Building recommender systems for modelling languages with Droid

Lissette Almonte, Esther Guerra, Iván Cantador, Juan de Lara
Universidad Autónoma de Madrid
Madrid, Spain

ABSTRACT

Recommender systems (RSs) are increasingly being used to help in all sorts of software engineering tasks, including modelling. However, building a RS for a modelling notation is costly. This is especially detrimental for development paradigms that rely on domain-specific languages (DSLs), like model-driven engineering and lowcode approaches.

To alleviate this problem, we propose a DSL called Droid that facilitates the configuration and creation of RSs for particular modelling notations. Its tooling provides automation for all phases in the development of a RS: data preprocessing, system configuration for the modelling language, evaluation and selection of the best recommendation algorithm, and deployment of the RS into a modelling tool. A video of the tool is available at https://www.youtube.com/watch?v=VHiObfKUhS0.

CCS CONCEPTS

• **Software and its engineering → Designing software**; • **Information systems → Information retrieval**.

KEYWORDS

Modelling Languages, Model-Driven Engineering, Domain-Specific Languages, Recommender Systems

ACM Reference Format:
Lissette Almonte, Esther Guerra, Iván Cantador, Juan de Lara. 2022. Building recommender systems for modelling languages with Droid. In *Proceedings of 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022).* ACM, New York, NY, USA, 4 pages. https://doi.org/x.x

1 INTRODUCTION

Recommender systems (RSs) [1] are information filtering systems that help users in choosing among a potentially large set of items. They aim at predicting the preferences of the user to offer a personalised list of items that she may find of interest. RSs are ubiquitous nowadays, being integrated in all sorts of commercial and leisure platforms, and providing suggestions on a variety of items for different applications, such as music (e.g., Spotify, Pandora) and video (e.g., Netflix, YouTube) to consume in streaming platforms, or products to buy in e-commerce sites (e.g., Amazon, eBay).

ASE 2022, October 10–14, 2022, Ann Arbor, Michigan

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/x.x

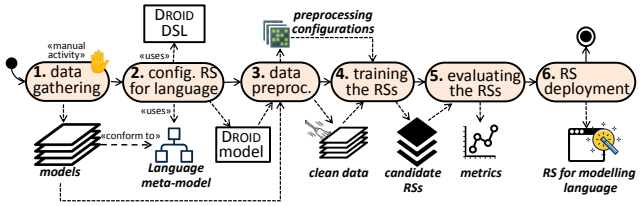

Figure 1: Process of generating a RS with Droid.

Likewise, RSs are increasingly being used as software engineering assistants [15] to support the phases of the software life cycle. For development, we can find RSs that suggest code refactorings [7], IDE commands [10] or function invocations [13], to name a few. For analysis and design, we are witnessing incipient efforts to support modelling via RSs [3], with proposals of recommenders for class diagrams [12], sequence diagrams [6] or Simulink models [17].

Building a RS by hand is costly, as it requires selecting and configuring the most convenient recommendation algorithm for the problem at hand – which demands specialised knowledge – and then integrating it into a tool. This is especially challenging when building a RS for modelling, as modelling languages can be of different nature and target diverse domains. This fact hinders the development of RSs for software paradigms that rely on domain-specific languages (DSLs), like model-driven engineering and lowcode approaches. Hence, a challenge in the area is to reduce the development effort and expertise needed to create a RS for an arbitrary DSL.

To tackle this challenge, we propose a tool called Droid. With the goal of facilitating the creation of RSs for modelling languages, Droid provides: (i) a DSL to configure the type of items that the RS will suggest (e.g., attributes and methods for class diagrams, activities for process models); (ii) an engine that automates data preprocessing and the evaluation of the candidate recommendation algorithms against configurable metrics; (iii) a RS generator that deploys the selected recommendation algorithm as a service, which heterogeneous modelling clients can integrate; and (iv) out-of-the-box integrations with Eclipse modelling tools.

