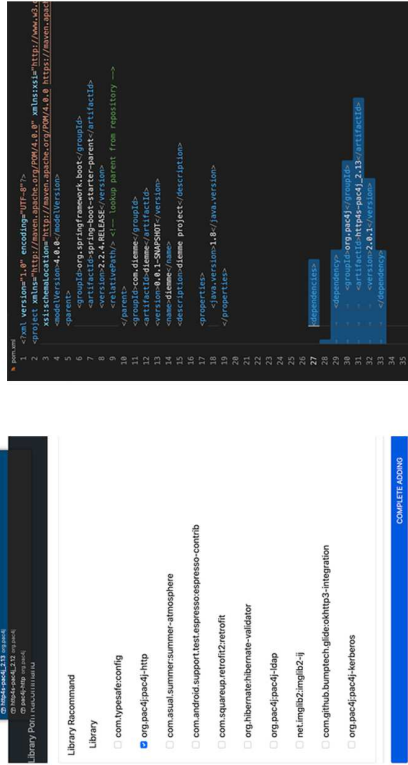
Swagger UI generated from OpenAPI specification

Fig. 9 Using LEV4REC RSs from IDEs.

Eclipse integration



VS Code integration



4 Case studies

By following the process presented in the previous section, we make use of LEV4REC to design, tune, and deploy two existing recommender systems, i.e., a k-nearest neighbor-based algorithm (named KNN hereafter) [52] and AURORA [41]. KNN aims to address the scalability problem in personalized recommendations, it combines the usage of an adaptive version of the KNN (AKNN) and ontologies to recommend relevant items for any user. In the scope of our paper, we elicit this approach since it provides movie recommendations, which is a well-known application domain [18]. In the context of this paper, we implement KNN by relying on the Surprise library.¹⁶ Meanwhile, AURORA classifies metamodels by using a feed-forward neural network model trained with a curated labeled dataset. To feed the underpinning model, feature vectors are extracted using three encodings, i.e., uni-grams, bi-grams, and n-grams. Like the KNN methods, we resemble the AURORA key functionalities by relying on an external Python library, namely Scikit-learn,¹⁷ which provides all the needed building blocks.

We discuss the development of KNN and AURORA in the following subsections. As we mentioned in Section 2.1, at the current stage of development the proposed platform does not support the generation of a dataset. Thus, in the scope of this evaluation, we focus on the specification, configuration, and generation of the two RSs.

4.1 RS Feature Selections

The first step involves the feature selections of the two systems by employing the devised feature model. Figure 10 shows a fragment of the KNN and AURORA specifications, which have been done utilizing LEV4REC. To resemble the KNN peculiarities, a *Supervised dataset* and *user-based collaborative filtering* algorithm are needed. To this end, the fragment of the selected features depicted in Fig. 10(a) shows that the *UserBased* feature has been selected accordingly.

It is worth noting that defining a partial set of features is sufficient to resemble the proposed environment, which relies on well-defined constraints to obtain a valid configuration. For instance, as shown in the lower-side of Fig. 10(a), the user manually selected eight features, and the environment automatically disabled 47 features and selected the remaining 24 ones.

Similarly, feature selections for AURORA can be easily specified as in Fig. 11(a). Since the tool exploits a *supervised classifier*, the RS developer is forced to use once again a *supervised dataset*. It is worth noting that such a constraint is automatically set using the feature model. Thus, the RS developer can elicit the wanted component without taking care of possible violations. Concerning

¹⁶ <http://surpriselib.com/>

¹⁷ <https://scikit-learn.org/>

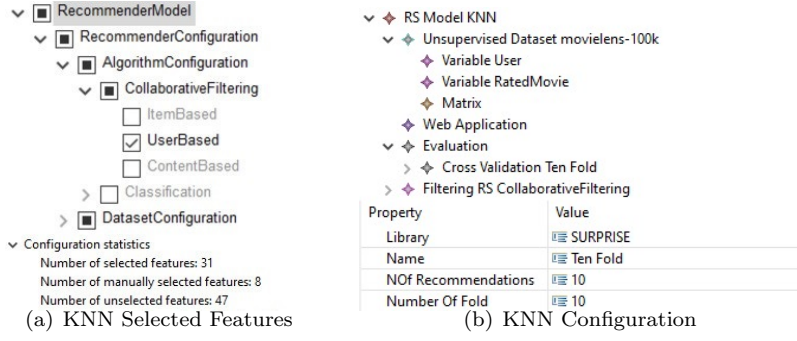


Fig. 10 Small fragment of the KNN specification.

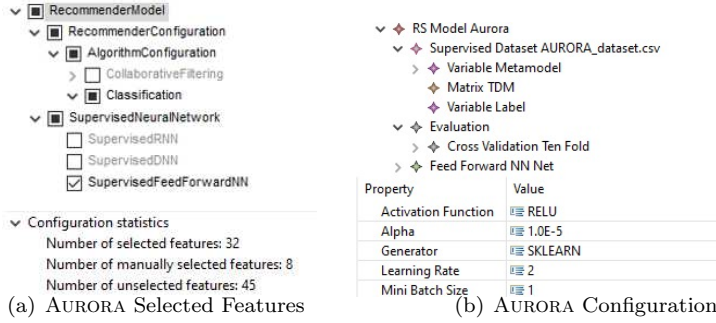


Fig. 11 Small fragment of the AURORA specification.

the underpinning algorithm, we defined a *FeedForward NN* concept to resemble the original behavior of AURORA.

4.2 RS configurations

Once the features of KNN and AURORA have been selected, as shown in the previous section, an initial version of their configuration models is automatically generated by LEV4REC. We refined such models to provide missing parameters. Concerning the KNN configuration, it includes the specification of a *Variable* element that represents the data sources exploited to perform the recommendations, i.e., the users and the rated movies.

After the dataset definition, the developer can configure the collaborative filtering algorithm by setting different parameters. The user can eventually set an *Evaluation Strategy* to assess the resulting system in terms of available *Metrics*. Moreover, as shown in the model fragment depicted in Fig. 10(b), we have specified the number of folds and number of wanted recommendations for the selected cross-validation method.

