Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Engineering Recommender Systems for Modelling Languages: Concept, Tool and Evaluation

Lissette Almonte · Esther Guerra · Iván Cantador · Juan de Lara

Received: date / Accepted: date

Abstract Recommender systems (RSs) are ubiquitous in all sorts of online applications, in areas like shopping, media broadcasting, travel and tourism, among many others. They are also common to help in software engineering tasks, including software modelling, where we are recently witnessing proposals to enrich modelling languages and environments with RSs. Modelling recommenders assist users in building models by suggesting items based on previous solutions to similar problems in the same domain. However, building a RS for a modelling language requires considerable effort and specialised knowledge.

To alleviate this problem, we propose an automated, model-driven approach to create RSs for modelling languages. The approach provides a domain-specific language called Droid to configure every aspect of the RS: the type of the recommended modelling elements, the gathering and preprocessing of training data, the recommendation method, and the metrics used to evaluate the created RS. The RS so configured can be deployed as a service, and we offer out-of-the-box integration with Eclipse modelling editors. Moreover, the language is extensible with new data sources and recommendation methods.

To assess the usefulness of our proposal, we report on two evaluations. The first one is an offline experiment measuring the precision, completeness and diversity of recommendations generated by several methods. The second is a user study – with 40 participants – to assess the perceived quality of the recommendations. The study also contributes with a novel evaluation methodology and metrics for RSs in model-driven engineering.

Keywords Recommender Systems · Modelling Languages · Model-driven Engineering · Domain-specific Languages · User Study

1 Introduction

Recommender systems (RSs) are information filtering systems that assist users in choosing from a potentially large collection of items. Their goal is to predict and

Computer Science Department
Universidad Autónoma de Madrid
E-mail: (Lissette.Almonte, Esther.Guerra, Ivan.Cantador, Juan.deLara)@uam.es

exploit the preferences of the user to offer a personalised list of items (Adomavicius and Tuzhilin, 2005). RSs are present within all sorts of platforms, providing suggestions on a variety of items, such as music and video on streaming platforms (e.g., Spotify and Netflix), or products to buy on e-commerce sites (e.g., Amazon).

RSs are also used to assist developers in software engineering activities (Robillard et al., 2010; Di Rocco et al., 2021). These include recommenders of code refactorings (Oliveira et al., 2019), API code examples (Abid et al., 2021), meaningful method names (Liu et al., 2022), or development team members (Tuarob et al., 2021), to name a few. Following this trend, proposals of RSs for software modelling and design tasks are starting to appear (Almonte et al., 2022b). Modelling is central to most engineering disciplines and essential for software engineering. Indeed, some software development paradigms like model-driven engineering (MDE) have models as the main assets of the development process (Brambilla et al., 2017). Therefore, mechanisms that help designers to find, create, complete, repair and reuse models based on existing knowledge are of great importance (Almonte et al., 2022b). For this reason, several researchers have proposed RSs for modelling tasks, such as model completion (Mazanek and Minas, 2009; Moha et al., 2010), model finding (Cerqueira et al., 2016; Matikainen et al., 2013), and model repair (Iovino et al., 2020; Neubauer et al., 2017). In this paper, we are interested in the creation of RSs that help completing existing models.

Building a RS for a modelling language involves several steps, most importantly selecting the data for training and testing the RS, preprocessing these data (e.g., to fix ambiguities or unify item names), configuring the recommendation algorithm, evaluating the RS with suitable metrics, and deploying the RS within a modelling tool. These tasks require specialised knowledge and high implementation effort. Moreover, software paradigms like MDE or low-code development (Di Ruscio et al., 2022) often need to create new modelling and domain-specific languages (DSLs) to express solutions in the targeted domain. Hence, given the potential variety of modelling languages and DSLs, mechanisms to facilitate the construction of RSs for them are required.

To tackle this challenge, we propose an MDE solution to automate the creation of RSs for modelling languages. It is supported by a tool, called Droid, which provides: (i) a DSL to configure the kind of items that the RS will recommend (e.g., attributes for class diagrams, activities for process models); (ii) an engine that automates the preprocessing of models for training and evaluation, and the evaluation of the candidate recommendation algorithms against configurable metrics; (iii) facilities to compare between different RS configurations and identify the most appropriate one; (iv) a generator that deploys the RS as a service, which heterogeneous modelling clients can integrate; and (v) an out-of-the-box integration of the RS with Eclipse modelling tools. Our solution is extensible, since it permits a light integration of new data sources, data encodings and recommendation algorithms.

This paper presents the approach and its supporting tool, and reports on the results of an offline evaluation and a user study that aim to answer the following research questions (RQs):

RQ1 *How precise, complete and diverse are the recommendations of* Droid *recommenders?*
RQ2 *How do users perceive the recommendations of* Droid *recommenders?*

RQ3 *How do the offline experiment results compare to the ones of the user study?*

This paper is an extension of our previous works (Almonte et al., 2020, 2021, 2022a), where we introduced Droid and performed a preliminary offline evaluation of its precision applied to UML models within two domains (literature and education). The novel contributions of the present paper are the following:

- We have made Droid's architecture extensible by means of extension points that facilitate the incorporation of new data sources, data encoding algorithms, and recommendation methods.
- We have integrated into Droid an additional recommendation method (context-aware collaborative filtering, as implemented in MemoRec (Di Rocco et al., 2023)) using one of the new extension points.
- We have performed an extended offline evaluation of the accuracy of the recommendations provided by Droid recommenders. In comparison to the preliminary evaluation reported in (Almonte et al., 2021), the new one uses a bigger training dataset to build the RSs (3× more models), and considers three recommendation domains (one more than in (Almonte et al., 2021)).
- We report on a novel user study that analyses the users' perception of the recommendations provided by Droid recommenders. This study considers both classical metrics of RSs as well as novel metrics specifically devised for model completion recommendations (serendipity, redundancy, contextualisation and generalisation). To the best of our knowledge, this is one of the first user studies of the model completion recommendation task, which permits analysing how well offline evaluations align with the users' perception for this particular case. This is an important task since, nowadays, most RSs for modelling are evaluated only using offline experiments (cf. Subsection 7.3).

The rest of the paper is organised as follows. First, Section 2 provides motivation for our approach, introducing a running example and background on RSs. Next, Section 3 overviews our proposal. Section 4 describes the Droid DSL, which is used to configure all aspects of the RS, and Section 5 gives details of its supporting tool. Section 6 reports on the results of the offline experiment and the user study, and answers the research questions. Finally, Section 7 analyses related research, and Section 8 ends with conclusions and future work.

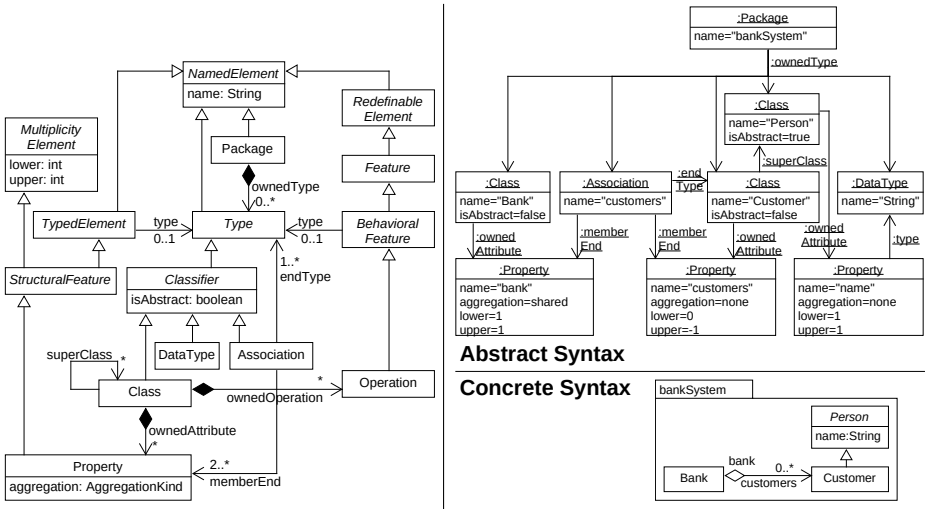
2 Motivation and Background

This section starts with a motivating example that will be used in the rest of the paper (Subsection 2.1). Then, it describes the contributions of our proposal (Subsection 2.2), and provides an overview of RSs (Subsection 2.3).

2.1 Motivating example

Assume we would like to build a RS to assist in the creation of UML class models by recommending attributes and operations for classes.

Models are defined using a modelling language. In MDE (Brambilla et al., 2017), it is a standard practice to describe the abstract syntax of modelling languages by means of a meta-model. Figure 1a shows an excerpt of the meta-model of UML 2.0 class diagrams (UML 2.5.1 OMG specification, 2017), which we use in our example. It structures UML class models into Packages that can contain Classes, Datatypes and Associations. Classes can declare attributes (class Property) and Operations, as well as inherit from other classes (relation Class.superClass). Figure 1b shows a UML model (i.e., an instance of the UML meta-model) in abstract syntax on top, and using the standard concrete syntax of class diagrams at the bottom.



(a) Excerpt of UML meta-model.

(b) UML class diagram example.

Fig. 1: UML meta-model excerpt and model in abstract and concrete syntax.

Figure 2 illustrates the behaviour of the RS that we aim to build. The left part of the figure shows a class model being created, which contains the classes Library and Book. To help the designer complete the target class Book, the RS would analyse similar classes previously defined, to return a ranked list of suitable new attributes and operations for the class. As an example, the figure shows a repository with previous models, among which one declares a class Journal. Since this class is "similar" to Book, the RS would propose adding Journal's attributes (publisher and topic) to Book.

Overall, the RS to build should recommend two types of items (attributes and operations) for a given target class. However, building such a RS by hand is costly. While there are well-known recommendation methods for building RSs (cf. Subsection 2.3), they need to be adapted for the modelling language (class models in our example) and task (recommendation of attributes and operations). Moreover, the performance of these methods may differ. For instance, one method may select classes with similar items to the ones in the target class, and recommend items from those classes; another method may recommend items that are similar to those the target class already has; while yet another method may recommend the

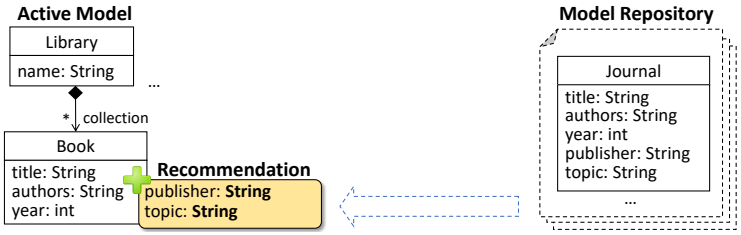


Fig. 2: Obtaining recommendations to assist in the creation of a UML class model.

most popular items (i.e., those appearing most frequently in classes). Selecting the best performing recommendation method for a modelling language needs to be an informed decision based on a set of metrics carefully chosen. However, computing those metrics by hand is costly. Finally, while some RSs for modelling have been proposed (Almonte et al., 2022b; Burgueño et al., 2021; Di Rocco et al., 2023; Di Rocco et al., 2021), they are typically hardwired to a specific modelling language and cannot be used for other languages, even if they are similar.

2.2 Contribution of Droid

To overcome the previous limitations, we propose Droid: a model-driven solution to facilitate the creation and evaluation of RSs that assist in creating models of a modelling language. Droid does not require deep knowledge of RSs or programming. To create a RS with Droid, the user only needs to provide a meta-model of the modelling language, and a dataset of models. Droid integrates recommendation methods that can be configured for a modelling language by specifying the target of the recommendation (e.g., UML classes) and the items to be recommended (e.g., attributes and operations). The performance of the recommendation methods can be assessed by the automatic calculation of metrics on the provided dataset. Finally, Droid generates automatically RSs for the modelling language, providing out-of-the-box integration with modelling environments within Eclipse.

Droid allows creating RSs for modelling languages independently of their visual or textual concrete syntax. This is so as it relies on the abstract syntax of the languages. The RSs created with Droid can recommend objects – instances of the meta-classes in the language meta-model – together with values for their fields. The RS in our running example suggests attributes and operations for UML classes, but with Droid, it would be easy to customise it to recommend, e.g., classes for packages, or superclasses for a given class.

In Figure 2, recommending an attribute for a class entails a basic editing operation (adding an object to another one in the underlying abstract syntax representation of the model). However, the difficulty relies on filtering which attributes are good recommendations for a given class. For example, an attribute `power` would not be appropriate for the class `Book`. RSs learn to make good recommendations by processing datasets of existing models (cf. Subsection 2.3). Droid permits configuring the RSs for arbitrary languages, targets and items.

Droid supports several recommendation methods, but new methods can be added. These include methods that extract information from existing datasets

of models, but also rule-based methods. The former methods are suitable when it is necessary to add new labelled information in the model (e.g., an attribute author for a class Book). The latter methods are adequate to make a model comply with certain syntactical laws or guidelines (e.g., avoid abstract classes without children). As a limitation, currently, DROID cannot recommend graphs made of related objects (e.g., a design pattern in the case of UML class diagrams).

DROID recommenders help in the task of completing models *semantically*. Other tasks, like reusing a model or parts of it (Stephan, 2019; Gomes, 2004), repairing an inconsistent model (Marchezan et al., 2023b,a), or improving a model (e.g., via critics (Robbins and Redmiles, 1998; Ali et al., 2013, 2010)), are currently not supported, but left for future work.

2.3 Recommender systems

Recommender systems (RSs) are software tools that suggest items that may be of interest to a particular user according to her preferences, supporting decision making tasks in situations of information overload (Ricci et al., 2022). As a research field, it emerged in the mid-90s with roots in areas such as cognitive science and information retrieval (Adomavicius and Tuzhilin, 2005). Nowadays, they are ubiquitous and key components in many types of applications, in areas such as music (e.g., Spotify), video (e.g., Netflix), streaming platforms (e.g., Twitch), e-commerce sites (e.g., Amazon), and social networks (e.g., Facebook), among many others.

The term *item* is used to indicate what a RS suggests to its users. RSs typically focus on a specific kind of item (e.g., movies), using customised graphical interfaces, and filtering and ranking algorithms to deliver useful and effective recommendations for that kind of item (Ricci et al., 2022). The construction of a RS typically relies on existing data about three types of entities: target users, items, and interactions between users and items (usually unary or numeric ratings) that express personal preferences. These data may come from diverse sources, and their effective exploitation depends on the applied recommendation algorithm and the available computational resources (Ricci et al., 2022).

When it comes to how recommendations are made, RSs are classified into three broad categories: *content-based*, where the items recommended to a user are similar to the ones the user preferred in the past; *collaborative filtering*, where the items recommended to a user are similar to the ones preferred by like-minded users; and *hybrid*, where combinations of the previous two techniques are used to alleviate their drawbacks. Providing useful recommendations for a particular task requires careful selection of the recommendation method. Both content-based and collaborative filtering methods may suffer from *user cold start* situations (i.e., the user has rated few items so that the RS cannot deeply understand the user's preferences (Adomavicius and Tuzhilin, 2005)). While content-based recommenders may suffer from *overspecialisation* – only suggesting items that are too similar to the ones the user already knows – collaborative-filtering is able to suggest novel, diverse or even unexpected (serendipitous) recommendations. However, collaborative filtering may suffer from the *item cold start* problem, in which an item can only be recommended after it has been rated by certain number of users.

The quality of the issued recommendations is critical for the success of a RS (Ricci et al., 2022). The most frequent approaches to evaluate RSs are offline experiments, user studies, and online experiments. *Offline experiments* are conducted on datasets of previous user-item interactions, which are split into training and test sets to build and evaluate the RS, respectively. This type of evaluation is the easiest to conduct, as it assesses the performance of the RS without requiring the intervention of real users in the evaluation process. In contrast, *user studies* allow evaluating a RS online, e.g., via A/B tests that capture the impact of recommendations in real time. This option is more expensive because it requires recruiting test subjects who are asked to perform a series of tasks. Even if offline experiments are useful, real users can provide additional information about the performance of the RS. Finally, *online experiments* allow measuring, among other issues, long-term gain or user retention in order to understand how overall goals are affected. Similar to user studies, online experiments require real users; however, in them, the users are typically unaware of the experiments, which are conducted at large scale on deployed RSs, in settings close to reality.