The users of Droid are RS developers, but more generally DSL developers who are not required to have deep knowledge of RS technologies. The RSs generated with Droid can be integrated into heterogeneous modelling clients, which then modellers can use.

In the next section, we showcase the most salient features of Droid using as a running example the creation of a RS of attributes, methods and superclasses for classes in a class diagram.

2 THE DROID TOOL

Droid (https://droid-dsl.github.io/) is an Eclipse plugin automating the construction of RSs for modelling languages, as Fig. 1 depicts.

The process starts by gathering the data to train and evaluate the RS. Since Droid targets RSs for modelling languages, these data consist of models conforming to the language meta-model. This step is manual, while the following ones are automated.

In a second step, the RS developer uses the Droid DSL to configure the RS to the modelling language, choosing the targets[1] of the recommendations (e.g., classes in a class diagram), and the items to be recommended (e.g., class attributes). Sec. 2.1 explains the usage of the DSL for this purpose.

In step 3, the developer can configure criteria to preprocess the input models (e.g., removing special characters) using the DSL. This yields a set of preprocessing configurations, which Droid evaluates to provide the developer with an informed selection of the most appropriate one (cf. Sec. 2.2). Then, Droid uses the cleaned data to train a set of candidate RSs according to an initial choice of recommendation algorithms (step 4), and evaluates such RSs over configurable metrics (step 5). Sec. 2.3 details these two steps.

Finally, based on the metric values, the developer selects the most suitable RS, which is automatically deployed as a REST service (cf. Sec. 2.4). Droid also provides an out-of-the-box integration of this recommendation service in the EMF tree editor (cf. Sec. 2.5).

2.1 Configuring a RS for a modelling language

Droid provides a DSL to configure the RS for a modelling language and to select the data preprocessing actions, recommendation algorithms and evaluation metrics. This subsection focuses on the configuration of the language elements subject to recommendation, and the next two subsections explain the other elements.

The listing to the right shows the fragment of a Droid program to configure a RS for UML class diagrams. Lines 1–3 declare the RS name, the meta-model of the modelling language the RS is developed for, and the location path of the input models. Lines 5–10 specify the modelling language items that will be recommended. In this case, the RS will suggest attributes, methods and superclasses for a target Class. Class (line 6) is a meta-class of the UML2 meta-model. ownedAttribute, ownedOperation and superClass (lines 7–9) are references of the meta-class Class, from where the items are obtained. Finally, lines 12–21 declare the attributes used as identifiers of the target class and recommended items.

```
1  Recommender: "EducationRecommender"
2  Metamodel: "http://www.eclipse.org/uml2"
3  Repository: "/EduRecommender/models"
4
5  Target {
6     class Class {
7        item "attributes" : ownedAttribute;
8        item "methods" : ownedOperation;
9        item "super classes" : superClass; }
10 }
11
12 Identifiers {
13    class Class { pk feature name; }
14    class Type { pk feature name; }
15    class Property {
16       pk feature name;
17       pk feature type; }
18    class Operation {
19       pk feature name;
20       pk feature type; }
21 }
```

Droid offers a dedicated Eclipse editor for the DSL, shown in Fig. 2. The editor (label 1) features code auto-completion and helps selecting proper meta-model classes and features to configure the RS (pop-up menu with label 2, and meta-model with label 3).

2.2 Data preprocessing

Data preprocessing is important in RSs to modify or delete irrelevant or misshaped information from the input training and test data [14]. In our context, the data are models conformant to the modelling language meta-model. Hence, our DSL offers four preprocessing options which can take one or more values, and Droid tries each combination of them.