Concerning AURORA, the employed configuration is shown in Fig. 11(b). As we can see, the *Variable* concept has been used to describe *Metamodels* since AURORA was originally conceived to categorize them given an initial *Label*. In such a way, we can reuse the same notation to describe different algorithms and input data. Afterward, the RSD can customize the feed-forward neural network selected in the previous phase, i.e., the RS feature selection, according to the available hyperparameters. For instance, a possible fine-tuning is depicted in the lower part of Fig. 11(b) where the *learning rate* has been set to 2 and *mini batch size* to 1. Since LEV4REC targets the Sklearn library when generating the actual code, the user can also set an activation function chosen among the available ones, i.e., *identity*, *logistic*, *tanh*, and *relu*.

Listing 2 Excerpt of the generated code for KNN.

```
# DATASET
import pandas as pd
from surprise import Dataset
data = Dataset.load_builtin('ml-100k')
# ALGORITHM SETTINGS
is_user_based=False
neighborhood=40
cutoff=5
sim_funct='cosine'
sim_settings = {'name': sim_funct,
'user_based': is_user_based}
from surprise import KNNWithMeans
algo = KNNWithMeans(k=neighborhood, sim_options=sim_settings)
# EVALUATION SETTINGS
from surprise.model_selection import KFold
from collections import defaultdict
threshold = 3.5
k=10
n_splits=10
kf = KFold(n_splits=n_splits)
for trainset, testset in kf.split(data):
    algo.fit(trainset)
    predictions = algo.test(testset)

    user_est_true = defaultdict(list)
    for uid, _, true_r, est, _ in predictions:
        user_est_true[uid].append((est, true_r))
    precisions = dict()
    ...
    precisions[uid] = n_rel_and_rec_k / n_rec_k if n_rec_k != 0 else 0
    recalls[uid] = n_rel_and_rec_k / n_rel if n_rel != 0 else 0

    precision= sum(prec for prec in precisions.values()) / len(
        precisions)
    recall =sum(rec for rec in recalls.values()) / len(recalls)
    f1_measure=(2*precision* recall) / (recall + precision)
```

4.3 Generation of RS source code

Listing 3 Excerpt of the generated code for AURORA.

```
# DATASET
dataset=pd.read_csv('AURORA_train.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values
#PREPROCESSING
from sklearn.feature_extraction.text import CountVectorizer
sc = CountVectorizer(ngram_range=(1,1))
# ALGORITHM SETTINGS
from sklearn.neural_network import MLPClassifier
solver='adam'
alpha=1e-5
hidden_layers=(5, 2)
random_state=1
clf = MLPClassifier(solver=solver, alpha=alpha,
                    hidden_layer_sizes=hidden_layers, random_state=random_state)
# EVALUATION SETTINGS
n_splits=10
...
from sklearn.model_selection import KFold
kf = KFold(n_splits = n_splits)
for train, test in kf.split(X):
    X_split, X_test, y_split, y_test = X[train],X[test], y[train], y
    [test]
...
X_train = sc.fit_transform(list_train)
X_test = sc.transform(list_test)
clf.fit(X_train, y_split)
y_pred = clf.predict(X_test)
from sklearn.metrics import precision_score
precision = precision_score(y_pred, y_test, average=None)
prec_all = prec_all + sum(precision)/len(precision)
...
from sklearn.metrics import f1_score
f1 = f1_score(y_pred, y_test, average=None)
f1_all = f1_all + sum(f1)/len(f1)
```

Once the configuration model has been finalized, LEV4REC can generate the source code of the modeled RS. The platform exploits Acceleo templates to produce the Python code as described in Section 3.3. This section explains how LEV4REC generates code for the two considered systems. As previously mentioned, currently, we support two different libraries to cover the two use cases, i.e., Scikit-Learn and Surprise. Our tool can be conveniently extended to work with other Machine Learning frameworks, such as TensorFlow or Keras.

The LEV4REC code generator produces the needed Python scripts by adhering to a predefined structure. First, the dataset loading is performed by relying on the *pandas* library. Then, LEV4REC configures the algorithm parameters according to specified values in the design phase. The generated RS eventually evaluates the algorithm by computing the metrics previously selected by the designer.

Listing 2 and Listing 3 represent fragments of the the generated code for KNN and AURORA, respectively. Concerning the former, we exploit the Surprise library as it supports a set of well-defined collaborative algorithms. We choose the K-NN mean algorithm that resembles the original CROSSREC settings. The resulting source code is then used to run experiments on a real-world dataset. Afterward, the evaluation phase is conducted to study the results using common quality metrics in Information Retrieval [36], i.e., precision, recall, and F_1 score.¹⁸

Listing 3 depicts the generated code for AURORA. Similar to KNN, we loaded a predefined dataset to test the capability to resemble the system’s structure. The underlying feed-forward neural network is implemented by exploiting the Scikit-learn MLP class that provides similar features. The performance is assessed using the cross-validation technique to compute the above-mentioned metrics.

The presented code is inserted in a Jupyter Notebook, a well-known open-source format that can be executed directly on several platforms, e.g., JupyterLab,¹⁹ or Google Colaboratory.²⁰ In such a way, LEV4REC provides an agnostic representation that allows for flexibility in choosing the executing environment.

5 Evaluation

To demonstrate the applicability of our approach, this section presents an empirical evaluation consisting of the reimplementing of different recommender systems and the comparison of the obtained results with the original implementations. We define the research question in Section 5.1 while Section 5.2 presents the datasets employed in the evaluation. The metrics computed are presented in Section 5.3 and the whole evaluation process is presented in Section 5.4.

5.1 Research question

We study the performance of our proposed approach by considering the following research question:

RQ: *Is LEV4REC capable of resembling the key functionalities of existing recommender systems?*

In this research question, we assess the capability of LEV4REC in resembling the key functionalities of two existing recommender systems that support software development, i.e., KNN and AURORA. To this end, we employ

¹⁸ For the sake of presentation, we omitted mathematical details in the metrics computation.

¹⁹ <https://jupyterlab.readthedocs.io/en/stable/>

²⁰ <https://colab.research.google.com/notebooks/intro.ipynb>

LEV4REC to generate these systems from the design to the actual deployment. Afterward, we validate the obtained results by using a set of well-known metrics.

5.2 Datasets

To evaluate each system, we elicit two different datasets belonging to two different domains, i.e., movies, and MDE. The KNN algorithm has been validated on MovieLens dataset [26], a well-known *supervised dataset* widely used in the recommendation system domain to test collaborative filtering approaches. The data was collected through the MovieLens website²¹ and includes 100,000 ratings from 943 users on 1,682 movies plus several demographic user information, e.g., age, gender, and occupation.