An important factor to consider when conducting evaluations is the selection of metrics (Ricci et al., 2022). Typical metrics of the ranking quality of the recommendation lists are *precision*, i.e., the likelihood that a suggested item is relevant; *recall*, i.e., the proportion of relevant items in the recommendation lists; *F1*, i.e., the harmonic mean of precision and recall; and *nDCG* (normalised Discounted Cumulative Gain), which considers if the most useful items are in the top positions of the recommendation lists. Additional metrics, such as *MAE* (Mean Absolute Error) and *RMSE* (Root Mean Squared Error), target the accuracy of rating predictions, and are in disuse in the field. Moreover, other complementary metrics can be used, such as *USC* (User Space Coverage), which assesses the percentage of users that the RS can recommend, and *ISC* (Item Space Coverage), which measures the diversity of the recommendations.

Our focus is on using RSs as assistants for creating models (Almonte et al., 2022b). Hence, in our setting, the *items* are model elements to be recommended (e.g., attributes and operations in a class diagram), and the *users* are re-interpreted to be the containers of the items (e.g., classes in a class diagram, which hold the attributes and operations being recommended). To avoid confusion, we use the term *target* – instead of *user* – to denote the model element for which items are recommended.

3 Overview of our Approach

The aim of our approach – called Droid – is to facilitate the construction of RSs for modelling languages of interest, assisting in the creation of models. For this purpose, it provides a textual DSL that allows the RS developer to configure, generate and evaluate candidate RSs. In addition, the approach deploys the selected RS in a REST API, and provides out-of-the-box integration with Eclipse editors. Figure 3 details the steps of the process to define and generate a RS with Droid.

As a first step, the RS developer must collect the necessary data to train and evaluate the RS. These data consist of models conformant to the meta-model of the modelling language the RS is built for. For convenience, our tooling has a facility to collect data from local folders, from the MDEForge model repository (Basciani

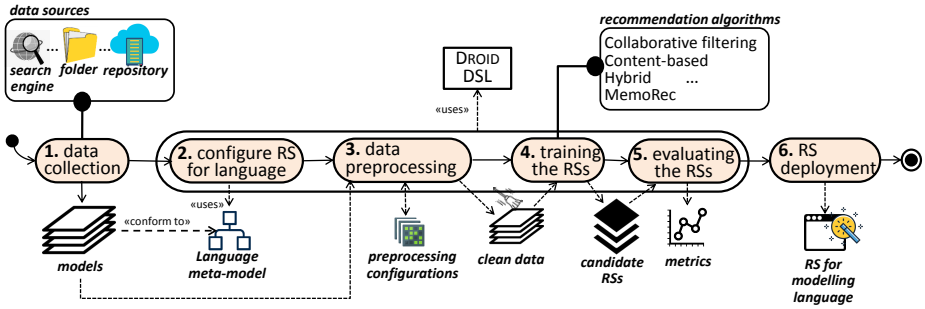


Fig. 3: Process to create a RS with DROID.

et al., 2014), and via the MAR model search engine (Hernández López and Sánchez Cuadrado, 2022). In addition, it is possible to include other data sources via extension points. Some recommendation methods, e.g., rule-based, may not require input data for training. Still, test data is required for the final evaluation stage.

Steps 2 to 5 are driven by the DSL. Specifically, in step 2, the RS developer uses the DSL to specify the items to be recommended (e.g., attributes, operations) and the target of the recommendation (e.g., classes). This effectively configures the RS for the particular modelling language, as the items and target must belong to the language meta-model.

In step 3, the RS developer can use the DSL to preprocess the models collected in step 1 to modify or remove irrelevant or mishaped information. For this purpose, the DSL provides useful preprocessing primitives to, e.g., remove special characters like blank spaces or non-alphabetic characters. The RS developer can specify several candidate preprocessing configurations, and obtain information on the effects of each of them, to apply the most appropriate one.

In step 4, the training of the RS takes place. The RS developer can choose several candidate recommendation methods, and use them with different sets of parameter values. Our implementation currently supports seven methods, and permits incorporating new ones via an extension point. DROID will produce a different RS for each candidate method, trained with the preprocessed models. This training step is omitted for recommendation methods that do not use training data, such as those based on rules.

In step 5, DROID automatically evaluates the resulting RSs against a set of performance metrics. The DSL permits selecting the set of metrics to apply, as well as the evaluation options. The result of the evaluation is a report with the value of the metrics achieved by each candidate recommendation method. This way, the RS developer can easily identify the best performing method.

Finally, in step 6, the RS developer selects the most suitable RS, which becomes automatically deployed as a REST service. This allows the integration of the RS with arbitrary modelling environments. Currently, our tooling provides an out-of-the-box integration of this service with the default tree editors that the Eclipse Modeling Framework (EMF) (Steinberg et al., 2008) generates from Ecore meta-models.

In the next section, we introduce our configuration DSL in more detail.

4 The Droid Domain-Specific Language

This section describes the Droid DSL. The language supports the configuration of RSs, automating all steps in their generation process: configuration of the recommendation targets and items (Subsection 4.1), data preprocessing (Subsection 4.2), training of the candidate RSs (Subsection 4.3), and their evaluation (Subsection 4.4).

We will illustrate the DSL through code fragments corresponding to the running example, each focused on a configuration aspect; however, all fragments together belong to the same file. Figure 4 shows the meta-model of the DSL. It extends the one presented in Almonte et al. (2021) by making the definition of recommendation methods extensible (see Subsection 5.2), and including data preprocessing options. We will refer to this meta-model in the following subsections.

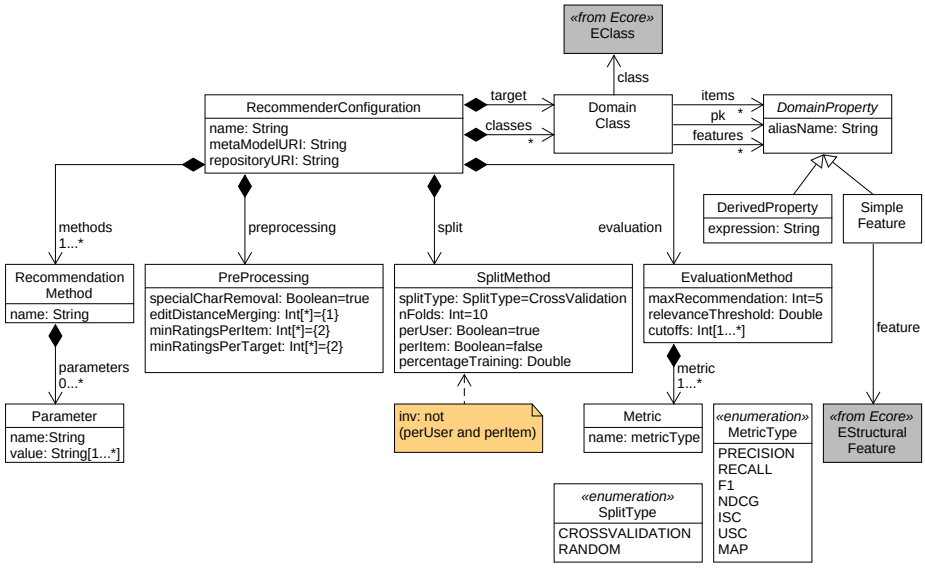


Fig. 4: Meta-model of the Droid DSL (extension of (Almonte et al., 2021)).

4.1 Configuring the RS to the modelling language

Droid supports the generation of RSs for models and meta-models (since a meta-model is also a model). Thus, the configuration of a RS for a (meta-)modelling language requires providing the meta-model of the language, together with a set of instances of this meta-model that will be used for training and evaluation (attributes metaModelURI and repositoryURI in class RecommenderConfiguration of Figure 4). In addition, the configuration must include the target language elements subject to recommendation (class DomainClass), the items to be recommended (class DomainProperty), and their identifiers (references DomainClass.pk and DomainClass.features).

Technology-wise, language meta-models should be described using EMF. Hence, Droid can generate RSs for Ecore meta-modelling (since Ecore has itself a meta-model), and for modelling using any EMF-defined language, either domain-specific or general-purpose like UML.

As an example, Listing 1 shows the configuration of the RS for UML class models. Lines 1–3 declare the name of the RS, the URI of the UML meta-model, and a local folder containing the dataset of models. These models may have been gathered using the data collection facilities of Droid (step 1 in Figure 3). Next, lines 5–10 specify the target of the recommendation (classes) and the modelling items that will be recommended (attributes and operations). Class (line 6) is a meta-class of the UML meta-model (cf. Figure 1a), and for each item to be recommended, the listing specifies an alias ("attributes", "methods") and the property by which such items are accessed from the meta-class. In this case, ownedAttribute and ownedOperation (lines 7–8) are two associations stemming from Class (cf. Figure 1a). Finally, lines 12–22 define the attributes to be used as identifiers of the target class and the items (their name in all cases). In addition to Class (which is the target), these lines declare the identifiers of meta-classes Property and Operation, as they are the target of associations ownedAttribute and ownedOperation in the UML meta-model.

```
1  Recommender: "ModellingRecommender"
2  Metamodel: "http://www.eclipse.org/uml2/5.0.0/UML"
3  Repository: "/ModellingRecommender/instances"
4
5  Target {
6    class Class {
7      item "attributes" : ownedAttribute;
8      item "methods" : ownedOperation;
9    }
10 }
11
12 Identifiers {
13   class Class {
14     pk feature name;
15   }
16   class Property {
17     pk feature name;
18   }
19   class Operation {
20     pk feature name;
21   }
22 }
```

Listing 1: Configuring recommendation targets and recommended item types.

This configuration mechanism enables the specification of the types of objects receiving the recommendation (targets), the types of objects to be recommended (items), and their features. This is enough to configure RSs where the items are objects with attributes, connected to the target of recommendations. In our example, this permits the definition of a wide set of RSs, e.g., recommending classes for packages, or parameters for operations. However, this mechanism is not able to recommend patterns of objects, e.g., UML design patterns.

4.2 Data preprocessing

Data preprocessing aims at improving the quality of the input data (models in our case) and, subsequently, of the RS built from them. It typically involves modifying or deleting irrelevant information from the original dataset (Ricci et al., 2022).

Our DSL offers four preprocessing options (class PreProcessing in Figure 4), which can be combined. The first one (*specialCharRemoval*) allows removing special characters (e.g., numbers, blank spaces, and non-alphabetic characters such as exclamation marks, commas, underscores, or symbols) from the textual information within the models of the dataset. This option can take the values *true* and *false*. The second option (*editDistanceMerging*) specifies the Levenshtein distance under which two strings can be considered equal. This distance is given by the number of deletions, insertions, or substitutions required to transform one string into another (Shahare, 2017) (e.g., "car" and "cat" have Levenshtein distance 1). The third option (*minRatingsPerItem*) removes from the dataset all items that do not appear in a minimum number of targets. Similarly, the last option (*minRatingsPerTarget*) removes all targets lacking a minimum number of items.

Listing 2 defines the preprocessing options for the running example. The DSL supports assigning several values to each option, and the engine will perform all possible combinations. Specifically, the listing enables preprocessing configurations by both removing (*true*) and keeping (*false*) special characters; applying string similarity for distances 2, 3 and 4; deleting the attributes and operations (items) that do not appear in at least 1, 2 or 3 classes (target); and deleting the classes (target) that have less than 1, 2 or 3 attributes or operations (items). Overall, the listing produces 2*3*3*3=54 preprocessing combinations. As we will see in Subsection 5.3, the Droid engine will execute each preprocessing combination, calculating several metrics so that the RS developer can choose the most appropriate one.

```
23 PreProcessing {
24   specialCharRemoval: true, false;
25   editDistanceMerging: 2,3,4;
26   minRatingsPerItem: 1,2,3;
27   minRatingsPerTarget: 1,2,3;
28 }
```

Listing 2: Configuring the data preprocessing options.

The supported preprocessing options were selected based on an analysis of the most common forms of preprocessing that modelling RSs use for models (e.g., merging similar items (Di Rocco et al., 2023)) and general-purpose RSs use for data (e.g., data cleaning, filtering scarcely rated items or targets (Amatriain et al., 2011)). Since the options can be combined and receive ranges of values, the RS designer can quickly specify large amounts of preprocessing configurations. Still, in the future, we aim at supporting specialised preprocessing options by using extension points.

4.3 Training the RS

Training the RS involves two tasks: splitting the data into training and test sets, and building the RS using the training set and one (or several) recommendation methods. DROID allows automating both tasks.

The DSL supports two data splitting techniques: *CrossValidation* and *Random* (class SplitMethod in Figure 4). *CrossValidation* ensures good generalisation and reduces overfitting (Reitermanová, 2010). It is configured with the number of subsets to create, called *k folds*. In this setting, one subset is used for testing and the rest for training, and this process is repeated k times assigning the role of test set to a different subset in each iteration. *Random* splitting is configured with the percentage of elements to be used for training/test, and the sampling is done randomly following a uniform distribution. Lines 30–31 of Listing 3 show the data splitting configuration for the example: 10-fold cross-validation.

```
29 Split {
30   splitType: CrossValidation;
31   nFolds: 10;
32   perUser: true;
33 }
```

Listing 3: Configuring the dataset splitting into training and test sets.

In addition, splits can be performed *per user* or *per item*. The former builds the subsets for each available user, and the latter for each available item. For example, a *perUser* random split with 80% training percentage would use 80% of the preferences of each user (i.e., 80% of the attributes and operations of each class) for training, and the remainder 20% for testing. Line 32 of Listing 3 selects a *perUser* split.

As for the actual training of the RS, the RS developer needs to select the candidate recommendation methods and the values of their parameters (classes RecommendationMethod and Parameter in Figure 4). Then, the DROID engine trains each RS with every specified method and parameter, presenting performance metrics of the resulting RSs so that the developer can choose the most appropriate one.

The DSL currently supports the following seven recommendation methods: item popularity (ItemPop), content-based cosine similarity (CosineCB), user-based collaborative filtering (UBCF), item-based collaborative filtering (IBCF), context-aware collaborative filtering (CACF), user-based collaborative filtering with content-based similarity (CBUB), and item-based collaborative filtering with content-based similarity (CBIB). However, as we will describe in Subsection 5.2, our architecture is extensible, since it permits adding new recommendation methods in an external way, and then the DSL enables their selection by name, and their configuration by providing values to their parameters.

Listing 4 shows the configuration of the recommendation methods for the example, where we have selected all the seven supported methods. The parameter values appear after the method name between parenthesis, and specify different

user/item neighbourhood sizes (i.e., number of most similar users/items with which generating recommendations).

```
34 Methods {
35   ItemPop, CosineCB, CBIB("5","10"), CBUB("5","10"),
36   CACF("5","10"), IBCF("5","10"), UBCF("5","10");
37 }
```

Listing 4: Selecting the candidate recommendation methods to train the RS.

Overall, extensions points make the DSL extensible by enabling the addition of new recommendation methods in an external way. Instead of hard-coding these methods, e.g., as an enumerate or class in the meta-model of Figure 4, the class RecommendationMethod specifies the name of an available method in its attribute name, and sets values to the method parameters. The DSL editor has information of the available methods (these are the externally implemented extension points), so it is able to type-check that the referenced method exists and the parameter values are consistent with the parameter types defined in the extension point (cf. Subsection 5.2).

4.4 Evaluating the RS

The DSL allows choosing the metrics to be used for evaluating the selected RS methods (class EvaluationMethod in Figure 4), and to configure some evaluation options.