The listing to the right configures the data preprocessing for the example RS. The option specialCharRemoval declares if special characters (e.g., numbers, symbols) must be removed. Line 2 enables preprocessing configurations both removing characters (true) and keeping them (false). Line 3 specifies the edit Levenshtein distances under which two strings are considered the same. The distance is given by the number of single-character edits required to transform one string into the other [11]. In line 4, minRatingsPerItem filters out the items that do not appear in a minimum number of targets. In this example, it excludes the attributes, methods and superclasses appearing in less than 1, 2 or 3 classes. Finally, in line 5, minRatingsPerTarget drops the targets lacking a minimum number of items (in this case, classes with less than 1, 2 or 3 attributes, methods or superclasses). Overall, this specification generates 2*3*3*3=54 preprocessing combinations.

```
1 PreProcessing {
2    specialCharRemoval: true,false;
3    editDistanceMerging: 2,3,4;
4    minRatingsPerItem: 1,2,3;
5    minRatingsPerTarget: 1,2,3;
6 }
```

The view at the bottom of Fig. 2 shows the results of each preprocessing combination. Sections *Data* (label 4) and *Target/Items* (label 5) give information of the input models provided for training and evaluation: initial number of models, how many of them can be loaded, how many are well-formed models, average/minimum/maximum model size based on the number of model elements, average number of targets and items (as configured in the DSL) both total and unique (i.e., without repetitions), and data sparsity. Section *Settings* (label 6) includes a combo box with all preprocessing combinations ordered by relevance. Selecting one displays its results in section *Pre-processing results* (label 7): number of unique items remaining after each preprocessing action, percentage of targets and items remaining after the preprocessing, and achieved target-item data sparsity. Based on this information, the RS developer selects the desired preprocessing configuration and proceeds with the RS training.

2.3 Training and evaluation

The last aspects to configure with the DSL are the candidate recommendation algorithms and the evaluation method and metrics. To simplify this configuration, Droid generates default options, which the developer can refine.

The listing to the right shows an example. Lines 2–6 select the recommendation algorithms of interest organised by type – collaborative filtering (CF), content-based (CB), hybrid – along with candidate parameters. For example, line 3 selects the item-based CF (IBCF) strategy with two candidate neighbourhood sizes: 10 and 20. Droid supports six recommendation algorithms: item popularity, item-based CF, user-based CF, cosine-based CB, and item-based

```
1  Recommendations {
2     Methods {
3        collaborativeFiltering: ItemPop,
                IBCF(10,20), UBCF(10,20);
4        contentBased: CosineCB;
5        hybrid: CBIB(10,20), CBUB(10,20);
6     }
7     Split {
8        splitType: CrossValidation;
9        nFolds: 10;
10       perUser: true;
11    }
12    Evaluation {
13       metrics: Precision, Recall, F1,
                NDCG, ISC, USC, MAP;
14       cutoffs: 1,5,10,15,20;
15       maxRecommendations: 50;
16       relevanceThreshold: 0.5;
17    }
18 }
```

[1] Recommendation *targets* are the *users* commonly referred in the RSs literature.