We used the same dataset [8] that has been considered to evaluate AURORA while testing the LEV4REC's performance. The dataset comprises 555 meta-models with 9 different application domains, i.e., Bibliography, Conference management, Bug/issue tracker, Build systems, MS Office products, Requirement/use case, Database, State machines, and Petri nets.

Table 2 Overview of the examined datasets.

System	Dataset	Data type	Features
KNN	Movielens-100k	Matrix	1,682 movies rated by 943 users
AURORA	Ecore dataset	Textual Data	555 metamodels labeled with 9 categories

Table 2 summarizes the features of the examined dataset as well as the corresponding tools. As we can notice, the input data has different dimensions and format, i.e., AURORA has to be fed with textual data while a matrix format is enough for the KNN-based approach. Such a heterogeneous knowledge can be handled by LEV4REC by using the feature model and the high-level concepts as discussed in Section 3.

5.3 Metrics

To evaluate the generated systems, we consider precision, recall, and F1 score that are widely used in the information retrieval domain [27]. First, we define the following notations:²²

- *True positive (TP)*: is the outcome where the system recommends the proper item;
- *False Positive (FP)*: is the outcome where the system has suggested the wrong item;

²¹ <https://grouplens.org/datasets/movielens/100k/>

²² This notation holds for all the considered systems in the evaluation. Thus, the term *item* refers to a movie or a predicted label if KNN or AURORA is considered, respectively

- *False Negative (FN)*: is the outcome where the system doesn't recommend an item that should be included in the delivered list.

Precision: This metric evaluates the ratio of number of correctly predicted items to the total number of retrieved items

$$P = \frac{TP}{TP + FP} \quad (1)$$

Recall: It measures the impact of the false positive on the recommended items.

$$R = \frac{TP}{TP + FN} \quad (2)$$

F₁ score: It represents the harmonic mean of the two previous metrics.

$$F_1 \text{ score} = \frac{2 \times P \times R}{P + R} \quad (3)$$

5.4 Evaluation methodology

The evaluation process conducted for the considered systems is depicted in Fig. 12. We adopt the ten-fold cross-validation technique consisting of splitting the *Initial data* of each tool into two different dataset, i.e., *train* and *test* data. The former is used to feed the examined *recommender system* implemented using LEV4REC. It is worth mentioning that each tool has been trained with data of different nature. For instance, the KNN algorithm needs a matrix composed of users, movies, and the corresponding rate. Meanwhile, AURORA is fed with a set of labeled metamodels by means of NLP encoding.

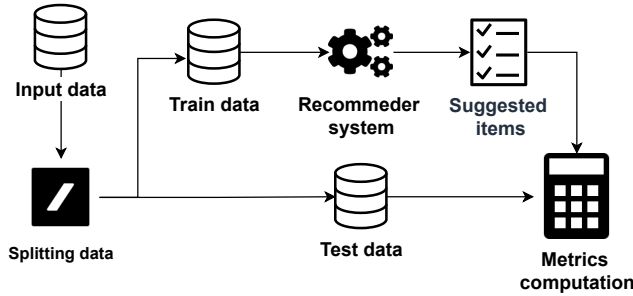


Fig. 12 The evaluation process.

The *test* data is used to validate the two systems. In particular, we used 80% of the initial data as training in the conducted study and the left part as testing. Afterward, the *Metrics computation* process evaluates the performances of the considered recommender system by computing the metrics described in Section 5.3, i.e., precision, recall, and F1 score. To this end, the *suggested item* returned by the system are compared with the elicited test data. It is worth noting that this process takes place for each evaluation round.

6 Results

This section analyzes the obtained results by answering the research question introduced in Section 5.1.

RQ: *Is LEV4REC capable of resembling the key functionalities of existing recommender systems?*

To answer the research question, we run the two recommender systems implemented using LEV4REC on the datasets presented in Section 5.2. Figure 13 reports the experimental results by running KNN on the MovieLens dataset. It is worth mentioning that the KNN performance resembles the results obtained with the standard version of the algorithm, i.e., precision, recall, and F1 scores are ≈ 0.60 with the better configuration. However, it is important to remark that our approach cannot resemble the augmented version of the KNN since it relies on a tailored algorithm at the current state of development. Thus, we focus on deploying the standard version of the KNN, which was considered as the baseline in the mentioned work [52].

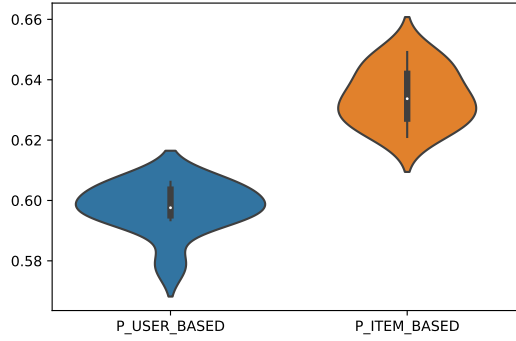


Fig. 13 Precision of the KNN-based RS developed with LEV4REC.

To highlight the LEV4REC's capabilities in fine-tuning the chosen algorithms, we compare the results obtained with user-based and item-based collaborative filtering. The item-based collaborative filtering technique can improve prediction than the user-based one for all considered metrics. For example, as depicted in Fig. 13, the precision distribution is centered on 0.64 with the item-based algorithm while the user-based median is 0.59, meaning that it obtains the worst results on average. A similar trend can be observed for the recall and F1 metrics which achieve 0.56 and 0.59 using the item-based technique, as depicted in Fig. 14 and Fig. 15, respectively. Meanwhile, the user-based median values for the recall and F1 are lower, i.e., 0.52 and 0.56. Altogether, the item-based strategy achieves better results, and this is consistent with the findings of existing work [31, 13]. Thus, we conclude that the produced implementation for KNN can provide recommendations as expected.

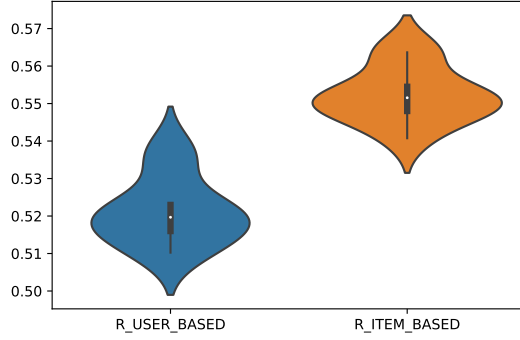


Fig. 14 Recall of the KNN-based RS developed with LEV4REC.