Listing 5 shows this configuration for the running example. Line 39 declares the metrics to compute. In this case, the list includes all the metrics currently supported by Droid: *Precision*, *Recall*, *F1* (the harmonic mean of *Precision* and *Recall*), *nDCG* (normalised discounted cumulative gain), *ISC* (item space coverage), *USC* (user space coverage), and *MAP* (mean average precision). Line 40 specifies the number of top items in the ranked recommendation lists with which computing the metrics (*cutoffs*). Then, line 41 sets the maximum number of items that the RS will recommend. Finally, line 42 sets the minimum value of the estimated preference score above which an item is considered a good recommendation (*relevanceThreshold*).

The metrics are computed on the input data reserved for testing by the split technique (Subsection 4.3). When considering a *perUser* split, for each target user (i.e., classes in our example), a metric is computed using the user's items (i.e., attributes and operations in our example) in the test set and the items recommended by a target method, measuring to what extent the two sets of items are close. For example, *precision* measures the percentage of recommended items that belong to the user's test set, and *recall* measures the percentage of the user's test items that the method recommends. This can be measured for the entire recommendation lists, or for a *cutoff* of the top k items of each list. Moreover, in this process, the relevance of a recommended item can be established by a *threshold* score value.

```
Evaluation {
  metrics: Precision, Recall, F1, NDCG, ISC, USC, MAP;
  cutoffs: 1,2,3,4,5;
  maxRecommendations: 5;
  relevanceThreshold: 0.5;
}
```

Listing 5: Choosing and configuring the evaluation metrics.

Overall, the supported metrics are some of the most common ones for evaluating RSs (Gunawardana et al., 2022; Zangerle and Bauer, 2023). However, we plan to add an extension point enabling new metrics to be defined externally.

5 The Droid Framework

Next, we provide a detailed description of the Droid framework, presenting its architecture (Subsection 5.1), extensibility mechanisms (Subsection 5.2), functionalities for the RS developer (Subsection 5.3), and generated recommendation services, including the automatic integration of RSs within the default EMF tree editors (Subsection 5.4).

5.1 Architecture

Figure 5 depicts the architecture of Droid, which has three main components: The *Droid Configurator*, the *Droid Service*, and the *Clients*. Additionally, the framework offers three extension points: DataCollection, DataEncoding, and RecommendationMethod.

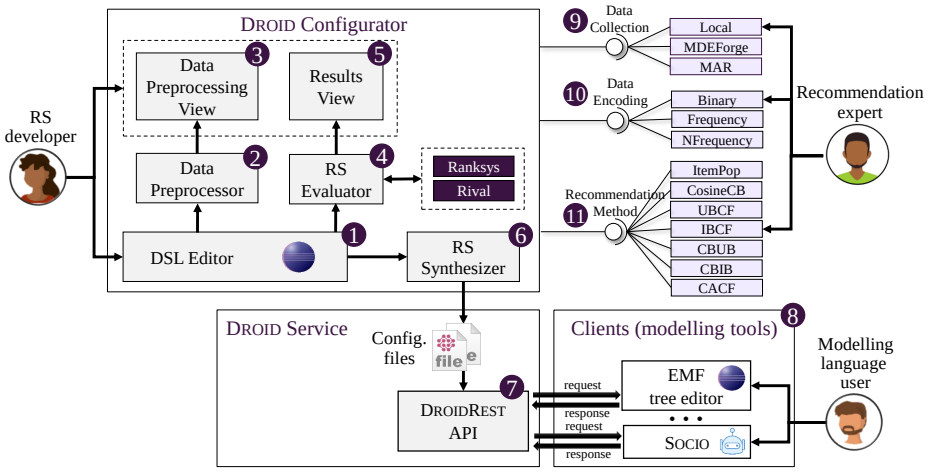


Fig. 5: Architecture of Droid.

The *Droid Configurator* is an Eclipse plugin that permits the RS developer to configure, evaluate and synthesise RSs. It provides an Eclipse textual editor for the DSL presented in Section 4, where the RS developer can configure multiple RSs for a given modelling language (label 1). The configuration specified with the DSL is the input to the *Data Preprocessor* (label 2). The results that each data preprocessing option yields are displayed in a dedicated view, where the RS developer can select the one to apply (label 3). Likewise, the *RS Evaluator* takes the specified RS configuration as input (label 4), and uses the external libraries RankSys (Vargas and Castells, 2011) and RiVal (Said and Bellogín, 2014) to evaluate the selected recommendation methods. The metrics chosen by the RS developer are computed for each RS, and the results are displayed in an Eclipse view (label 5). Subsection 5.3 offers more details about the *Droid Configurator*.

The RS developer can select the preferred RS based on the reported metrics, and the *RS Synthesizer* generates a set of configuration files for the recommendation service (label 6). This service, called *DroidREST* (label 7), is a REST API implementing a generic recommendation service that can be customised for particular modelling languages using the configuration files that the *RS Synthesizer* produces. The service uses a JSON-based model representation that allows clients to request recommendations. *DroidREST* processes these requests and sends the generated recommendations as a response.

In the *Client* side, any modelling tool can use the *Droid Service* to obtain recommendations and make them available to the modelling language users (label 8). Currently, our tooling supports the automatic integration of the resulting RSs within the default tree editor that EMF provides for Ecore-based languages. However, other modelling clients are possible, even outside the Eclipse/EMF ecosystem, as we showed with the Socio modelling chatbot in (Almonte et al., 2021), or the Dandelion low-code platform in (Martínez-Lasaca et al., 2023). Subsection 5.4 provides more details on the *Droid Service* and its clients.

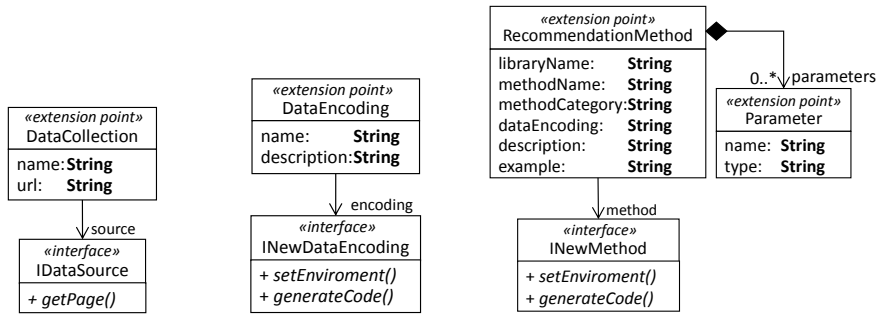
Finally, the framework has three extension points (labels 9–11) that recommendation experts can use to extend Droid, as we explain next.

5.2 Extensibility options

Droid can be extended externally, via three extension points (cf. Figure 5) enabling the incorporation of new data sources (DataCollection), data encodings (DataEncoding) and recommendation methods (RecommendationMethod). Extension points are the mechanism that Eclipse provides to extend the behaviour of a program in an external way. They declare a contract that extensions must conform to. At runtime, an extended program can query for existing extensions implementing an extension point, and invoke them as needed.

Figure 6a depicts the extension point to define new data sources (for collecting models and meta-models) in Droid. Specifically, adding a new data source to Droid requires providing the name of the data source, the URL of the data, and a Java class implementing the interface IDataSource, with details on how to collect the data. Droid currently integrates three data sources, implemented as extensions conforming to this extension point. The first one allows gathering data via browsing local folders. The second one gathers data by accessing the MAR

model search engine (Hernández López and Sánchez Cuadrado, 2022), which returns models satisfying a given set of keywords. The third one retrieves models from the MDEForge (meta-)model repository (Basciani et al., 2014), which supports queries via keywords, a minimum model size, and quality metrics that the models must satisfy.



(a) Data source. (b) Data encoding. (c) Recommendation method.

Fig. 6: Droid extension points.

The second extension point allows the definition of data encodings. These represent the interactions between targets and items in some format – typically as (rating) matrices – which recommendation methods use for training and evaluation. Since each recommendation method requires a specific data encoding, this extension point permits its declaration and implementation. As the class diagram in Figure 6b shows, the extension point requires providing the name of the encoding technique, a description, and an implementation of the Java interface INewDataEncoding performing the actual encoding.

Currently, Droid provides three extensions conforming to the DataEncoding extension point: binary, frequency, and normalised frequency. They build matrices with the targets as rows, and the items as columns. Then, in the binary encoding, each matrix cell is set to 1 if the corresponding target and item are "related" (e.g., a class contains certain attribute), and 0 otherwise; in the frequency encoding, the cells contain a real value that weights the relationship between the target and the item (e.g., the number of classes with same name in different models that contain a given attribute); and in the frequency normalised encoding, the cells contain the frequency value normalised to some range (usually [0,1]).

The last extension point, shown in Figure 6c, allows adding recommendation methods. Implementing this extension point entails providing a Java class (implementing the interface INewMethod) that invokes the associated recommendation method; the name of the used recommendation library (if any); the method name and its category; the name of the data encoding technique used by the method (which must be the name of a DataEncoding extension); a description; an example of use (to be displayed in the editor to help the end-user, like UBCF("5", "10")); and the name and type of the method parameters.

Currently, Droid supports seven recommendation algorithms from two libraries, which have been integrated using the extension point. Specifically, from

MemoRec (Di Rocco et al., 2023), it supports CACF, and from RankSys (Vargas and Castells, 2011), it supports ItemPop, CosineCB, UBCF, IBCF, CBUB and CBIB.

5.3 Droid Configurator

The *Droid Configurator* is an Eclipse plug-in designed to help RS developers build RSs for modelling languages. It is available at `https://droid-dsl.github.io/`.

The configurator includes a wizard – with three steps – for the creation of Droid projects, as shown in Figure 7. The first step requires specifying the name of the new RS, and the modelling (UML, XMI) or meta-modelling (Ecore) technology that the RS will serve. To ease the configuration of the RSs, it also provides an option to automatically generate a default Droid configuration model with typical values. In step 2, the wizard permits selecting one of the available data sources to collect models for training and evaluating the RS. When choosing a data source, the wizard shows a page to configure a query for the selected data source. In particular, the step 3 of the figure displays the query configuration page for the MAR engine.

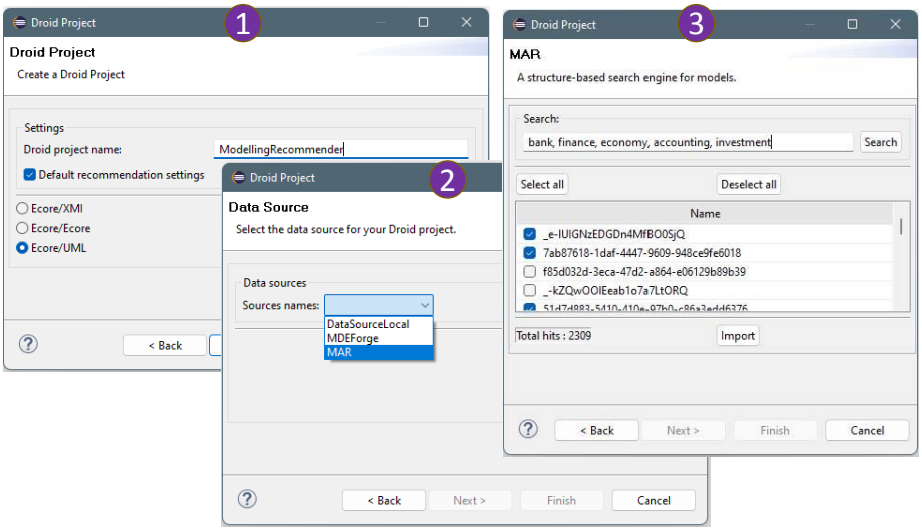


Fig. 7: The three steps of Droid's wizard, with configuration of MAR search in step 3.

Figure 8 shows a screenshot of the *Droid Configurator* environment. The editor (label 1) allows the configuration of RSs through the DSL presented in Section 4. This editor was built using Xtext[1], and features syntax highlighting, autocompletion, and markers for errors and warnings. A pop-up window (label 2) presents autocompletion options by listing the possible items that can be recommended

[1] `http://www.eclipse.org/Xtext/`

for a given target. For instance, for the running example, it shows the properties that the meta-class Class from the UML meta-model declares. The figure shows an excerpt of the UML meta-model (label 3) with arrows to the elements selected in the screenshot.

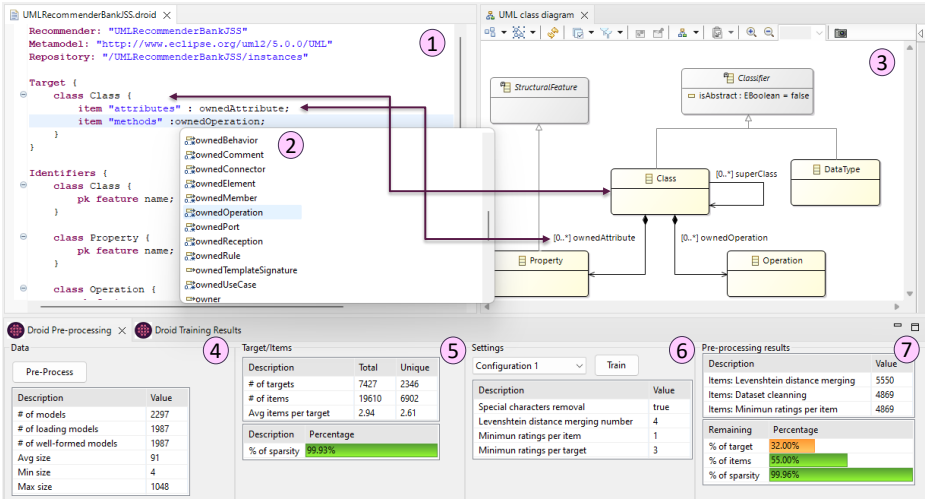
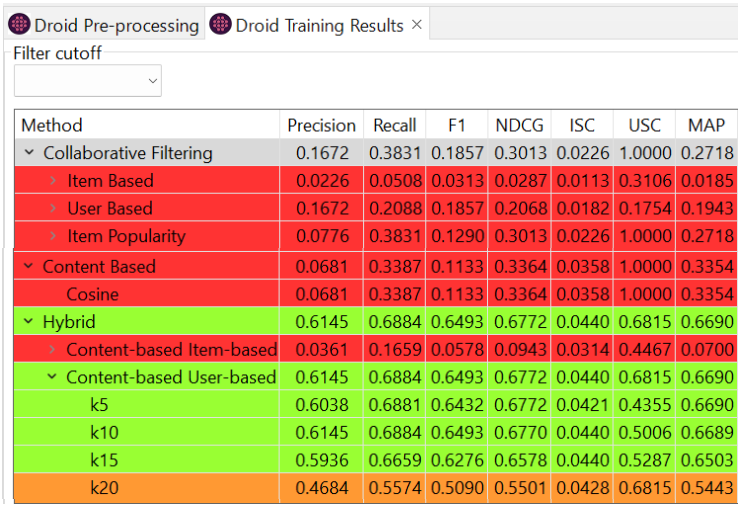


Fig. 8: Screenshot of the *Droid Configurator*.

The *Data Preprocessing* view at the bottom helps in understanding the dataset, and displays information of each preprocessing configuration. The data description section (label 4) provides details about the dataset, namely, the total number of models, loading models (i.e., not broken), and well-formed models; together with the minimum, maximum and average model size (measured as the number of model elements). The target/items section (label 5) shows the number of targets and items (total and unique), the average number of items per target, and the percentage of sparsity (i.e., the percentage of the target-item matrix that is not populated). The settings section (label 6) displays the options of each preprocessing configuration, which concern the removal of special characters, the Levenshtein distance for item merging, and the minimum rating per target and per item (cf. Subsection 4.2). Finally, the panel with label 7 shows the preprocessing results for the currently selected preprocessing configuration in the settings section (*Configuration 1* in the figure). The upper table reports the number of remaining items after applying each preprocessing configuration, and the bottom table displays the percentage of targets and items left after cleaning the data.