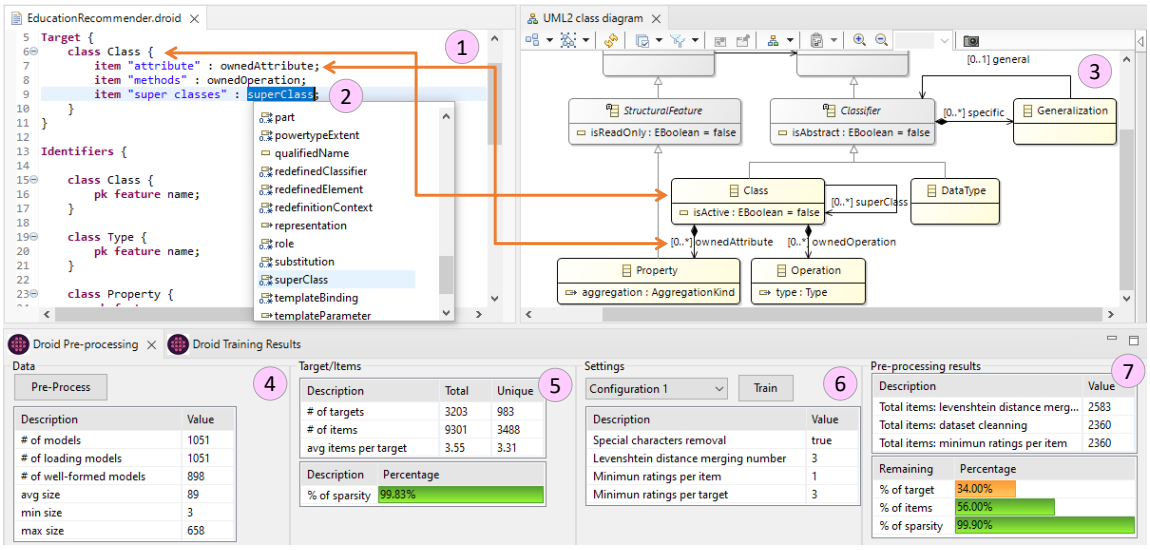

Figure 2: Droid development environment.

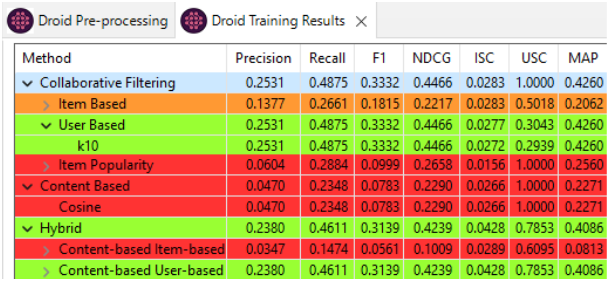

Figure 3: Training results view.

and user-based CBCF hybrids. Next, lines 7–11 specify how to split the data into training and test sets. This comprises the splitting approach (cross-validation or random), the number of folds (e.g., 10), and the division technique (per user or per item). Finally, lines 12–18 describe the evaluation protocol. This covers the metrics to evaluate the candidate RSs – a subset among precision, recall, F1, MAP (mean average precision), nDCG (normalised discounted cumulative gain), ISC (item space coverage), USC (user space coverage) –, the number of cut-offs (i.e., number of most relevant items used to calculate the metrics), the number of recommendations that the RS will suggest, and a threshold value upon which recommendations are deemed relevant.

To train the candidate RSs, the developer needs to choose a preprocessing configuration, and click on button *Train* (label 6 in Fig. 2). This opens the view of Fig. 3 with the metrics for all trained RSs. The view shows the RSs whose F1 value is in the top 20% in green, those with F1 under the median in red, and the rest in orange. In the figure, the best performant method is user-based CF.

2.4 Deployment of the RS

The developer can double-click on the most suitable recommendation algorithm in the view of Fig. 3, and Droid deploys it into a REST service called DroidREST. This is a generic recommendation service which becomes customised with the selected recommender. Client modelling tools can make POST requests to the service passing as parameters the recommender's name, the recommendation target (a class in our example) and the items it contains. The result is a prioritised list of recommended items for the target (attributes, methods and superclasses in our example).

2.5 Using the RS with EMF-based languages

Droid provides an out-of-the-box integration of the recommendation service with the tree editors generated by default for EMF-based languages. These editors are synthesized from the language's Ecore meta-model via predefined JET templates (https://bit.ly/3DyfN0F). We overwrote those templates to extend the editors with a menu that is available on the objects of the recommendation target, allows choosing the kind of recommended items, and then shows a list of recommendations which can be incorporated to the model.

Fig. 4 shows a generated RS for an object-oriented modelling language. The RS is integrated within the tree editor synthesized from the language meta-model. The figure shows the pop-up menu activated on an object of type Klass, the selection of items of type Attribute, and a dialog with the recommended attributes.

3 EVALUATION

The data preprocessing of Droid is a novel contribution of this paper. Hence, we performed an offline experiment to answer the question: "*Can data preprocessing improve the recommendations provided by Droid recommenders?*". To answer this question, we created a RS for class diagrams using the datasets and configuration for data splitting, recommendation methods and evaluation from [4], also shown in the listings of Sec. 2. For space limits, we report one dataset only, full results are available at https://bit.ly/3qTI4v1.