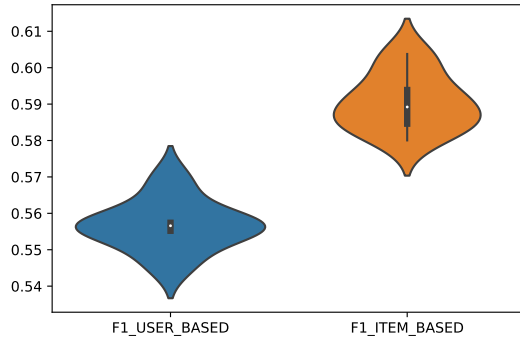


Fig. 15 F_1 score of the KNN-based RS developed with LEV4REC.

The results achieved by the reimplementation of AURORA done with the proposed LEV4REC approach are shown in Fig. 16, Fig. 17, and Fig. 18. It is worth noting that LEV4REC succeeded in resembling the AURORA underpinning algorithm used in the classification task: the values of precision, recall, and F_1 score are similar to those reported in the original work [41]. Furthermore, according to the work presenting AURORA, the best configuration is reached using uni-gram encoding, which has been resembled by LEV4REC by using the `Count Vectorizer` class provided by Sklearn. In particular, our findings confirm that the proposed environment can mimic AURORA’s functionalities, i.e., the original tool achieves better performance when the uni-gram encoding is used. Furthermore, concerning the examined metrics, the results are very high since we are considering a well-curated dataset composed of metamodels that are very similar. Nevertheless, the conducted study aims to evaluate to what extent LEV4REC can conceive the original design of the examined tools rather than improving their performances.

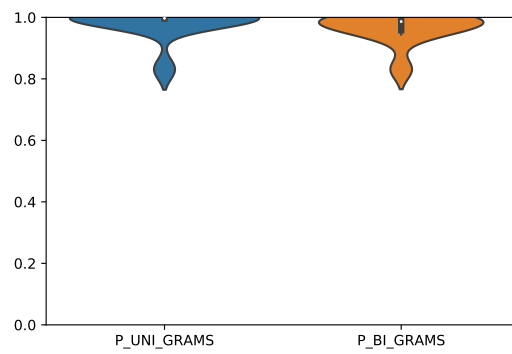


Fig. 16 Precision of AURORA as developed with LEV4REC.

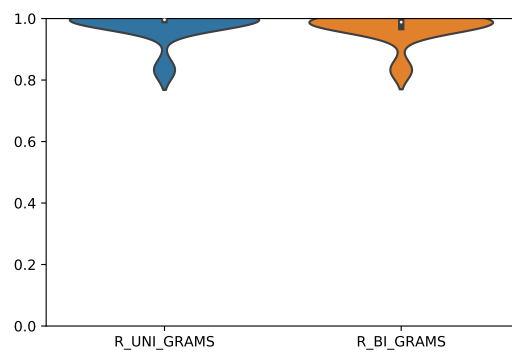


Fig. 17 Recall of AURORA as developed with LEV4REC.

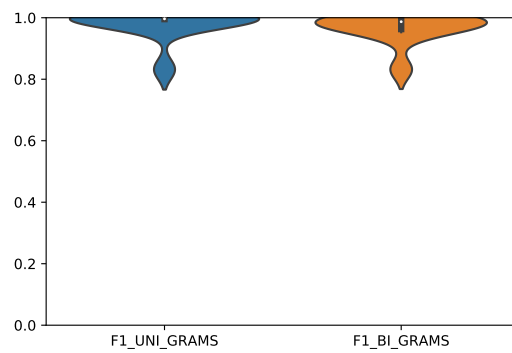


Fig. 18 F_1 score of AURORA as developed with LEV4REC.

Table 3 compares the results obtained by the original tools, i.e., KNN and AURORA, and the corresponding ones that have been developed with

LEV4REC. It is worth noting that the proposed approach achieves almost the same values for all the examined metrics on average.

Table 3 Comparison with original results.

Avg. metrics	Collaborative filtering		Classification	
	KNN	LEV4REC	AURORA	LEV4REC
Precision	0.623	0.634	0.945	0.950
Recall	0.622	0.552	0.938	0.949
F1	0.623	0.590	0.942	0.949

Answer to RQ. The conducted evaluation suggests that LEV4REC can resemble existing recommender systems geared by different algorithms. Furthermore, the obtained results are comparable to those presented in the original studies.

7 Threats to validity

This section discusses some threats that may affect the validity of our findings and identifies possible countermeasures to minimize the potential issues.

Threats to *internal validity* concern the selected methodology to generate the RS, i.e., the devised low-code environment. An RS developer who is not familiar with modeling concepts and the Eclipse environment could have some difficulties in using LEV4REC. We provide a constraint-based mechanism to support users while selecting the RS features to mitigate such an issue. Also, the RS configuration phase is supported by a metamodel, which enforces a well-defined structure to the specifications being defined.

External validity is related to the generalizability of the approach. This work integrated the Surprise and Scikit-learn utilities even though many libraries might be alternatively used. However, LEV4REC can be extended to support different algorithms and programming languages. Finally, selecting just two approaches for the evaluation could be seen as a threat to the generalizability of LEV4REC. However, the two systems correspond to two main algorithms in the recommender system domain, i.e., collaborative filtering technique and feed-forward neural network.

8 Related work

This section makes an overview of existing approaches supporting the development of RSs (Section 8.1), and integrating them into different IDEs (Section 8.2).

8.1 Supporting the design of recommender systems

The most related work to LEV4REC is Elliot [6], which provides RS designers with a practical means to design, implement, and evaluate a recommender system. Starting from a configuration file written in YAML, the tool produces the desired system by supporting all the needed phases, i.e., dataset loading, filtering, splitting, or retrieving recommendations. To this end, Elliot offers different models that the user can modify by selecting splitting rules, parameters, and evaluation metrics. The system eventually retrieves the outcomes for each system. Compared to Elliot, LEV4REC provides users with a model-based environment, which supports the users during the whole development process, from selecting the wanted components to the automated generation of the final source code. Furthermore, the adoption of feature models and constraints reduces the burden of choosing suitable reusable components and transparently managing their compatibility for the user.