To train the candidate RSs, the RS developer needs to choose a preprocessing configuration, and click on the button *Train* (label 6 in Figure 8). This opens the view of Figure 9, which shows metrics for all trained RSs. They are displayed in a drill-down table, where each row corresponds to a recommendation method, and each column to a selected metric. The table displays the method categories (e.g., Collaborative Filtering, Hybrid) according to the extension point details. Within each category, the recommendation methods that were selected using the Droid

DSL are listed, and for each method, there is a block of rows corresponding to the values of each method parameter, if any. Columns 2–8 display the value of the metrics selected in the DSL. As an example, the Hybrid category includes two methods: content-based item-based (CBIB) and content-based user-based (CBUB). These methods have one parameter, named k, which is the neighbourhood size. Consequently, a block is displayed with one row for each specified value of k in the DSL. For CBUB, the parameter k takes the values 5, 10, 15, and 20. To facilitate understanding, the colour of the rows depends on the *F1* value achieved by the methods: green for methods with an *F1* value in the top 20% values, red for methods whose *F1* value is below the median, and orange for the rest.



Method	Precision	Recall	F1	NDCG	ISC	USC	MAP
Collaborative Filtering	0.1672	0.3831	0.1857	0.3013	0.0226	1.0000	0.2718
Item Based	0.0226	0.0508	0.0313	0.0287	0.0113	0.3106	0.0185
User Based	0.1672	0.2088	0.1857	0.2068	0.0182	0.1754	0.1943
Item Popularity	0.0776	0.3831	0.1290	0.3013	0.0226	1.0000	0.2718
Content Based	0.0681	0.3387	0.1133	0.3364	0.0358	1.0000	0.3354
Cosine	0.0681	0.3387	0.1133	0.3364	0.0358	1.0000	0.3354
Hybrid	0.6145	0.6884	0.6493	0.6772	0.0440	0.6815	0.6690
Content-based Item-based	0.0361	0.1659	0.0578	0.0943	0.0314	0.4467	0.0700
Content-based User-based	0.6145	0.6884	0.6493	0.6772	0.0440	0.6815	0.6690
k5	0.6038	0.6881	0.6432	0.6772	0.0421	0.4355	0.6690
k10	0.6145	0.6884	0.6493	0.6770	0.0440	0.5006	0.6689
k15	0.5936	0.6659	0.6276	0.6578	0.0440	0.5287	0.6503
k20	0.4684	0.5574	0.5090	0.5501	0.0428	0.6815	0.5443

Fig. 9: Training results view of the *Droid Configurator*.

5.4 Recommendation service and clients

Upon selecting a RS in the training results view (cf. Figure 9), Droid synthesises a set of configuration files, which are uploaded to a server. These files store the information needed to generate the recommendations, such as the items to recommend for a given target and context. This way, our generic recommendation service *DroidREST* can compute the recommendations based solely on these configuration files. This design facilitates re-deploying the RSs defined with Droid by just uploading new configuration files, without changing the server or the client code.

Moreover, our decision to deploy the RSs as services (instead of their direct deployment within a specific modelling tool) decouples the recommendation computation task from the modelling environment. This way, the same RS can be reused from different modelling tools, even if developed with technologies other than Eclipse – see (Almonte et al., 2021) for an example of integration of a Droid recommender within a modelling chatbot.

Clients (i.e., modelling tools) can solicit recommendations via POST requests to *DroidREST*. The requests should include the name of a deployed recommender, and a JSON file containing the name of the recommendation target and its context (i.e., the items of the target). Then, the service responds with the list of new items recommended for that target. The POST requests accept optional parameters like the maximum number of recommended items to retrieve (*newMaxRec*), the recommendation score threshold for considering an item relevant (*threshold*), and the type of the items (*itemType*).

DroidREST is implemented in Java using Jersey[2] and Tomcat[3]. It has three main classes: *Recommender*, which handles the requests from the clients; *ContextItem*, which parses the received JSON files to extract the recommendation target and its items from the modelling context; and *RecommenderGenerator*, which generates the recommendations for the given target taking its context and the query parameters into account. Since retrieving the recommendations is direct, the service has response times in the order of milliseconds.

While any modelling tool can integrate RSs built with Droid via the REST service, for convenience, we provide an out-of-the-box integration of the service within the default tree editors that the Eclipse Modeling Framework (EMF) generates from Ecore meta-models. By default, these editors permit the creation of instances of a meta-model using a tree view. EMF generates the code of the editors automatically by applying a set of code generation templates to the modelling language's meta-model. Such templates are written in a model-to-text language called Java Emitter Templates (JET)[4] and produce the Java code of the editors. For the integration of the recommenders within such editors, we overwrote those built-in templates so that now, in addition, they generate a *Recommender* pop-up menu on the objects that can be target of the recommendations, and a submenu with the possible item types to be recommended. Implementation-wise, this new menu is an instance of the Java class org.eclipse.jface.action.MenuManager. Clicking on one of these submenus for an item type sends a POST request to *DroidREST* specifying the RS name as a path parameter, and providing a JSON file with the following information: the object target of the recommendation, its items, the objects in its context, and the item type to recommend. *DroidREST* responds with a ranked list of item recommendations in JSON format, which details, for each recommended item, the value of its features (those declared in the Droid configuration file) and a ranking value. Then, the retrieved recommendations are presented in a table inside a Shell window, ordered by their relevance. The user can select recommendations from this table, which are incorporated into the model under construction.

As an illustration, Figure 10 shows a Droid recommender integrated in the tree editor of an object-oriented modelling language similar to UML. The figure shows a class model being edited (label 1). Right-clicking on an object of type Klass (*BankEmployee* in the figure) brings up the pop-up menu *Recommender*, with submenu options for obtaining recommendations of Attributes or Methods for the class (label 2). In the figure, the user has selected the first submenu, so a collection of attributes is recommended (label 3). Internally, the recommendations are retrieved by invoking *DroidREST* passing as parameter a JSON file with the selected object (i.e., *BankEmployee*), its items (none in this case), its context (i.e., object *MyBank*), and the item type to recommend (i.e., *Property*). Then,

[2] https://eclipse-ee4j.github.io/jersey/
[3] https://tomcat.apache.org/
[4] https://projects.eclipse.org/projects/modeling.m2t.jet

the JSON list of recommendations that the service returns is converted into Java objects, which are displayed in a table. As shown in the figure, in addition to the name, each recommended attribute has a rating that indicates the confidence in the recommendation, where higher rating values imply greater confidence that the recommendation is relevant.

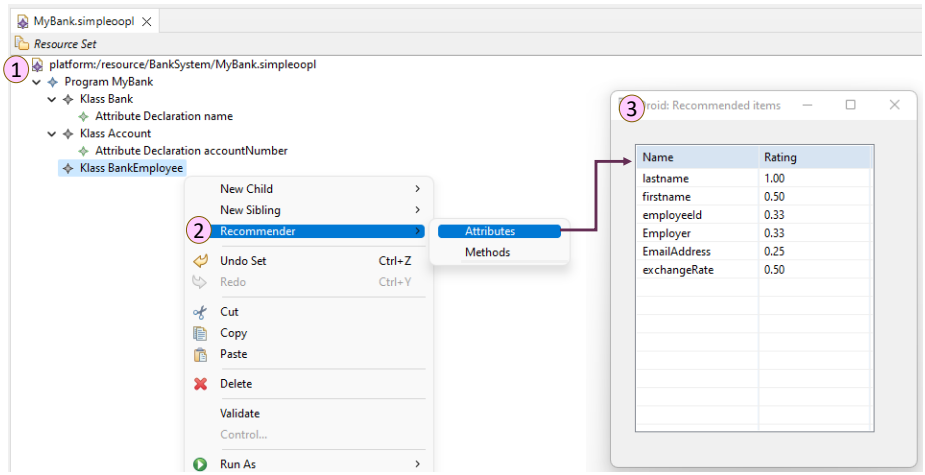


Fig. 10: Out-of-the-box integration of a Droid recommender in a tree modelling editor. (1) Model being edited. (2) Recommender menu displayed upon clicking on a class. (3) Recommendations shown to the user.

Figure 11 shows a screenshot of the example UML RS integrated (manually) within Obeo's UML Designer. For this integration, we created a recommendation layer that, when active, enables to request recommendations for UML classes. In the figure, the user requested recommendations for class *Student*, so a pop-up dialog presents a ranked list of recommendations with their rating. Once the user selects the items of interest, they are added to the class.

6 Evaluation of Droid Recommendations

In this section, we evaluate the recommendations that the RSs built with Droid produce. We pursued two types of experiments. First, in order to assess how good Droid recommendations are with respect to "ground truth" data in existing models, we designed an offline experiment to answer the following RQ:

> **RQ1** *How precise, complete and diverse are the recommendations of Droid recommenders?*

Second, since users of modelling languages may have different perceptions on the quality and usefulness of the issued recommendations, we designed a user study to answer:

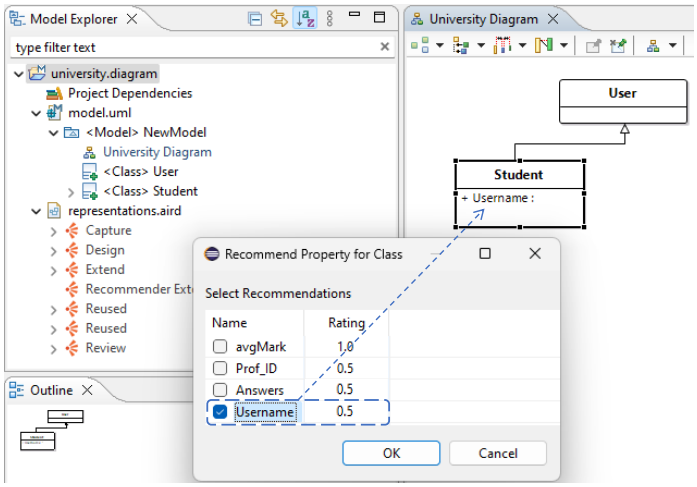


Fig. 11: Integration of the UML RS within a Sirius editor.

RQ2 *How do users perceive the recommendations of* Droid *recommenders?*

Additionally, we wanted to study to what extent the results of the offline evaluation and the user study differ, answering to:

RQ3 *How do the offline experiment results compare to the ones of the user study?*

Figure 12 displays our evaluation process. First, we collected three datasets on three different domains (label 1, Subsection 6.1.1). Then, we used Droid to create a set of candidate RSs (labels 2 and 3, Subsection 6.1.2). To answer RQ1, we compared the resulting RSs by domain using several metrics (label 4, Subsections 6.1.3 and 6.1.4). Subsequently, to answer RQ2, we conducted a user study where 40 participants rated the recommendations produced by three of the recommenders with respect to five criteria (label 5, Subsection 6.2). Finally, the answer to RQ3 comes from the comparison of the results of the offline experiment and the user study (label 6, Subsection 6.2.10). We discuss threats to the validity of these experiments in Subsection 6.3.

The datasets and the raw data of the experiments are available at `https://github.com/Droid-dsl/DroidConfigurator`.

6.1 RQ1: Offline experiment

Next, we describe the setup of the offline experiment (Subsection 6.1.1), the experiment design (Subsection 6.1.2) and the obtained results (Subsection 6.1.3), and answer RQ1 (Subsection 6.1.4).

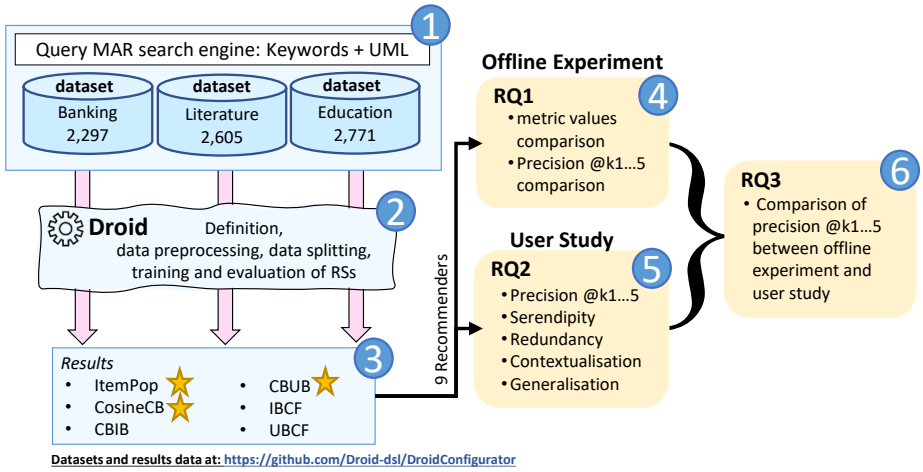


Fig. 12: Evaluation process.

6.1.1 Experiment setup

To understand how good the recommendations of Droid recommenders are, we built several RSs for UML class diagrams in three different domains: *Banking*, *Literature* and *Education*. Just as in the running example, the RSs recommend attributes and operations for classes.

We collected the models used to train and test the RSs from MAR (Hernández López and Sánchez Cuadrado, 2022). Table 1 shows the search keywords we used to retrieve the models for each domain, and some statistics of the resulting datasets: number of models, number of targets (i.e., classes), number of items (i.e., attributes and operations), and average number of items per target. The number of models in the datasets is balanced, ranging from 2,297 models (for *Banking*) to 2,771 (for *Education*). All models conform to the UML 2.0 meta-model, and contain class diagrams. The datasets are available at `https://github.com/Droid-dsl/DroidConfigurator`.

Table 1: Description of the datasets.

Domain	Keywords	Models	Targets	Items	Items/target (avg.)
Banking	bank, finance, economy, accounting, investment	2,297	2,346	6,902	3.15
Literature	bibliography, book, author, journal, magazine	2,605	2,272	7,202	2.94
Education	professor, teacher, student, alumn, school	2,771	2,154	6,789	3.17
	Total:	**7,673**	**6,772**	**20,893**	

These datasets extend those used in the offline experiment conducted in (Almonte et al., 2021) by considering an additional domain (*Banking*) and 3× more

models. This enables us to assess whether the results of the previous preliminary experiment are consistent with those obtained for this larger and more diverse dataset.

6.1.2 Experiment design

We built multiple RSs for each domain, using the configuration options for data preprocessing, data splitting, RS training and RS evaluation in Listings 1–5 of Section 4. More in detail:

- For data preprocessing (cf. Subsection 4.2), we set *specialCharRemoval* to both *true* and *false*; *editDistanceMerging* to 2, 3 and 4; and both *minRatingsPerItem* and *minRatingsPerTarget* to 1, 2 and 3.
- For data splitting (cf. Subsection 4.3), we used 10-fold cross-validation with a *perUser* technique to split the datasets into training and test sets.
- For training (cf. Subsection 4.3), we selected the recommendation methods ItemPop, CosineCB, CBIB, CBUB, IBCF, and UBCF. We excluded CACF from the experiment as it is not applicable to UML models, but just to meta-models. We parameterised all methods but ItemPop and CosineCB with user/item neighbourhood sizes (k) of 5, 10, 15, 20, 25, 50 and 100. As commonly done in the RSs field (Ji et al., 2020), we consider the popularity-based method ItemPop as a baseline to beat, being a non-personalised recommendation approach but capable of providing effective recommendations due to potential popularity biases in the data.
- For evaluating the RSs (cf. Subsection 4.4), we applied the ranking quality metrics *precision* (p), *recall* (r), *F1*, *nDCG* and *MAP*; and the coverage and diversity metrics *USC* and *ISC*. Additionally, we used a relevance threshold of 0.5, cut-off values from 1 to 5, and a maximum number of recommendations of 5.

6.1.3 Experiment results

Table 2 shows the performance achieved by every recommendation method in each domain/dataset (*Banking*, *Education* and *Literature*) as well as the average results across the three domains. The rows correspond to the recommendation methods and domains/datasets, and the columns to the performance metrics. For the sake of brevity, the table just shows three representative recommendation methods: ItemPop (item popularity, a non-personalised baseline), CosineCB (content-based with cosine similarity, a representative of content-based approaches), and CBUB (content user-based hybrid, the best performing collaborative filtering method in this experiment). For CBUB, the table presents the results for neighbourhood size k=5, since it was the best performant. ItemPop and CosineCB do not have parameters.

We can observe that CBUB was the best-performing recommendation method across all domains. By contrast, ItemPop performed the worst across the domains except on *Banking*, where it performed better than CosineCB.

Considering *F1* – the harmonic mean between *precision* and *recall* – the best recommendations were generated for the *Banking* domain, with an *F1* value of 0.643 for CBUB. The other two domains show a significant, but more undersized

Table 2: Performance of the recommendation methods in the offline experiment.

Method	Domain	p	r	F1	MAP	nDCG	USC	ISC
ItemPop	All domains	0.041	0.201	0.069	0.138	0.155	1.000	0.016
CosineCB		0.043	0.216	0.072	0.212	0.213	1.000	0.025
CBUB		0.370	0.515	0.431	0.482	0.492	0.436	0.032
ItemPop	Banking	0.078	0.383	0.129	0.272	0.301	1.000	0.023
CosineCB		0.068	0.339	0.113	0.335	0.336	1.000	0.036
CBUB		0.604	0.688	0.643	0.669	0.677	0.436	0.042
ItemPop	Education	0.019	0.084	0.031	0.066	0.073	1.000	0.008
CosineCB		0.026	0.132	0.044	0.129	0.130	1.000	0.016
CBUB		0.175	0.384	0.241	0.326	0.342	0.423	0.020
ItemPop	Literature	0.028	0.135	0.046	0.076	0.090	1.000	0.016
CosineCB		0.036	0.178	0.059	0.171	0.173	1.000	0.022
CBUB		0.331	0.472	0.389	0.449	0.456	0.451	0.034

performance. Furthermore, the performance results were quite positive over the whole dataset (i.e., considering the three addressed domains together), with a maximum *F1* value of 0.431 for CBUB.

Table 3 presents the precision of the recommendation methods on each domain at cut-off k, $p@k$, for k from 1 to 5. We observe that the smaller the value k, the higher the precision. Likewise, CBUB outperformed ItemPop and CosineCB across all domains.

Table 3: Precision@k of the recommendation methods in the offline experiment.

Method	Domain	p@1	p@2	p@3	p@4	p@5	Avg.
ItemPop	All domains	0.102	0.077	0.061	0.048	0.041	0.066
CosineCB		0.210	0.106	0.071	0.054	0.043	0.097
CBUB		0.464	0.250	0.171	0.130	0.105	0.224
ItemPop	Banking	0.194	0.154	0.127	0.095	0.078	0.129
CosineCB		0.334	0.167	0.112	0.084	0.068	0.153
CBUB		0.663	0.347	0.235	0.176	0.141	0.312
ItemPop	Education	0.058	0.044	0.029	0.022	0.019	0.034
CosineCB		0.126	0.066	0.044	0.033	0.026	0.059
CBUB		0.293	0.176	0.122	0.098	0.078	0.154
ItemPop	Literature	0.053	0.033	0.028	0.025	0.028	0.033
CosineCB		0.169	0.085	0.056	0.044	0.036	0.078
CBUB		0.437	0.229	0.155	0.117	0.095	0.207

6.1.4 Answering RQ1

To answer RQ1 (*How precise, complete and diverse are the recommendations of* Droid *recommenders?*), we analysed the performance of the recommendation methods looking at their *F1* values (cf. Table 2). *F1* is a commonly used metric in the information retrieval and RSs fields. It is the harmonic mean of *precision* and *recall*, and thus provides a compromise between the precision of the recommendations (the proportion of relevant items among the retrieved items) and their recall

(the proportion of relevant items that were retrieved), ensuring that the evaluation considers both the relevance and coverage of the recommendations.

The highest *F1* value was 0.643 for the *Banking* domain, 0.241 for *Education*, 0.389 for *Literature*, and 0.431 for all domains together. In all cases, CBUB achieved the highest *F1* value. These results show that the RSs built with Droid can provide sensible recommendations for the three domains (cf. Table 2). Additionally, we observe that the performance of the RSs varies across domains. Several factors can play a role here depending on the quality of the dataset, like the average number of preferences per target/item, or the rating sparsity, which is the proportion of existing target-item relations.

Our results confirm the conclusions of our preliminary experiment (Almonte et al., 2021), in that collaborative filtering methods outperform content-based ones for the recommendation task of completing classes. However, while UBCF generally performed better in (Almonte et al., 2021), now the hybrid method CBUB is the one with the best performance. This is due to the use of data preprocessing and a bigger dataset, which allows establishing more valuable content-based user similarities within the collaborative filtering heuristic of the method. In particular, the inclusion of a data preprocessing phase, which was absent in (Almonte et al., 2021), permits creating RSs with an order of magnitude higher performance. This way, while preprocessing may lead to higher values of precision-based metrics (*precision*, *recall*, *F1*), it may decrease diversity/coverage metrics (like *ISC* and *USC*) which had higher values in (Almonte et al., 2021). This can be attributed to the fact that preprocessing deleted items that did not appear in a minimum number of targets (2 at least), and targets lacking a minimum number of items (1 item).

6.2 RQ2 and RQ3: User study

In the following, we report on a user study to validate and complement the results of the offline experiment. This study aims at evaluating how users perceive the recommendations issued by Droid recommenders. Next, we present the methodology of the user study (Subsection 6.2.1), the considered assessment metrics (Subsection 6.2.2), the participants (Subsection 6.2.3), the results per assessment metric (Subsections 6.2.4 to 6.2.8), and answer RQ2 and RQ3 (Subsections 6.2.9 and 6.2.10).

6.2.1 Design of the user study

Roughly, the user study involves human participants performing one task, consisting in the assessment of the recommendations generated by the three recommendation methods discussed in the offline experiment: ItemPop, CosineCB, and CBUB. Analogously to the offline experiment, the user study analyses the recommendations for three domains: *Banking*, *Education* and *Literature*.

We conducted the study remotely and asynchronously. The participants received an email invitation that included the URL of the website to participate in the experiment, and a unique identifier. To encourage maximum participation, we granted seven days to complete the experiment. After logging into the website with their identifier, the participants had to evaluate either 3 or 6 cases[5]. Each case

[5] To ensure the evaluation of 45 cases, the last 5 participants only evaluated 3 cases.

presented a class diagram composed of a target class and its modelling context, together with a list of recommended items (a mix of attributes and operations) for the target class. Then, for each recommended item, participants had to state whether they perceived the recommendation as:

- *Correct*: The item is suitable for the target class.
- *Obvious*: The item could have easily been proposed by the participant.
- *Redundant*: The item exists or is similar to an existing one in the diagram.
- *Contextualised*: The item belongs to the domain of the diagram.
- *Generalisable*: The item is also applicable to other classes of the diagram.

As an example, Figure 13 shows one of the cases from the experiment, as presented to the participants. The web page displays: (1) a class diagram with the target class (tagged as «Target») and its context, (2) the recommended attributes and operations for the target class, (3) the five criteria to be evaluated by the participant, (4) the participant's level of confidence in their assessment, and (5) a textbox to specify missed items for the target class not found among the recommendations.

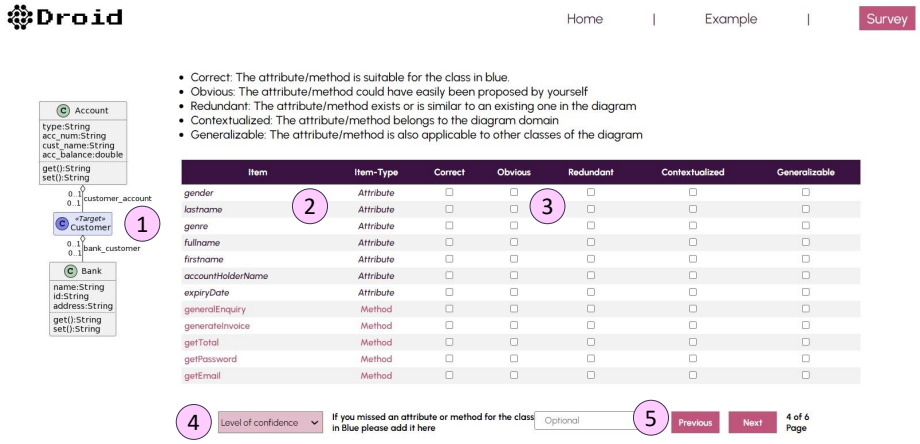


Fig. 13: One of the cases evaluated in the user study.

We assigned the cases to each participant randomly from the three domains, ensuring that each case was evaluated by exactly five participants. The recommendation lists for each target class included the five top items suggested by each recommendation method (ItemPop, CosineCB and CBUB). This means that each target class received a maximum of 3x5=15 recommendations, but there could be fewer, since we eliminated duplicates (i.e., items recommended by more than one method). We selected this size to make the evaluation less burdensome for the participants by recommending a reasonable, but not overwhelming number of options. Altogether, participants evaluated 45 cases, each one of them containing 15 recommendations, which resulted in 3,375 recommendation evaluations. The recommended items were listed in random order, so participants could not know

which method each recommended item came from (not even if the recommendations were automatically generated).

6.2.2 Assessment metrics

To evaluate the participants' perception of the recommendations, we used their assessments about them (i.e., whether they were correct, obvious, redundant, contextualised or generalisable) to compute the following metrics:

- *Precision@k* (for k = 1,...,5), which measures the likelihood that a suggested item is relevant.
- *Serendipity*, which measures how unexpected the correct recommendations are.
- *Redundancy*, which measures to what extent correct recommendations are unnecessary for the user.
- *Contextualisation*, which measures whether a correct recommendation is related to the target's context (in the study, the class diagram of the target class).
- *Generalisation*, which measures whether a correct recommendation is extensible (valuable) to other targets.

Precision is calculated out of the *correct* assessments of participants. It is measured as the percentage of *positively agreed* evaluations about correctness out of the total number of target-item pairs of the study (3,375). As commonly done in user studies with inter-rater agreement (Fleiss et al., 1981), we consider a target-item pair as *positively agreed* if it was marked correct by at least three participants out of the five who evaluated the pair. The computation of the other metrics is analogous. Serendipity considers the negatively agreed *obvious* assessments, i.e., the target-item pairs that the participants marked as not obvious. Similarly, redundancy, contextualisation and generalisation consider the *positively agreed* redundant, contextualised and generalisable assessments, respectively. We do not measure recall, since participants did not have the complete set of correct items for a given target (as stated by human subjects).

It must be stressed that, to the best of our knowledge, this is the first time that metrics serendipity, redundancy, contextualisation, and generalisation are used to evaluate the model completion recommendation task. They, however, have been proposed or used as valuable and complementary metrics for evaluating RSs (Raza and Ding, 2019; Silveira et al., 2019).

6.2.3 Subjects

We recruited the participants by email, inviting a potential set of candidates from the modelling and recommendation areas. Overall, 40 people completed the user study within the given seven day limit (cf. Subsection 6.2.1).

Prior to the experiment, the participants filled out a demographic questionnaire to collect some statistical data. Overall, 80% of the participants were male and 20% female. The majority were between 25 and 34 years old (55%), with the rest between 35 and 44 (27.5%), 45 and 54 (12.5%), and 18 and 24 (5%). Half of the participants were PhD students (50%), while the rest were PhD holders (37.5%), MSc holders (7.5%), and MSc students (5%). Participants were mostly academics

(65%), but also industry employees (22.5%), researchers (7.5%) and students (5%). Note that, in this classification, researchers do not work at a university but in a research center, and industry employees work in a company with no research duty. Most participants (90%) had a high level of English, and all had studied computer-related subjects. A 95% of the participants declared a high level of knowledge on object-oriented programming, and 87.5% on class diagrams. Participants had between 1 to 27 years of experience in software development (10 years in average).

6.2.4 Precision results

In this and the following four subsections, we discuss the results of the user study in terms of the assessment metrics introduced in Subsection 6.2.2. We start by analysing precision. This is the only metric that is common to both the offline experiment and the user study, enabling their comparison. It measures how accurate the recommendations are, being an indicator of the relevance of the recommendations for the users.

In the user study, we compute precision as the percentage of target-item pairs that the participants marked as *correct*. Table 4 shows the precision of the recommendation methods on each domain. Columns three to seven display the precision at k, $p@k$, which is the precision of the top k recommended items, with $k = 1, ..., 5$. The recommendation method with the highest precision for the complete dataset (i.e., considering all domains jointly) is CBUB, with an average precision of 0.373. This is in accordance with the offline experiment, where CBUB was the best performing method. CBUB also achieved the best precision in the *Banking* (0.493) and *Education* (0.373) domains, but not in the *Literature* domain, where CosineCB had a higher precision (0.453).

Table 4: User study: Precision@k by domain.

Method	Domain	p@1	p@2	p@3	p@4	p@5	Avg.
ItemPop	All domains	0.267	0.356	0.267	0.356	0.289	0.307
CosineCB		0.244	0.133	0.467	0.311	0.244	0.280
CBUB		0.444	0.356	0.333	0.400	0.333	0.373
ItemPop	Banking	0.467	0.667	0.333	0.400	0.267	0.427
CosineCB		0.467	0.200	0.533	0.267	0.200	0.333
CBUB		0.600	0.667	0.400	0.400	0.400	0.493
ItemPop	Education	0.067	0.267	0.333	0.267	0.200	0.227
CosineCB		0.067	0.000	0.067	0.133	0.000	0.053
CBUB		0.467	0.133	0.333	0.533	0.400	0.373
ItemPop	Literature	0.267	0.133	0.133	0.400	0.400	0.267
CosineCB		0.200	0.200	0.800	0.533	0.533	0.453
CBUB		0.267	0.267	0.267	0.267	0.200	0.253

More in detail, Table 5 shows the $p@k$ values (for all domains) depending on the confidence level that the participants indicated in the study for each case evaluated. The table only considers those assessments where the participants felt somewhat/fairly/completely confident. In this case, CBUB outperforms CosineCB and ItemPop in all domains. In particular, focusing only on the cases of *completely confident* participants, CBUB achieved an average precision value of 0.543.

Table 5: User study: Precision@k by confidence level for all domains.

Method	Confidence	#Part.	#Evals	p@1	p@2	p@3	p@4	p@5	Avg.
ItemPop	Completely confident	9	225	0.357	0.429	0.286	0.714	0.429	0.443
CosineCB				0.429	0.500	0.643	0.500	0.500	0.514
CBUB				0.429	0.714	0.571	0.429	0.571	0.543
ItemPop	Completely confident, Fairly confident	34	1860	0.341	0.409	0.432	0.500	0.455	0.427
CosineCB				0.386	0.386	0.591	0.477	0.409	0.450
CBUB				0.545	0.455	0.500	0.455	0.568	0.505
ItemPop	Completely confident, Fairly confident, Somewhat confident	40	2835	0.267	0.444	0.311	0.422	0.311	0.351
CosineCB				0.311	0.289	0.533	0.378	0.289	0.360
CBUB				0.467	0.400	0.356	0.422	0.422	0.413

Finally, Table 6 restricts the analysis of precision further by considering only *completely confident* assessments in two scenarios. The first scenario only considers those cases where the target class was contextualised, i.e., the class diagram contained other classes beyond the target class (Table 6a). In the second scenario, in addition to be contextualised, the target class had at least one attribute or operation (Table 6b). The precision of CosineCB and CBUB in these two scenarios increases, achieving $p@k$ values of 0.582 and 0.600. However, as the table shows, these values come from a small number of participants and evaluations, so their statistical support is small. We will use these scenarios later to analyse contextualisation in Subsection 6.2.7.

Table 6: User study: Precision@k of *completely confident* assessments in two scenarios: (a) target class with context; (b) target class with context and items (attributes or operations).