Almonte *et al.* [2] present a generic low-code approach to simplify the development of an RS by employing model-driven engineering. Starting from a predefined metamodel, the system can generate a model and a DSL that expresses relevant concepts to configure the building blocks of an RS. To generate the results, the proposed DSL is built on top of the RankSys external library [57] that provides all the necessary utilities to create, execute and test collaborative filtering RSs. Compared to the framework presented in [2], LEV4REC supports the whole RS development and evaluation process, and it is not limited to collaborative filtering techniques. The same authors propose DROID, a model-based tool that makes use of a dedicated DSL to configure and deploy different RSs into modeling frameworks [4]. The first step is the definition of the key operations needed to conceive the system, i.e., candidate recommender methods, the training dataset, and the evaluation metrics. Afterward, the system automatically evaluates each recommendation method using the specified metrics. DROID eventually synthesizes an RS service that can be integrated into different modeling tools. With respect to DROID, we aim to realize an agnostic environment specifically dedicated to designing and deploying recommender systems using different IDEs rather than modeling frameworks.

Recently, the Lavoisier tool has been conceived to automate the data management pipeline typically employed in data analysis applications [12]. By relying on a textual DSL specified through Xtext, the Lavoisier language allows the user to specify the mining algorithm that is used to extract data in a tabular format. To this end, the proposed approach offers high-level primitives to select and rearrange any specific-domain data. In such a way, the low-level data transformation operations are completely handled by Lavoisier that produces tabular datasets ready to be digested by mining algorithms with less effort. Since LEV4REC does not support this specific phase, we consider integrating similar tools in our approach as possible future work.

Similar to LEV4REC, RECOLIBRY SUITE [30] drives the user during the design and deployment of a recommender system from scratch by exploiting a set of existing intelligent tools. The design phase is supported by the

RECOLIBRY-CORE component that offers a catalog of ready-to-use recommender algorithms, i.e., collaborative filtering, machine learning-based strategy. Once the user has selected the system architecture, the deployment phase is realized by using **RECOLIBRY-STUDIO** application that allows the specification of the desired system in terms of input and output. Both the aforementioned intelligent components make use of a common ontology to verify possible constraints on the specified system components. **Reco server** component eventually deploys the system as an external web service that can execute the generated code. Compared to **RECOLIBRY SUITE** that supports the definition of a recommender system within the provided environment, **LEV4REC** goes one step further as it is equipped with a code generation module that is agnostic with respect to the domain, i.e., the user can export directly the generated system without relying on a specific platform. Furthermore, our tool provides an extension mechanism that allows the user to specify new components besides the existing ones using feature model specification and a custom domain syntax language, fostering the personalization of black-box components.

A recurrent neural network (RNN) based on the Gated Recurrent Unit (GRU) technique has been used to design a general-purpose recommender system [34]. The approach employs ontologies to represent users' profiles by collecting external knowledge. The extracted data is then used to feed the underpinning model to produce a recommender system that fulfills the user's specification. Like **LEV4REC**, the proposed method also has the underlying concepts to generate RS's configuration. However, our code generator is more extensible as the employed **Acceleo** module is language-agnostic, i.e., the final user can select the wanted programming language to generate the actual code.

ktrain [35] is a low-code Python library that supports AutoML by relying on various Machine Learning frameworks, e.g., TensorFlow, and Keras. Using the proposed system, a non-expert user can select and customize different models and estimate their learning rate to fine-tune them. At the time of writing, *ktrain* covers text and image classification tasks by including well-known ML datasets and proper preprocessing techniques. Similarly, the *fastai* [28] and Ludwig [38] frameworks support various cutting-edge technologies to speed up the development of an ML application from scratch. *fastai* focuses on the deep learning domain by employing a layered structure composed of high, mid, and low-level Python API. In such a way, end-users with different skill levels can profitably practice complex deep learning utilities. Meanwhile, Ludwig employs a tailored declarative language to design a complete ML workflow. The system can create, optimize, and evaluate several ML off-the-shelf approaches by automatically generating a complete pipeline starting from an initial user specification.

Recently, the **AutoRec** platform [58] has been proposed to automatize the deployment of RSs based on deep neural networks. The system uses TensorFlow to implement a highly flexible pipeline that allows for model selection and hyperparameter tuning. **AutoRec** exposes all the supported utilities using a user-friendly API to build the designed system. Similarly, **Auto-Surprise** [5] tool is built on top of the **Surprise** Python library to automatize the selection

and the evaluation of 11 different algorithms implemented by the library, e.g., SVD, KNN-means to name a few. The approach employs a sequential model-based optimization strategy that evaluates the algorithms mentioned above in parallel to select the best one given the initial configuration.

We embedded only two Python libraries in the current implementation, i.e., Scikit-learn, and Surprise in LEV4REC. Providing a low-code environment that fulfills all possible tools and libraries is out of the scope of our work. Nevertheless, our approach has been designed to support a high-level recommender system architecture that could involve miscellaneous technologies alongside ML models, e.g., collaborative filtering, and data mining. Furthermore, we plan to incorporate further Machine Learning models and frameworks in our future work, e.g., fasttext, and BERT.

8.2 Integrating recommender systems into IDEs

Extremo [39] is a recommender system to support modeling activities integrated as an Eclipse plugin that relies on heterogeneous information obtained from different sources. A common data model is devoted to displaying this information in a dedicated repository IDE view. Users can browse and filter the results employing a search wizard component. Extremo can eventually be used to create DSLs for tailored domains. Similarly, a Papyrus plugin [20] aims to assist modelers in designing domain-specific models that conform to the OMG UML 2 standard. By relying on an EMF generator model, and UML profiling, the deployed plugin can generate the corresponding Java code given the model specification. Furthermore, the tool supports the creation of viewpoints and CSS-style sheets that could be used to customize the generated DSL.