(a) Precision@k contextualised

Method	Confidence	#Part.	#Evals	p@1	p@2	p@3	p@4	p@5	Avg.
ItemPop	Completely confident	8	180	0.364	0.364	0.273	0.636	0.455	0.418
CosineCB				0.455	0.545	0.636	0.636	0.636	0.582
CBUB				0.455	0.636	0.455	0.455	0.636	0.527

(b) Precision@k with contextualised and with items

Method	Confidence	#Part.	#Evals	p@1	p@2	p@3	p@4	p@5	Avg.
ItemPop	Completely confident	6	150	0.444	0.333	0.333	0.667	0.556	0.467
CosineCB				0.444	0.556	0.667	0.667	0.667	0.600
CBUB				0.444	0.667	0.556	0.444	0.667	0.556

6.2.5 Serendipity results

Serendipity is the ability to suggest items that go beyond popular or trending ones (Ricci et al., 2022). It measures the unexpectedness of the recommendations, which the user would not otherwise think of, even though they are relevant to the task at hand. In this sense, serendipitous recommendations are not necessarily novel (fresh) items recently added into the system. By contrast, they are items that, in addition to being unknown to the user, generate certain surprise the first time they are presented.

In the particular recommendation task that we are addressing – class model completion – serendipitous recommendations can be very valuable as they consist of non-evident attributes and operations appropriate for the target class being modelled.

We measure serendipity as the percentage of target-item pairs that the participants marked as *non-obvious*. That is, recommendations marked as obvious are not considered serendipitous, and vice versa.

Table 7 shows the average serendipity values of the recommendation methods per domain. In general, the participants perceived the recommendations as non-obvious, as serendipity values are close to 1. As expected, ItemPop generated slightly more serendipitous recommendations than CosineCB and CBUB when considering the dataset of all domains together, as some recommendations were generic or did not clearly belong to the target domain. If we look at each domain separately, the *Education* domain has the highest serendipity values, and the *Banking* domain the lowest ones. This may be due to the closeness of the *Education* domain to participants, who were mostly academics. We found no significant difference between the serendipity values of the recommendation methods for a same domain.

Table 7: User study: Serendipity@k by domain.

Method	Domain	Evaluations	s@1	s@2	s@3	s@4	s@5	Avg.
ItemPop	All domains	3,375	0.822	0.822	0.956	0.844	0.822	0.853
CosineCB			0.844	0.889	0.600	0.822	0.844	0.800
CBUB			0.756	0.800	0.889	0.800	0.867	0.822
ItemPop	Banking	1,125	0.533	0.533	0.867	0.867	0.867	0.733
CosineCB			0.667	0.867	0.533	0.800	1.000	0.773
CBUB			0.467	0.533	0.733	0.867	0.733	0.667
ItemPop	Education	1,125	1.000	1.000	1.000	0.933	0.867	0.960
CosineCB			0.933	1.000	1.000	1.000	1.000	0.987
CBUB			0.867	0.867	0.933	0.667	0.933	0.853
ItemPop	Literature	1,125	0.933	0.933	1.000	0.733	0.733	0.867
CosineCB			0.933	0.800	0.267	0.667	0.533	0.640
CBUB			0.933	1.000	1.000	0.867	0.933	0.947

Finally, we have calculated the Pearson correlation scores between all pairs of metrics considered in the study (cf. Subsection 6.2.2), and in particular, the correlation between precision and serendipity is 0.545. This value is moderate but relatively high with respect to other pairs of metrics, for which we did not find any significant correlation. This is a positive result since it indicates that the generated recommendations tended to be both accurate and serendipitous. Note that, as reported in Subsection 6.1.1, in our dataset, the average number of items per user, i.e., attributes/methods per class in the UML diagrams, is 3.15, 2.94 and 3.17 for the *Banking*, *Literature* and *Education* domains, respectively. Hence, it is likely that the correctly recommended test items are popular attributes/methods for the classes at hand.

6.2.6 Redundancy results

We propose using the redundancy metric to assess to what extent the recommended items (attributes and operations) are unnecessary, given the current items of the target (class). It is related to diversity, which considers the absence of duplicated or very similar items within a recommendation list. In general, redundant recommendations should be avoided. Specifically for class modelling, designers should not get suggestions of attributes and operations that exist or are too similar to those in a class.

Redundancy is the percentage of recommendations that the participants perceived as *redundant*. Table 8 shows the redundancy values of the recommendation methods by domain. Low values (close to zero) indicate that the recommendations were sufficiently distinct from the attributes and operations of the target class, being consequently useful. We observe no significant differences between the redundancy values of the different methods and domains, except for the *Banking* domain. Redundancy is slightly higher in this domain, meaning that, compared to the other domains, a higher percentage of the recommended items were alike to existing items. This is consistent with the serendipity values that *Banking* obtained (cf. Subsection 6.2.5), as redundant values reflect less serendipity.

Table 8: User study: Redundancy@k by domain.

Method	Domain	Evaluations	r@1	r@2	r@3	r@4	r@5	Avg.
ItemPop	All domains	3,375	0.044	0.156	0.000	0.022	0.022	0.049
CosineCB			0.000	0.022	0.089	0.111	0.022	0.049
CBUB			0.044	0.089	0.067	0.067	0.022	0.058
ItemPop	Banking	1,125	0.133	0.467	0.000	0.000	0.000	0.120
CosineCB			0.000	0.000	0.133	0.267	0.000	0.080
CBUB			0.133	0.267	0.200	0.133	0.000	0.147
ItemPop	Education	1,125	0.000	0.000	0.000	0.000	0.000	0.000
CosineCB			0.000	0.000	0.000	0.000	0.000	0.000
CBUB			0.000	0.000	0.000	0.067	0.067	0.027
ItemPop	Literature	1,125	0.000	0.000	0.000	0.067	0.067	0.027
CosineCB			0.000	0.067	0.133	0.067	0.067	0.067
CBUB			0.000	0.000	0.000	0.000	0.000	0.000

This result could be due to several factors: the input dataset itself, in which classes have been designed with "redundant" attributes and operations; the frequency distribution of items in the *Banking* domain follows a more pronounced heavy tail than the other domains, and thus there is a greater bias towards recommending popular (and therefore redundant) items; the item neighbourhoods in collaborative filtering are smaller, i.e., there are fewer items per target. An exhaustive study should be done to provide a well-argued explanation. Nonetheless, according to the values reported in Table 8, it seems that the last two aspects may apply; the ItemPop and CBUB methods show higher redundancy than CosineCB.

6.2.7 Contextualisation results

Contextualisation is a metric we propose to analyse the relation of the recommended items to the target's context. In the class completion task, the context of a target class refers to the set of other classes (together with their attributes and operations) that surround the target class. In our study, we measure contextualisation as the percentage of recommendations that the participants marked as *contextualised*.

Table 9 shows the average and contextualisation @k values of each recommendation method and domain. As expected, CBUB generated the most contextualised recommendations since it exploits content-based information from related classes in a collaborative filtering fashion. By contrast, CosineCB was the worst-performing method in terms of contextualisation since it only exploits information of the target class to generate recommendations.

Table 9: User study: Contextualisation@k by confidence level in three scenarios: (a) all target classes; (b) target classes with context; (c) target classes with context and items (attributes or operations).

(a) Contextualisation@k by confidence level									
Method	Confidence	#Part.	#Evals	c@1	c@2	c@3	c@4	c@5	Avg.
ItemPop	All	40	3,375	0.289	0.222	0.222	0.200	0.267	0.240
CosineCB				0.156	0.156	0.200	0.244	0.222	0.196
CBUB				0.333	0.200	0.289	0.311	0.267	0.280
ItemPop	Completely confident	9	225	0.429	0.214	0.143	0.357	0.286	0.286
CosineCB				0.143	0.286	0.214	0.214	0.214	0.214
CBUB				0.286	0.357	0.429	0.357	0.500	0.386
(b) Contextualisation@k by confidence level (classes with context)									
Method	Confidence	#Part.	#Evals	c@1	c@2	c@3	c@4	c@5	Avg.
ItemPop	All	40	2,775	0.324	0.243	0.243	0.243	0.297	0.270
CosineCB				0.162	0.162	0.216	0.297	0.243	0.216
CBUB				0.378	0.189	0.324	0.324	0.324	0.308
ItemPop	Completely confident	8	180	0.545	0.182	0.182	0.364	0.273	0.309
CosineCB				0.182	0.364	0.273	0.273	0.273	0.273
CBUB				0.364	0.455	0.455	0.455	0.636	0.473
(c) Contextualisation@k by confidence level (classes with context and items)									
Method	Confidence	#Part.	#Evals	c@1	c@2	c@3	c@4	c@5	Avg.
ItemPop	All	40	2,250	0.300	0.233	0.267	0.200	0.267	0.253
CosineCB				0.133	0.133	0.167	0.200	0.233	0.173
CBUB				0.367	0.200	0.367	0.333	0.333	0.320
ItemPop	Completely confident	6	150	0.444	0.111	0.111	0.222	0.222	0.222
CosineCB				0.111	0.333	0.222	0.222	0.111	0.200
CBUB				0.333	0.444	0.444	0.333	0.556	0.422

More in detail, Table 9a contains the results when considering all cases evaluated by the participants (3,375), where CBUB achieved an average contextualisation value of 0.280. Tables 9b and 9c show contextualisation results when filtering out cases with little context information. In particular, discarding diagrams with only the target class (9b), and discarding diagrams with only the target class or

where the target class had just one attribute/operation (9c). In these scenarios, the contextualisation values increase to 0.308 and 0.320.

For all the above scenarios, as with previous metrics, if we limit the analysis to those assessments with which participants felt completely confident, the contextualisation values of all methods increase, being 0.473 the maximum one, achieved by CBUB. However, when doing so, the computation of such values considers a significantly smaller number of participants and evaluations.

6.2.8 Generalisation results

In the studied class modelling task, generalisation is a metric we propose to measure whether recommendations are applicable (valuable) to classes "related" to the target one. It represents the possibility of using recommended attributes and operations with classes linked to the class that is being defined, e.g., via the subclass relation.

In our study, generalisation was measured as the percentage of recommendations that the participants deemed as *generalisable*. Table 10 shows the generalisation values of the methods, in the upper part considering all assessments, and in the bottom part considering only those assessments for which the participants were completely confident. In average, CosineCB is the method that generates less generalisable recommendations. This can be expected since it is (over)specialised in the target's profile, that is, in the attributes and operations of the target class. ItemPop and CBUB obtained the same generalisation values when taking just completely confident assessments into account, but ItemPop tends to slightly outperform CBUB when considering all assessments. However, checking the most popular items that ItemPop recommends uncovers that they correspond to generic attributes like *name*, *title* and *address*.

Table 10: User study: Generalisation by confidence level.

Method	Confidence	#Part.	#Evals	g@1	g@2	g@3	g@4	g@5	Avg.
ItemPop	All	40	3,375	0.089	0.111	0.044	0.022	0.067	0.067
CosineCB				0.022	0.022	0.044	0.089	0.000	0.036
CBUB				0.089	0.022	0.178	0.044	0.044	0.076
ItemPop	Completely confident	9	225	0.429	0.286	0.357	0.571	0.286	0.386
CosineCB				0.500	0.286	0.214	0.429	0.286	0.343
CBUB				0.429	0.500	0.286	0.286	0.429	0.386

6.2.9 Answering RQ2

To answer RQ2 (*How do users perceive the recommendations of* Droid *recommenders?*), we focus on the metrics used in the user study.

Regarding precision, on individual domains, we observe average values ranging from 0.373 to 0.493 for the hybrid recommendation method CBUB (cf. Table 4). If we limit our attention to those evaluations in which participants stated a complete confidence with their assessments, the global average precision value of CBUB

rises to 0.543 (cf. Table 5). Likewise, in contextualised cases, i.e., in diagrams with more than one class and more than one attribute/operation in the target class, CosineCB achieves precision values as high as 0.600, followed by CBUB with 0.556 (cf. Table 6). Comparing these results with others from related user studies (Cerqueira et al., 2016; Elkamel et al., 2016; Kuschke and Mäder, 2017), we can claim that the recommendations provided by Droid are perceived as highly precise, even considering that we did not use ad hoc recommendation methods for the task at hand, and we did not exploit very large datasets.

Moreover, the participants evaluated the generated recommendations as predominantly serendipitous, with average values ranging from 0.640 to 0.987 (cf. Table 7). In this sense, having both high precision and high serendipity makes Droid recommendations very valuable for users.

When it comes to the perceived redundancy of the recommendations, we observe significant differences between domains (cf. Table 8). Whereas the participants perceived almost no redundancy on the *Education* domain, recommendations on the *Literature* domain were evaluated as slightly redundant, and recommendations on the *Banking* domain seemed the most redundant ones. We argue that these results may depend on certain characteristics of the input datasets, and not on the recommendation method used. Nonetheless, specific recommendation methods to avoid redundancy (e.g., based on diversification techniques) could be researched (Adomavicius and Kwon, 2011).

As expected, the more context the class diagrams have, the more contextualised the recommendations are perceived (cf. Table 9). For instance, average contextualisation values varied from 0.280 to 0.473 for the CBUB method, in non-contextualised and contextualised cases, respectively. The impact of these results in practice should be studied carefully. However, in our opinion, they are promising results that encourage further research on specific context-aware recommendation approaches for the addressed task.

Finally, the participants did not feel the recommendations to be particularly generalisable to other classes in the context (cf. Table 10). Looking only at the evaluations coming from participants completely confident in their assessments, more recommendations were found to be generalisable, although in the case of the ItemPop method, they were also overly generic.

6.2.10 Answering RQ3

To answer RQ3 (*How do the offline experiment results compare to the ones of the user study?*), we compare the precision values of the recommendation methods in the offline experiment (Table 3) and the user study (Table 4). We observe that the precision achieved in both evaluations is high, being the user study results slightly higher. Specifically, the precision of CBUB was particularly similar in both evaluations, respectively achieving average values of 0.224 and 0.373 on the three addressed domains together. Likewise, considering each domain separately, the best performing method in both evaluations was CBUB on the *Banking* domain, with average values of 0.312 and 0.493.

When it comes to the other methods, the precision achieved in the user study is considerably higher for the CosineCB and ItemPop methods, except for CosineCB on the *Education* domain, where the method performed slightly better in the offline experiment (0.053 vs. 0.059 in average).

Generally, higher precision values in the user study could be attributed to the fact that users may evaluate as correct valuable recommendations (e.g., an attribute called account_number for a class Account) that nonetheless are flagged as incorrect by the offline method, which may perform a stricter comparison (e.g., it may flag account_number as incorrect, and account_id as correct). However, we cannot always expect those results, since the quality of recommendations – as perceived by the users – can only be proportional to the quality of the data the RS was trained with. Still, even with low-quality data, an offline evaluation may yield good results (as given by the metrics of choice).

Overall, all the above results confirm that hybrid recommendations – exploiting both content-based and collaborative filtering information – are the most accurate since, as reported in the RSs literature, they help mitigating the particular weaknesses of the two types of approaches (Amatriain et al., 2011; Burke, 2002). Our experiment shows good correlation between the user study and the offline experiment. This is an important result, since most RSs built today are only evaluated with offline experiments, or not at all (Almonte et al., 2022b; Burgueño et al., 2021; Weyssow et al., 2022; Di Rocco et al., 2021, 2022) (see also Subsection 7.3). Even if this may suggest that offline experiments may be a reasonable surrogate for user studies – much more costly to perform – this indication needs to be taken with caution. We argue that user studies – like the one presented here – are advisable to better understand the weak and strong points of a RS, providing an assessment of the real value of recommendations.

6.3 Threats to validity

In this section, we examine the threats to the validity of our evaluations.

External validity. External validity refers to the degree of generalisability of the results of an experiment.

Regarding the offline experiment presented in Subsection 6.1, we used three different domains, and for each one of them, we constructed a specific dataset by searching models containing representative keywords of the domain (cf. Table 1). To increase even further the generality of our results, we could expand the selection of keywords, and incorporate more domains. Moreover, our experiment focused on a particular recommendation task, namely, the recommendation of class attributes and operations. Hence, we cannot claim that our conclusions apply to other different modelling tasks.