The *Recombee* framework²³ offers a recommender system as a service to support several domains, e.g., multimedia content, job boards, feed aggregators. To enable recommendations, the system uses a set of item catalogs as well as user attributes and real-time interactions. *Recombee* exposes all the supported features as SDKs client libraries for different languages, i.e., Java, Python, Node.js, to name a few. Challenges in designing and deploying code completion IDE plugins have been highlighted in a recent work [60]. The authors conducted an extensive user study and developed a prototype code completion recommender system named tranX-plugin²⁴ for PyCharm IDE to suggest relevant Python code given a textual query. The outcomes indicate that the current IDE code assistants have some limitations in terms of correctness and quality even though developers provide several suggestions for improvement. Recently, GitHub developed Copilot²⁵ – an AI-based recommender system deployed as a VS Code extension that analyzes developers’ context to suggest relevant code elements, i.e., missing API function calls. To this end, the underpinning AI model has been trained on publicly available

²³ <https://www.recombee.com/>

²⁴ <https://github.com/neulab/tranX-plugin>

²⁵ <https://copilot.github.com/>

source code and natural language content. However, according to the current programming task, the retrieved suggestions need to be eventually adapted by the developer.

Concerning IDE integrations, LEV4REC can be extended to support different systems apart from Eclipse and VS Code by relying on the guidelines described in Section 3.4. In particular, we provide an interface that relies on three components, namely the feature model, metamodel, and Acceleo templates. Furthermore, the feature model and metamodel can be augmented with additional concepts not yet supported in the proposed tool. In this way, once the essential blocks have been properly designed, the developer can define custom Acceleo templates to realize the required business logic.

9 Conclusion and future work

Due to the urgent need to support the development of recommender systems, LEV4REC has been proposed as a workable solution to assist developers that do not have strong experience in designing and programming recommender systems. Our approach is a low-code environment to foster an RS's design, configuration, and deployment from scratch using such a cutting-edge paradigm. LEV4REC is flexible and extensible as it relies on three core techniques, i.e., feature model, metamodel, and Acceleo templates. Starting from a feature model, RS designers can specify the system's features and then progressively enrich a configuration model automatically generated out of the selected features. Once the RS configuration has been refined, the system employs a model-driven code generator to produce the actual code of the specified RS. LEV4REC allows developers to refine the produced system by experimenting with different algorithms, experimental settings, and evaluation metrics.

We evaluated the approach empirically by reimplementing two existing RSs that rely on different algorithms, i.e., collaborative filtering technique, and feed-forward neural network. At its current implementation, the platform supports the specification of configurations and the evaluation exploiting two different Python libraries. We plan to improve our conceived tool for future work by equipping with it the ability to generate code in other languages, e.g., Java. Furthermore, we will also perform more evaluations by incorporating other recommender systems. We plan to cover the missing components, e.g., dataset generation. Moreover, we will develop a proper extensible mechanism to allow RS designers to specify new components in terms of e.g., new Eclipse plugins.

Acknowledgements The research described in this paper has been partially supported by the AIDOaRT Project, which has received funding from the European Union's H2020-ECSEL-2020, Federal Ministry of Education, Science and Research, Grant Agreement n° 101007350.

References

1. Adomavicius, G., Zhang, J.: Impact of data characteristics on recommender systems performance. *ACM Transactions on Management Information Systems (TMIS)* **3**(1), 1–17 (2012)
2. Almonte, L., Cantador, I., Guerra, E., de Lara, J.: Towards automating the construction of recommender systems for low-code development platforms. In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '20*, pp. 1–10. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3417990.3420200. URL <http://doi.org/10.1145/3417990.3420200>
3. Almonte, L., Guerra, E., Cantador, I., de Lara, J.: Recommender systems in model-driven engineering. *Software and Systems Modeling* (2021). DOI 10.1007/s10270-021-00905-x. URL <https://doi.org/10.1007/s10270-021-00905-x>
4. Almonte, L., Pérez-Soler, S., Guerra, E., Cantador, I., de Lara, J.: Automating the Synthesis of Recommender Systems for Modelling Languages p. 14 (2021)
5. Anand, R., Beel, J.: Auto-surprise: An automated recommender-system (autorecsys) library with tree of parzens estimator (tpe) optimization. In: *Fourteenth ACM Conference on Recommender Systems, RecSys '20*, p. 585–587. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3383313.3411467. URL <https://doi.org/10.1145/3383313.3411467>
6. Anelli, V.W., Bellogin, A., Ferrara, A., Malitesta, D., Merra, F.A., Pomo, C., Donini, F.M., Di Noia, T.: Elliot: A comprehensive and rigorous framework for reproducible recommender systems evaluation. In: *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '21*, p. 2405–2414. Association for Computing Machinery, New York, NY, USA (2021). DOI 10.1145/3404835.3463245. URL <https://doi.org/10.1145/3404835.3463245>
7. Arnoldus, B., Brand, van den, M., Serebrenik, A., Brunekreef, J.: Code generation with templates. *Atlantis studies in computing*. Atlantis Press, Netherlands (2012). DOI 10.2991/978-94-91216-56-5
8. Önder Babur: A labeled Ecore metamodel dataset for domain clustering (2019). DOI 10.5281/zenodo.2585456. URL <https://doi.org/10.5281/zenodo.2585456>
9. Baxter, P., Jack, S.M.: Qualitative case study methodology: Study design and implementation for novice researchers. *The Qualitative Report* **13**, 544–559 (2008)
10. Bobadilla, J., Ortega, F., Hernando, A., Gutiérrez, A.: Recommender systems survey. *Knowledge-Based Systems* **46**, 109–132 (2013). DOI 10.1016/j.knosys.2013.03.012
11. Dagenais, B., Ossher, H., Bellamy, R.K.E., Robillard, M.P., de Vries, J.P.: Moving into a new software project landscape. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, pp. 275–284. ACM, New York, NY, USA (2010). DOI 10.1145/1806799.1806842. URL <http://doi.acm.org/10.1145/1806799.1806842>
12. de la Vega, A., García-Saiz, D., Zorrilla, M., Sánchez, P.: Lavoisier: A dsl for increasing the level of abstraction of data selection and formatting in data mining. *Journal of Computer Languages* **60**, 100987 (2020). DOI <https://doi.org/10.1016/j.cola.2020.100987>. URL <https://www.sciencedirect.com/science/article/pii/S2590118420300472>
13. Deshpande, M., Karypis, G.: Item-based top- k recommendation algorithms. *ACM Trans. Inf. Syst.* **22**(1), 143–177 (2004). DOI 10.1145/963770.963776. URL <https://doi.org/10.1145/963770.963776>
14. Di Rocco, J., Di Ruscio, D., Di Sipio, C., Nguyen, P.T., Rubel, R.: Development of recommendation systems for software engineering: the crossminer experience. *Empirical Software Engineering Journal* (2021). URL <https://arxiv.org/abs/2103.06987>
15. Di Sipio, C., Di Rocco, J., Di Ruscio, D., Nguyen, P.T.: A low-code tool supporting the development of recommender systems. In: *Fifteenth ACM Conference on Recommender Systems, RecSys '21*, p. 741–744. Association for Computing Machinery, New York, NY, USA (2021). DOI 10.1145/3460231.3478885. URL <https://doi.org/10.1145/3460231.3478885>
16. Di Sipio, C., Di Ruscio, D., Nguyen, P.T.: Democratizing the development of recommender systems by means of low-code platforms. In: *Proceedings of the 23rd*

- ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '20. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3417990.3420202. URL <https://doi.org/10.1145/3417990.3420202>
17. Di Sipio, C., Rubei, R., Di Ruscio, D., Nguyen, P.T.: A multinomial naïve bayesian (mnb) network to automatically recommend topics for github repositories. In: Proceedings of the Evaluation and Assessment in Software Engineering, EASE '20, p. 71–80. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3383219.3383227. URL <https://doi.org/10.1145/3383219.3383227>
 18. Diao, Q., Qiu, M., Wu, C.Y., Smola, A.J., Jiang, J., Wang, C.: Jointly modeling aspects, ratings and sentiments for movie recommendation (jmars). In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, p. 193–202. Association for Computing Machinery, New York, NY, USA (2014). DOI 10.1145/2623330.2623758. URL <https://doi.org/10.1145/2623330.2623758>
 19. Dueñas, S., Cosentino, V., Robles, G., Gonzalez-Barahona, J.M.: Perceval: software project data at your will. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, pp. 1–4. ACM (2018)
 20. Dupont, G., Mustafiz, S., Khendek, F., Toeroe, M.: Building Domain-Specific Modelling Environments with Papyrus: An Experience Report. In: 2018 IEEE/ACM 10th International Workshop on Modelling in Software Engineering (MiSE), pp. 49–56 (2018). ISSN: 2575-4475
 21. Ekstrand, M.D.: LensKit for Python: Next-Generation Software for Recommender Systems Experiments. In: Proceedings of the 29th ACM International Conference on Information & Knowledge Management, pp. 2999–3006. ACM, Virtual Event Ireland (2020). DOI 10.1145/3340531.3412778
 22. Felfernig, A., Le, V.M., Popescu, A., Uta, M., Tran, T.N.T., Atas, M.: An overview of recommender systems and machine learning in feature modeling and configuration. In: 15th International Working Conference on Variability Modelling of Software-Intensive Systems, VaMoS'21. Association for Computing Machinery, New York, NY, USA (2021). DOI 10.1145/3442391.3442408. URL <https://doi-org.univaq.clas.cineca.it/10.1145/3442391.3442408>
 23. Gantner, Z., Rendle, S., Freudenthaler, C., Schmidt-Thieme, L.: Mymedialite: A free recommender system library. In: Proceedings of the Fifth ACM Conference on Recommender Systems, RecSys '11, p. 305–308. Association for Computing Machinery, New York, NY, USA (2011). DOI 10.1145/2043932.2043989. URL <https://doi-org.univaq.clas.cineca.it/10.1145/2043932.2043989>
 24. Gasparic, M., Janes, A., Ricci, F., Murphy, G.C., Gurbanov, T.: A graphical user interface for presenting integrated development environment command recommendations: Design, evaluation, and implementation. *Information and Software Technology* **92**, 236–255 (2017). DOI <https://doi.org/10.1016/j.infsof.2017.08.006>. URL <https://www.sciencedirect.com/science/article/pii/S0950584916303524>
 25. Gronback, R.: Eclipse Modeling Project | The Eclipse Foundation (2021). URL <https://www.eclipse.org/modeling/emf/>
 26. Harper, F.M., Konstan, J.A.: The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.* **5**(4) (2015). DOI 10.1145/2827872. URL <https://doi.org/10.1145/2827872>
 27. Herlocker, J.L., Konstan, J.A., Terveen, L.G., Riedl, J.T.: Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.* **22**(1), 5–53 (2004). DOI 10.1145/963770.963772. URL <https://doi.org/10.1145/963770.963772>
 28. Howard, J., Gugger, S.: fastai: A Layered API for Deep Learning. *Information* **11**(2), 108 (2020). DOI 10.3390/info11020108. URL <http://arxiv.org/abs/2002.04688>. ArXiv: 2002.04688
 29. Hug, N.: Surprise: A Python library for recommender systems. *Journal of Open Source Software* **5**(52), 2174 (2020). DOI 10.21105/joss.02174. URL <https://joss.theoj.org/papers/10.21105/joss.02174>
 30. Jorro-Aragoneses, J.L., Díaz-Agudo, B., Recio-García, J.A., Jimenez-Díaz, G.: RecoLibry Suite: a set of intelligent tools for the development of recommender systems. *Auto-*