In the user study reported in Subsection 6.2, we used the same domains and datasets as in the offline experiment. Therefore, similarly to the offline experiment, the generality of the conclusions of our study could be strengthened if more domains and a bigger dataset were taken into account. Secondly, the participants in the user study performed the evaluation online from their respective work/study locations. We assumed that they had correctly understood their assignment, and did not monitor how long they took to complete it (counting from the moment they started doing it). To assess the participants' engagement, we inspected their responses and found no fixed pattern in their assessments (e.g., always marking the first five recommendations as precise). However, to reduce the likelihood of participants giving random scores, and to make sure that everyone understood

the task at hand, a more controlled user study in person could be conducted. Finally, the user study involved 40 participants, who evaluated 3,375 cases. This represents a substantial amount of data, but further evidence could be obtained with a bigger set of participants (even if the sample size is generally bigger than other user studies evaluating modelling recommendations found in the literature, cf. Subsection 7.3).

Internal validity. Internal validity refers to which extent a causal relationship exists between the conducted experiment and the presented conclusions. In the offline experiment, we attempted to avoid any bias on the data by using a third-party search engine to collect the diagrams of our datasets, and by using a set of model search keywords instead of hand-picking the models. Likewise, we tried to reduce any bias in the results of the user study by choosing the targets for the experiment, the evaluation cases for each participant, and the order of the recommendation lists at random. Moreover, to avoid spurious effects caused by subjective individual assessments, five participants evaluated each recommendation. Fleiss Kappa's coefficients (Fleiss, 1971) for inter-rater agreement show statistically significant ($p < 0.001$), moderate agreement among raters in their assessments of correctness, serendipity and redundancy; and reveal fair agreement among raters regarding the assessments of contextualisation and generalisation[6]. This is reasonable since the latter aspects are more challenging to interpret and evaluate without direct involvement in a class diagram modelling task.

Construct validity. Construct validity is the extent to which an experiment accurately measures the concept it was intended to evaluate. To avoid an inadequate definition of the measured concepts, our two evaluations (offline experiment and user study) applied established metrics in the RSs community. Specifically, the user study used precision, serendipity, redundancy, contextualisation and generalisation. However, the last two of them had to be slightly reformulated for the modelling domain. In addition, the fact that we considered these five indicators to assess the perception of users about the recommendations (rather than a single indicator) mitigates the possibility of construct underrepresentation.

Conclusion validity. Statistical conclusion validity is the degree in which the conclusions are founded on an adequate analysis of the data. On the one hand, the offline experiment employed widely-used software (RankSys and RiVal) to measure the performance of the RSs, thus preventing any spurious measurements. On the other hand, the user study revealed no significant correlation between the metrics, except between precision and serendipity (0.545). In this case, to avoid unreliability in our measures, two authors of the paper carefully revised that they were computed correctly. However, using a small sample size can lead to low statistical power (i.e., finding no correlation when there is one). The somewhat homogeneous background of our participants mitigates this problem. In any case, further studies with more participants could be performed to strengthen our conclusions.

[6] Discarding *"not confident at all"* and *"slightly confident"* assessments, we obtain agreement values of 0.467 for feature correct, 0.522 for obvious, 0.504 for redundant, 0.356 for contextualised and 0.288 for generalisable, with p-values lower than 0.001.

7 Related Work

In this section, we review related research works on RSs developed to assist in software modelling (Subsection 7.1), generative approaches for RSs (Subsection 7.2), and user studies targeting the evaluation of RSs for modelling tasks (Subsection 7.3).

7.1 RSs in modelling tasks

The interest in modelling assistants is increasing in the research community, as shown in our mapping review (Almonte et al., 2022b). There, we characterised, categorised and analysed existing research works on RSs for MDE. In this subsection, we first summarise the approaches analysed in the review, and then, we expand it to include recent proposals in the field.

The most common purposes of RSs in MDE are the completion, finding, repair, reuse and, to a lesser extent, creation of modelling artefacts. As an example, to support model creation and extension, DoMoRe (Agt-Rickauer et al., 2018) exploits the relationships of a knowledge base of domain-specific terms to provide context-sensitive recommendations. To assist in model finding, Cerqueira et al. (2016) propose a content-based approach that finds and recommends sequence diagrams. For model reuse, SimVMA (Stephan, 2019) suggests Simulink models to be imported or cloned, and REBUILDER (Gomes, 2004) finds UML diagrams similar to a given query.

Some approaches have been improved since we performed our survey. For example, PARMOREL (Barriga et al., 2020; Iovino et al., 2020), which applies reinforcement learning to find optimal sequences of model repair actions, has been extended to deal with inter-model inconsistencies (Barriga et al., 2022). Likewise, ReVision (Ohrndorf et al., 2018), an approach that suggests consistency-preserving model editing rules for model repair, has been enhanced to generate repairs for inconsistencies introduced by incomplete editing actions (Ohrndorf et al., 2021). RapMOD (Kuschke and Mäder, 2017; Kuschke et al., 2013), which proposed a ranking approach for the auto-completion of UML class diagrams, has been expanded into an entire auto-completion approach for graphical models supported by a working prototype (Mäder et al., 2021). Finally, the assistant envisioned by Savary-Leblanc (2019), which recommends semantically related terms based on lexical databases like WordNet, has been recently integrated into the Papyrus3 modelling tool (Savary-Leblanc et al., 2021).

Some recent approaches apply novel machine learning techniques to build RSs. Specifically for modelling and meta-modelling, Burgueño et al. (2021) propose a RS for class diagrams based on natural language processing; Weyssow et al. (2022) apply a deep learning model to abstract domain concepts for recommending meta-model concepts; Di Rocco et al. (2021) use graph neural networks (GNNs) to assist modellers in the specification of meta-models and models; and Shilov et al. (2023) also use GNNs, but to assist in enterprise modelling processes. There are also new approaches based on classical recommendation methods, like MemoRec (Di Rocco et al., 2023), which employs context-aware collaborative filtering to provide recommendations for meta-model completion.

Additionally, other recent works propose RSs for editing operations. Specifically, NEMO (Di Rocco et al., 2022) uses an encoder-decoder neural network to aid modellers in executing model editing operations; Ockham (Tinnes et al., 2021) uses an approach based on labelled graphs to learn edit operations from model histories; and Nair et al. (2021) define RSs that improve the modelling experience by informing the modeller about the next modelling action to execute.

As we have discussed, recommenders helping in completing a model typically rely on existing data, which they process to filter and recommend potentially interesting items for the user. However, RSs for other modelling tasks may require different techniques. For example, RSs helping to repair inconsistent models (Marchezan et al., 2023a,b) need to consider the language syntax and semantics, and use specialised techniques, like calculating trees of possible repair actions, and filtering them according to some heuristics. Recommenders helping to improve a model via critics (Robbins and Redmiles, 1998; Ali et al., 2013) are typically based on agents (modelled, e.g., as rules) that continuously analyse the model to detect anti-patterns or potential mistakes. Critics are language-specific, and approaches exist to help defining critics for modelling languages (Ali et al., 2010). Instead, RSs for model completion do not focus on detecting problematic parts of a model, but instead can be invoked to suggest items to include into the model, learned from an existing body of knowledge (e.g., existing models).

Overall, there is a good number of proposals of RSs for modelling. However, the recommendation method is typically fixed, the recommendations cannot be customised, there is no support for data preprocessing, and they do not integrate mechanisms for evaluating the quality of the recommenders, as Droid supports.

7.2 Automated generation of RSs

As we have seen in previous section, most RSs for modelling were developed ad-hoc, by hand and from scratch. Since this requires high expertise and effort (Mussbacher et al., 2020), some studies (Almonte et al., 2022b) identify the need for methods and tools that automate their construction. Following this idea, some recent works propose solutions to reduce the amount of time, effort and expertise required to develop and integrate RSs. The work presented in this paper aims at filling this gap. Next, we compare Droid with other related approaches.

LEV4REC (Di Sipio et al., 2021) is a low-code tool that allows the configuration of RSs for non-modelling environments by means of a feature model. Some aspects that can be customised include the recommendation algorithm or the underlying libraries. From this information, it generates the source code of the RS ready for deployment. However, being a generic tool, LEV4REC does not support the customisation of RSs for modelling languages, or the deployment and integration of the RSs with modelling tools. Likewise, it does not support the exploration of data preprocessing policies or the analysis of the RS performance.

Fellmann et al. (2018) propose a reference model focused on data requirements of RSs for process modelling. The model can be used as a guide for developing new process modelling recommenders or evaluating existing ones. Hence, it is specific to process modelling and does not provide automation or code synthesis.

Hermes (Dyck et al., 2014) permits building Eclipse-based RSs that help in completing models with recommended elements from other models in a repository.

It can be applied to models and meta-models within the EMF ecosystem. Similarly to Droid, it has a plugin-based architecture with extension points for defining new recommendation methods, data repositories and the integration with modelling editors. However, the recommenders are specific for Eclipse modelling tools, and need to be coded. In contrast, Droid recommenders are synthesized automatically from a high-level description, using a DSL, and without coding. Moreover, they can be integrated with any modelling tool via a REST API, and be evaluated to assess their performance.

Rojas and Uribe (2013) propose an MDE framework to create mobile RSs of geographic points of interest. It allows the definition of the structure, behaviour and navigation of the RSs, as well as the customisation of the user preferences and similarity criteria for points of interest. Later on, they proposed a similar solution for trips and tours recommendation (Rojas et al., 2009). In both cases, the domain of the recommendation is fixed.

Espinosa et al. (2013, 2019) present an MDE solution to assist non-expert users in applying data mining. They propose a framework that reuses past experiences of data mining experts to calculate how accurate a new dataset is and recommend the one with the best performance. The framework supports the customisation of the data mining task to perform, the mining algorithm, the evaluation method and the metrics. Although it allows customisation flexibility, recommendations are specific to data mining applications.

7.3 User studies of RSs for modelling

Even though multiple approaches have been proposed to assist in modelling tasks, only a few of them have been evaluated through user studies. Interactions with actual users can yield very valuable knowledge about the RS performance even when offline testing is feasible.

Some reported user studies about recommenders for modelling make use of questionnaires to collect user feedback. This is the case of DSL-maps (Pescador and de Lara, 2016), a recommender of patterns for meta-model construction; Magnet (Abid et al., 2019), an assistant to speed up the learning curve of a modelling tool; and IPSE (Garbe, 2012), an intelligent problem-solving environment for UML class diagrams. Their user studies respectively involved 7, 9, and 14 participants responding to questions after using the proposed tools. Thus, the number of participants is small, and the evaluations measured general, after-use satisfaction.

Other user studies involve performing some modelling tasks with and without the proposed assistants. For instance, Koschmider et al. (Hornung et al., 2009, 2008; Koschmider et al., 2011) evaluated a recommender of process model fragments with 10 participants, who created process models with and without the RS, comparing the time spent, the quality of the result, and the user satisfaction in each case. Similarly, Mora Segura et al. (2023) evaluated the search facilities of the Extremo modelling assistant. In the study, 20 participants built a meta-model with/without assistance. The study measured the productivity of the participants, and the completeness and correctness of the participants' solutions with respect to a reference solution. Sánchez Cuadrado et al. (2018) evaluated a recommender of quick fixes for model transformation within AnATLyzer, by asking 2 experts to repair erroneous transformations without the recommender, and checking if any of

the recommended quick fixes was comparable to their solution. Huh et al. (2009) assessed a recommender of data mappings by asking 4 participants to carry out a set of mapping tasks, and measuring the time spent and the user satisfaction. Nair et al. (2021) conducted a user study to evaluate a recommender of modelling actions, in which 5 participants had to solve several modelling exercises, and their actions were analysed to see if they were among the recommended ones; afterwards, the participants responded to a questionnaire to collect feedback. Finally, Paydar and Kahani (2015a,b) evaluated an approach to facilitate the reuse of functional requirements models. For this purpose, they collected the opinion of 7 experts on the accuracy of the underlying algorithms. Overall, these works were assessed by user studies, but the number of participants was low.

Studies with a higher number of participants evaluate recommendations in different manners. RapMOD (Kuschke and Mäder, 2017), a tool that offers auto-completion of UML models, reports two user studies. In the first one (Kuschke et al., 2013), 16 participants performed a modelling task, which was compared with the recommendations of the approach. In the second one (Mäder et al., 2021), 37 participants performed several modelling tasks with and without the tool. Both studies analysed the performance and quality of the recommendations, and additionally, the latter one collected user feedback. Cerqueira et al. (2016) proposed an approach to find software artefacts in the form of UML sequence diagrams. This was evaluated by presenting 26 participants with instructions to perform searches, and measuring the accuracy and level of satisfaction. Finally, to evaluate an assistant for the design phase of software projects, Elkamel et al. (2016) asked 20 participants to design a UML class diagram with and without the RS. In all these studies, although the number of participants is slightly higher, it is still relatively small, and most lack a clear evaluation design. In contrast, our user study involved more participants, followed a sound methodology, and used assessment metrics both from the RSs community and others specifically devised for class model completion. To our knowledge, this is the first work for modelling recommenders that compares results of an offline experiment with results of a user study.

8 Conclusions and Future Work

This paper has presented Droid, a model-driven approach to create RSs for modelling languages. It offers a DSL to configure every aspect of the RS: the kind of recommended modelling items, the recommendation method, the gathering and preprocessing of training data, and the methodology and metrics to evaluate the created RS. The RS is deployed as a REST service to facilitate its integration with arbitrary modelling tools. Moreover, Droid provides an out-of-the-box integration of the RSs with the default tree editors of EMF-based modelling languages.

We have evaluated the effectiveness of Droid recommenders by means of an offline experiment using over 7,600 UML models across three domains. In addition, we have performed a user study with 40 participants, who perceived the recommendations as highly precise, predominantly serendipitous, slightly or not redundant (depending on the domain), but not particularly generalisable. We also found that the more context the models have, the more contextualised the recommendations are perceived. The precision results of the offline experiment and

the user study are consistent, just slightly better for the user study. Finally, the hybrid recommendation methods were the most accurate in our evaluation.

As we have seen, Droid recommenders can be automatically incorporated into EMF tree editors. Likewise, we have recently released the tool IronMan (Almonte et al., 2023), which facilitates the integration of recommendation services – like those of Droid – into existing graphical modelling editors implemented with the Sirius[7] language workbench. This integration adds an additional graphical layer to the editor, which enables the option to invoke the RS when a shape corresponding to an instance of the target class is selected, displaying a list of recommendations for the user to choose from. In the same line, we would like to provide automated support for the integration of our RSs within textual editors built with technologies like Xtext.

Furthermore, we plan to increase the current library of extensions to integrate additional data sources, data encodings, and recommendation methods (e.g., based on neural networks or built ad-hoc). We would also like to include new extension points in Droid, in particular, to enable the inclusion of specific preprocessing options and user-defined evaluation metrics. For instance, Droid has particular metric implementations of recommendation diversity and coverage, but there are alternative ways to measure these characteristics. In addition, one could define metrics oriented to the task at hand, such as the balance of recommendations in terms of the underlying item types, e.g., classes, methods and attributes in object-oriented modelling.

Droid currently automates the offline evaluation of the created RSs. To complement this facility, we plan to incorporate an automatic generator of the infrastructure needed to conduct user studies of RSs, assisting in the selection of the evaluation cases, the recommendation methods to include in the study, and automating the analysis of the evaluation results. Moreover, we will consider extending Droid to support other modelling tasks, like (sub-)model reuse, model repair, or model optimisation and improvement.

To complement our empirical experiments (i.e., the offline and user studies), we will aim at performing an online experiment in which users are requested to freely perform modelling tasks through a tool, with and without the support of a RS. In this experiment, the utility of recommendations could be defined by alternative aspects (Jannach and Adomavicius, 2016), such as the percentage of recommendations accepted/rejected by users, the time spent to perform the tasks with/without using a recommender, and the quality of the results.

Data Availability Statements

The datasets generated and/or analysed during the current study are available in the following repository: `https://github.com/Droid-dsl/DroidConfigurator`.

The official Droid repository provides documentation, all links to datasets, and any other relevant information: `https://droid-dsl.github.io/`.

[7] `https://projects.eclipse.org/projects/modeling.sirius`

Acknowledgements

This project has received funding from the EU Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 813884, and from the Spanish Ministry of Science (projects TED2021-129381B-C21, PID2021-122270OB-I00, PID2019-108965GB-I00, and RED2022-134647-T).

Conflict of interest

The authors declare that they have no conflict of interest.

References

Abid S, Shamail S, Basit HA, Nadi S (2021) FACER: an API usage-based code-example recommender for opportunistic reuse. Empir Softw Eng 26(5):110

Abid SB, Mahajan V, Lucio L (2019) Machine learning for learnability of MDD tools. In: 31st International Conference on Software Engineering and Knowledge Engineering (SEKE), pp 355–468

Adomavicius G, Kwon Y (2011) Improving aggregate recommendation diversity using ranking-based techniques. IEEE Trans Knowl Data Eng 24(5):896–911

Adomavicius G, Tuzhilin A (2005) Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. IEEE Trans Knowl Data Eng 17(6):734–749

Agt-Rickauer H, Kutsche R, Sack H (2018) DoMoRe - A recommender system for domain modeling. In: 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), SciTePress, pp 71–82

Ali NM, Hosking JG, Grundy JC, Huh J (2010) End-user oriented critic specification for domain-specific visual language tools. In: ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, ACM, pp 297–300

Ali NM, Hosking JG, Grundy J (2013) A taxonomy and mapping of computer-based critiquing tools. IEEE Trans Software Eng 39(11):1494–1520

Almonte L, Cantador I, Guerra E, de Lara J (2020) Towards automating the construction of recommender systems for low-code development platforms. In: MODELS Companion Proceedings, ACM, pp 66:1–66:10

Almonte L, Pérez-Soler S, Guerra E, Cantador I, de Lara J (2021) Automating the synthesis of recommender systems for modelling languages. In: 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE), ACM, pp 22–35

Almonte L, Guerra E, Cantador I, de Lara J (2022a) Building recommenders for modelling languages with Droid. In: 37th IEEE/ACM International Conference on Automated Software Engineering (ASE), ACM, pp 1–4

Almonte L, Guerra E, Cantador I, de Lara J (2022b) Recommender systems in model-driven engineering. Softw Syst Model 21(1):249–280

Almonte L, Garmendia A, Guerra E, de Lara J (2023) Reuse and automated integration of recommenders for modelling languages. In: 16th ACM SIGPLAN

International Conference on Software Language Engineering (SLE), ACM, pp 97–110

Amatriain X, Jaimes A, Oliver N, Pujol JM (2011) Data mining methods for recommender systems. In: Recommender Systems Handbook, 1^{st} Edition, Springer, pp 39–71

Barriga A, Rutle A, Heldal R (2020) Improving model repair through experience sharing. Journal of Object Technology 19(2):13:1–21

Barriga A, Heldal R, Rutle A, Iovino L (2022) PARMOREL: A framework for customizable model repair. Softw Syst Model 21(5):1739–1762

Basciani F, Di Rocco J, Di Ruscio D, Di Salle A, Iovino L, Pierantonio A (2014) MDEForge: An extensible web-based modeling platform. In: CloudMDE@MoDELS, CEUR-WS.org, CEUR Workshop Proceedings, vol 1242, pp 66–75

Brambilla M, Cabot J, Wimmer M (2017) Model-Driven Software Engineering in Practice, Second Edition. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers

Burgueño L, Clarisó R, Gérard S, Li S, Cabot J (2021) An NLP-based architecture for the autocompletion of partial domain models. In: 33rd International Conference on Advanced Information Systems Engineering (CAiSE), Springer, LNCS, vol 12751, pp 91–106

Burke RD (2002) Hybrid recommender systems: Survey and experiments. User Model User Adapt Interact 12(4):331–370

Cerqueira T, Ramalho F, Marinho LB (2016) A content-based approach for recommending UML sequence diagrams. In: 28th International Conference on Software Engineering and Knowledge Engineering (SEKE), pp 644–649

Di Rocco J, Di Ruscio D, Di Sipio C, Nguyen PT, Rubei R (2021) Development of recommendation systems for software engineering: the CROSSMINER experience. Empir Softw Eng 26(4):69

Di Rocco J, Di Sipio C, Di Ruscio D, Nguyen PT (2021) A GNN-based recommender system to assist the specification of metamodels and models. In: 24th International Conference on Model Driven Engineering Languages and Systems (MoDELS), IEEE, pp 70–81

Di Rocco J, Di Sipio C, Nguyen PT, Di Ruscio D, Pierantonio A (2022) Finding with NEMO: A recommender system to forecast the next modeling operations. In: 25th International Conference on Model Driven Engineering Languages and Systems (MoDELS), ACM, pp 154–164

Di Rocco J, Di Ruscio D, Di Sipio C, Nguyen PT, Pierantonio A (2023) MemoRec: A recommender system for assisting modelers in specifying metamodels. Softw Syst Model 22(1):203–223

Di Ruscio D, Kolovos DS, de Lara J, Pierantonio A, Tisi M, Wimmer M (2022) Low-code development and model-driven engineering: Two sides of the same coin? Softw Syst Model 21(2):437–446

Di Sipio C, Di Rocco J, Di Ruscio D, Nguyen PT (2021) A low-code tool supporting the development of recommender systems. In: 15th Conference on Recommender Systems (RecSys), ACM, pp 741–744

Dyck A, Ganser A, Lichter H (2014) A framework for model recommenders - requirements, architecture and tool support. In: MODELSWARD, pp 282–290

Elkamel A, Gzara M, Ben-Abdallah H (2016) An UML class recommender system for software design. In: 13th IEEE/ACS International Conference of Computer

Systems and Applications (AICCSA), IEEE Computer Society, pp 1–8
Espinosa R, García-Saiz D, Zorrilla ME, Zubcoff JJ, Mazón J (2013) Development of a knowledge base for enabling non-expert users to apply data mining algorithms. In: 3rd International Symposium on Data-driven Process Discovery and Analysis, CEUR-WS.org, CEUR Workshop Proceedings, vol 1027, pp 46–61
Espinosa R, García-Saiz D, Zorrilla ME, Zubcoff JJ, Mazón J (2019) S3Mining: A model-driven engineering approach for supporting novice data miners in selecting suitable classifiers. Computer Standards and Interfaces 65:143–158
Fellmann M, Metzger D, Jannaber S, Zarvic N, Thomas O (2018) Process modeling recommender systems - A generic data model and its application to a smart glasses-based modeling environment. Bus Inf Syst Eng 60(1):21–38
Fleiss JL (1971) Measuring nominal scale agreement among many raters. Psychological Bulletin 76(5):378–382
Fleiss JL, Levin B, Paik MC, et al. (1981) The measurement of interrater agreement. Statistical Methods for Rates and Proportions pp 212–236
Garbe H (2012) Intelligent assistance in a problem solving environment for UML class diagrams by combining a generative system with constraints. In: eLearning, IADIS
Gomes P (2004) Software design retrieval using bayesian networks and wordnet. In: 7th European Conf. on Advances in Case-Based Reasoning (ECCBR), Springer, LNCS, vol 3155, pp 184–197
Gunawardana A, Shani G, Yogev S (2022) Evaluating recommender systems. In: Recommender Systems Handbook, 3^{3d} Edition, Springer US, pp 547–601
Hernández López JA, Sánchez Cuadrado J (2022) An efficient and scalable search engine for models. Softw Syst Model 21(5):1715–1737, see also `http://mar-search.org/`
Hornung T, Koschmider A, Lausen G (2008) Recommendation based process modeling support: Method and user experience. In: 27th International Conference on Conceptual Modeling (ER), Springer, LNCS, vol 5231, pp 265–278
Hornung T, Koschmider A, Oberweis A (2009) A recommender system for business process models. In: 17th Annual Workshop on Information Technologies & Systems (WITS)
Huh J, Grundy JC, Hosking JG, Li KN, Amor R (2009) Integrated data mapping for a software meta-tool. In: 20th Australian Software Engineering Conference (ASWEC), IEEE Computer Society, pp 111–120
Iovino L, Barriga A, Rutle A, Heldal R (2020) Model repair with quality-based reinforcement learning. Journal of Object Technology 19(2):17:1–21
Jannach D, Adomavicius G (2016) Recommendations with a purpose. In: 10th ACM Conference on Recommender Systems (RecSys), ACM, pp 7–10
Ji Y, Sun A, Zhang J, Li C (2020) A re-visit of the popularity baseline in recommender systems. In: 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, pp 1749–1752
Koschmider A, Hornung T, Oberweis A (2011) Recommendation-based editor for business process modeling. Data Knowl Eng 70(6):483–503
Kuschke T, Mäder P (2017) RapMOD - in situ auto-completion for graphical models: poster. In: 39th International Conference on Software Engineering (ICSE), Companion Volume, IEEE Computer Society, pp 303–304
Kuschke T, Mäder P, Rempel P (2013) Recommending auto-completions for software modeling activities. In: 16th International Conference on Model-Driven

Engineering Languages and Systems (MoDELS), Springer, LNCS, vol 8107, pp 170–186

Liu F, Li G, Fu Z, Lu S, Hao Y, Jin Z (2022) Learning to recommend method names with global context. In: 44th International Conference on Software Engineering (ICSE), ACM, ICSE '22, pp 1294–1306

Marchezan L, Assunção WKG, Michelon GK, Egyed A (2023a) Do developers benefit from recommendations when repairing inconsistent design models? a controlled experiment. In: 27th International Conference on Evaluation and Assessment in Software Engineering (EASE), ACM, pp 131–140

Marchezan L, Kretschmer R, Assunção WKG, Reder A, Egyed A (2023b) Generating repairs for inconsistent models. Softw Syst Model 22(1):297–329

Martínez-Lasaca F, Díez P, Guerra E, de Lara J (2023) Dandelion: A scalable, cloud-based graphical language workbench for industrial low-code development. J Comput Lang 76:101217

Matikainen P, Furlong PM, Sukthankar R, Hebert M (2013) Multi-armed recommendation bandits for selecting state machine policies for robotic systems. In: 2013 IEEE International Conference on Robotics and Automation (ICRA), IEEE, pp 4545–4551

Mazanek S, Minas M (2009) Business process models as a showcase for syntax-based assistance in diagram editors. In: 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Springer, LNCS, vol 5795, pp 322–336

Moha N, Sen S, Faucher C, Barais O, Jézéquel J (2010) Evaluation of Kermeta for solving graph-based problems. Int J Softw Tools Technol Transf 12(3-4):273–285

Mora Segura A, de Lara J, Wimmer M (2023) Modelling assistants based on information reuse: A user evaluation for language engineering. Softw Syst Model In press

Mussbacher G, Combemale B, Kienzle J, Abrahão S, Ali H, Bencomo N, Búr M, Burgueño L, Engels G, Jeanjean P, Jézéquel J, Kühn T, Mosser S, Sahraoui HA, Syriani E, Varró D, Weyssow M (2020) Opportunities in intelligent modeling assistance. Softw Syst Model 19(5):1045–1053

Mäder P, Kuschke T, Janke M (2021) Reactive auto-completion of modeling activities. IEEE Trans Software Eng 47(7):1431–1451

Nair A, Ning X, Hill JH (2021) Using recommender systems to improve proactive modeling. Softw Syst Model 20(4):1159–1181

Neubauer P, Bill R, Mayerhofer T, Wimmer M (2017) Automated generation of consistency-achieving model editors. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE Computer Society, pp 127–137

Ohrndorf M, Pietsch C, Kelter U, Kehrer T (2018) ReVision: a tool for history-based model repair recommendations. In: 40th International Conference on Software Engineering (ICSE), Companion Proceeedings, ACM, pp 105–108

Ohrndorf M, Pietsch C, Kelter U, Grunske L, Kehrer T (2021) History-based model repair recommendations. ACM Trans Softw Eng Methodol 30(2):15:1–15:46

Oliveira MCD, Freitas D, Bonifácio R, Pinto G, Lo D (2019) Finding needles in a haystack: Leveraging co-change dependencies to recommend refactorings. Journal of Systems and Software 158

Paydar S, Kahani M (2015a) A semantic web enabled approach to reuse functional requirements models in web engineering. Autom Softw Eng 22(2):241–288

Paydar S, Kahani M (2015b) A semi-automated approach to adapt activity diagrams for new use cases. Inf Softw Technol 57:543–570

Pescador A, de Lara J (2016) DSL-maps: from requirements to design of domain-specific languages. In: 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), ACM, pp 438–443

Raza S, Ding C (2019) Progress in context-aware recommender systems—an overview. Computer Science Review 31:84–97

Reitermanová Z (2010) Data splitting. In: WDS'10 Proceedings of Contributed Papers, Part I, pp 31–36

Ricci F, Rokach L, Shapira B (2022) Recommender Systems Handbook, 3rd edn. Springer US

Robbins JE, Redmiles DF (1998) Software architecture critics in the argo design environment. Knowl Based Syst 11(1):47–60

Robillard MP, Walker RJ, Zimmermann T (2010) Recommendation systems for software engineering. IEEE Software 27(4):80–86

Rojas G, Uribe C (2013) A conceptual framework to develop mobile recommender systems of points of interest. In: SCCC, IEEE Computer Society, pp 16–20

Rojas G, Domínguez F, Salvatori S (2009) Recommender systems on the web: A model-driven approach. In: 10th International Conference on E-Commerce and Web Technologies (EC-Web), Springer, LNCS, vol 5692, pp 252–263

Said A, Bellogín A (2014) Rival: a toolkit to foster reproducibility in recommender system evaluation. In: 8th ACM Conference on Recommender Systems (RecSys), ACM, pp 371–372, see also `https://github.com/recommenders/rival`

Sánchez Cuadrado J, Guerra E, de Lara J (2018) Quick fixing ATL transformations with speculative analysis. Softw Syst Model 17(3):779–813

Savary-Leblanc M (2019) Improving MBSE tools UX with ai-empowered software assistants. In: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS), Companion Volume, IEEE, pp 648–652

Savary-Leblanc M, Le-Pallec X, Gérard S (2021) A modeling assistant for cognifying MBSE tools. In: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp 630–634

Shahare FF (2017) Sentiment analysis for the news data based on the social media. In: 2017 International Conference on Intelligent Computing and Control Systems (ICICCS), pp 1365–1370

Shilov N, Othman W, Fellmann M, Sandkuhl K (2023) Machine learning for enterprise modeling assistance: An investigation of the potential and proof of concept. Softw Syst Model 22(2):619–646

Silveira T, Zhang M, Lin X, Liu Y, Ma S (2019) How good your recommender system is? A survey on evaluations in recommendation. International Journal of Machine Learning and Cybernetics 10:813–831

Steinberg D, Budinsky F, Paternostro M, Merks E (2008) EMF: Eclipse Modeling Framework, 2^{nd} Edition. Addison-Wesley Professional, see also `http://www.eclipse.org/modeling/emf/`

Stephan M (2019) Towards a cognizant virtual software modeling assistant using model clones. In: 41st International Conference on Software Engineering: New Ideas and Emerging Results (NIER@ICSE), IEEE, pp 21–24

Tinnes C, Kehrer T, Joblin M, Hohenstein U, Biesdorf A, Apel S (2021) Learning domain-specific edit operations from model repositories with frequent subgraph

mining. In: 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 930–942

Tuarob S, Assavakamhaenghan N, Tanaphantaruk W, Suwanworaboon P, Hassan S, Choetkiertikul M (2021) Automatic team recommendation for collaborative software development. Empir Softw Eng 26(4):64

UML 251 OMG specification (2017) http://www.omg.org/spec/UML/2.5.1/

Vargas S, Castells P (2011) Rank and relevance in novelty and diversity metrics for recommender systems. In: 5th ACM Conference on Recommender Systems (RecSys), ACM, pp 109–116, see also http://ranksys.github.io/

Weyssow M, Sahraoui H, Syriani E (2022) Recommending metamodel concepts during modeling activities with pre-trained language models. Softw Syst Model 21(3):1071–1089

Zangerle E, Bauer C (2023) Evaluating recommender systems: Survey and framework. ACM Comput Surv 55(8):170:1–170:38