- mated Software Engineering **27**(1-2), 63–89 (2020). DOI 10.1007/s10515-020-00269-4. URL <http://link.springer.com/10.1007/s10515-020-00269-4>
31. Karypis, G.: Evaluation of item-based top-n recommendation algorithms. In: Procs. of the tenth international conf. on information and knowledge management, CIKM '01, pp. 247–254. ACM, New York, NY, USA (2001)
 32. Knijnenburg, B.P., Reijmer, N.J., Willemsen, M.C.: Each to his own: How different users call for different interaction methods in recommender systems. In: Proceedings of the Fifth ACM Conference on Recommender Systems, RecSys '11, p. 141–148. Association for Computing Machinery, New York, NY, USA (2011). DOI 10.1145/2043932.2043960. URL <https://doi.org/10.1145/2043932.2043960>
 33. Kolovos, D., Neubauer, P., Barmpis, K., Matragkas, N., Paige, R.: Crossflow: a framework for distributed mining of software repositories. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 155–159. IEEE (2019)
 34. Korotaev, A., Lyadova, L.: Method for the Development of Recommendation Systems, Customizable to Domains, with Deep GRU Network:. In: Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, pp. 231–236. SCITEPRESS - Science and Technology Publications, Seville, Spain (2018). DOI 10.5220/0006933302310236. URL <http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006933302310236>
 35. Maiya, A.S.: ktrain: A Low-Code Library for Augmented Machine Learning. arXiv:2004.10703 [cs] (2020). URL <http://arxiv.org/abs/2004.10703>. ArXiv: 2004.10703
 36. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, USA (2008)
 37. McKinney, W., et al.: pandas: a foundational python library for data analysis and statistics. Python for High Performance and Scientific Computing **14**(9), 1–9 (2011)
 38. Molino, P., Dudin, Y., Miryala, S.S.: Ludwig: a type-based declarative deep learning toolbox. arXiv:1909.07930 [cs, stat] (2019). URL <http://arxiv.org/abs/1909.07930>. ArXiv: 1909.07930
 39. Mora Segura, A., de Lara, J.: Extremo: An Eclipse plugin for modelling and meta-modelling assistance. Science of Computer Programming **180**, 71–80 (2019). DOI 10.1016/j.scico.2019.05.003. URL <https://www.sciencedirect.com/science/article/pii/S0167642319300644>
 40. Murphy, G.C.: Attacking information overload in software development. In: IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2009, Corvallis, OR, USA, 20-24 September 2009, Proceedings, p. 4 (2009)
 41. Nguyen, P.T., Di Rocco, J., Di Ruscio, D., Pierantonio, A., Iovino, L.: Automated classification of metamodel repositories: A machine learning approach. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 272–282 (2019). DOI 10.1109/MODELS.2019.00011
 42. Nguyen, P.T., Di Rocco, J., Di Sipio, C., Di Ruscio, D., Di Penta, M.: Recommending api function calls and code snippets to support software development. IEEE Transactions on Software Engineering pp. 1–1 (2021). DOI 10.1109/TSE.2021.3059907
 43. Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., Lanza, M.: Prompter: Turning the IDE into a self-confident programming assistant. Empirical Software Engineering **21**(5), 2190–2231 (2016). DOI 10.1007/s10664-015-9397-1. URL <http://link.springer.com/10.1007/s10664-015-9397-1>
 44. Proksch, S., Bauer, V., Murphy, G.C.: How to build a recommendation system for software engineering. In: B. Meyer, M. Nordio (eds.) Software Engineering - International Summer Schools, LASER 2013-2014, Elba, Italy, Revised Tutorial Lectures, *Lecture Notes in Computer Science*, vol. 8987, pp. 1–42. Springer (2014). DOI 10.1007/978-3-319-28406-4_1. URL https://doi.org/10.1007/978-3-319-28406-4_1
 45. Ricci, F., Rokach, L., Shapira, B.: Introduction to Recommender Systems Handbook, pp. 1–35. Springer US, Boston, MA (2011). DOI 10.1007/978-0-387-85820-3_1. URL https://doi.org/10.1007/978-0-387-85820-3_1
 46. Richardson, C., Rymer, J.R.: The forrester wave: Low-code development platforms, q2 2016 (2016). Technical report, Forrester Research

47. Robillard, M.P., Maalej, W., Walker, R.J., Zimmermann, T. (eds.): Recommendation Systems in Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). DOI 10.1007/978-3-642-45135-5. URL <http://link.springer.com/10.1007/978-3-642-45135-5>
48. Sahay, A., Indamutsa, A., Di Ruscio, D., Pierantonio, A.: Supporting the understanding and comparison of low-code development platforms. In: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 171–178. IEEE, Portoroz, Slovenia (2020). DOI 10.1109/SEAA51224.2020.00036
49. Said, A., Bellogín, A.: Comparative recommender system evaluation: Benchmarking recommendation frameworks. In: Proceedings of the 8th ACM Conference on Recommender Systems - RecSys '14, pp. 129–136. ACM Press, Foster City, Silicon Valley, California, USA (2014). DOI 10.1145/2645710.2645746
50. Sjöberg, D., Hannay, J., Hansen, O., Kampenes, V., Karahasanovic, A., Liborg, N.K., Rekdal, A.: A survey of controlled experiments in software engineering. IEEE Transactions on Software Engineering **31**(9), 733–753 (2005). DOI 10.1109/TSE.2005.97
51. Song, Q., Zhu, X., Wang, G., Sun, H., Jiang, H., Xue, C., Xu, B., Song, W.: A machine learning based software process model recommendation method. Journal of Systems and Software **118**, 85–100 (2016)
52. Subramaniaswamy, V., Logesh, R.: Adaptive KNN based Recommender System through Mining of User Preferences. Wireless Personal Communications **97**(2), 2229–2247 (2017). DOI 10.1007/s11277-017-4605-5. URL <http://link.springer.com/10.1007/s11277-017-4605-5>
53. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: Featureide: An extensible framework for feature-oriented software development. Science of Computer Programming **79**, 70–85 (2014)
54. Thum, T., Kastner, C., Erdweg, S., Siegmund, N.: Abstract features in feature modeling. In: 2011 15th International Software Product Line Conference, pp. 191–200 (2011). DOI 10.1109/SPLC.2011.53
55. Van Der Walt, S., Colbert, S.C., Varoquaux, G.: The numpy array: a structure for efficient numerical computation. Computing in science & engineering **13**(2), 22–30 (2011)
56. Vargas, S.: Novelty and diversity enhancement and evaluation in recommender systems and information retrieval. In: Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval, SIGIR '14, p. 1281. Association for Computing Machinery, New York, NY, USA (2014). DOI 10.1145/2600428.2610382. URL <https://doi.org/10.1145/2600428.2610382>
57. Vargas, S., Castells, P.: Rank and relevance in novelty and diversity metrics for recommender systems. In: Proceedings of the Fifth ACM Conference on Recommender Systems, RecSys '11, p. 109–116. Association for Computing Machinery, New York, NY, USA (2011). DOI 10.1145/2043932.2043955. URL <https://doi.org/10.1145/2043932.2043955>
58. Wang, T.H., Hu, X., Jin, H., Song, Q., Han, X., Liu, Z.: Autorec: An automated recommender system. In: Fourteenth ACM Conference on Recommender Systems, RecSys '20, p. 582–584. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3383313.3411529
59. Waszkowski, R.: Low-code platform for automating business processes in manufacturing. IFAC-PapersOnLine **52**(10), 376–381 (2019). DOI 10.1016/j.ifacol.2019.10.060
60. Xu, F.F., Vasilescu, B., Neubig, G.: In-ide code generation from natural language: Promise and challenges. CoRR **abs/2101.11149** (2021). URL <https://arxiv.org/abs/2101.11149>