The left of Table 1 characterises the dataset, reporting on the number of models, targets (i.e., classes), items (i.e., attributes, methods and superclasses), items per target, model size, and data sparsity.

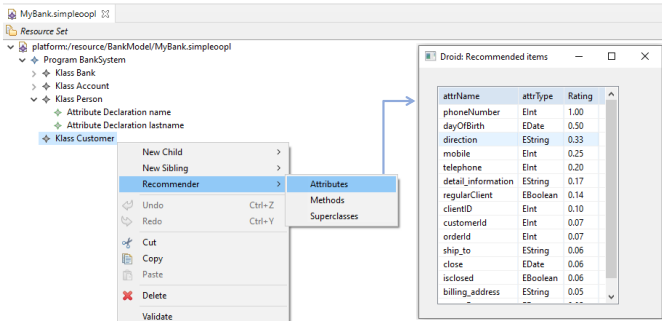

Figure 4: EMF tree editor augmented with the RS.

Table 1: Experiment: dataset characterisation and results.

Dataset		Results		
Num. models	1051	**Metric**	**No preproc.**	**With preproc.**
Num. targets	983	Precision	0.035	0.253
Num. items	3488	Recall	0.337	0.488
Items per target	3.31	F1	0.064	0.333
Avg. model size	89	MAP	0.184	0.426
Min. model size	3	nDCG	0.227	0.447
Max. model size	658	USC	0.830	0.294
Sparsity	0.9982	ISC	0.056	0.027

We used Droid to configure and train multiple RSs using the algorithms shown in the listing of Sec. 2.3. Then, we compared the RSs that resulted with and without preprocessing, in the former case using the preprocessing setting of Sec. 2.2.

The right of Table 1 compares the best RS configuration without preprocessing (*user-based item-based k10*), and the same configuration with preprocessing. F1 increased from 6.4% to 33.3%, precision from 3.5% to 25.3%, and recall from 33.7% to 48.8%. MAP and nDCG also improved about 50%. Instead, ISC and USC decreased, which was expected as there is a compromise between the precision-based and the diversity/coverage metrics. We thus answer our question positively: preprocessing improves the precision-based metrics, maintaining a balance with diversity/coverage. While further experiments are needed to better characterise the preprocessing effects, the metrics show improvement w.r.t. existing hand-crafted RSs for class diagrams, like [5], which reports a precision around 4%, or *MemoRec* [8], with F1 scores around 17%.

4 RELATED WORK

The modelling community has a growing interest in building RSs for modelling languages [3]. Most of them target popular structural languages like UML class diagrams or Ecore meta-models, but are built in an ad-hoc way. For example, DoMoRe [2] suggests domain concepts and names for new model elements, based on knowledge bases like WordNet. Burgueño et al. [5] propose a RS for class diagrams that extracts knowledge from text documents related to the project to recommend. Both RSs were created manually, with ad-hoc integration for a particular modelling language and tool.

RapMOD [12] recognises ongoing complex operations (e.g., pulling up common attributes), which the system completes. This different kind of assistance complements the one Droid offers.

Droid aims at reducing the effort when developing RSs, and other approaches also follow this line. Hermes [9] is an Eclipse framework to build RSs by the configuration of recommendation strategies. These must be specified manually and are limited to work only with Eclipse modelling tools. Hence, Hermes solves the integration with modelling tools, but the RS needs to be programmed. LEV4REC [16] is a lowcode tool to configure RSs. Similar to Droid, users can configure aspects of the RS, like the RS algorithm or the underlying libraries. For this purpose, users can select a configuration of a feature model, which yields a model that they can refine and use to generate code for the library of choice. The approach does not target RSs for modelling languages, so there is no way to customise the RS for the modelling language or to deploy or integrate the RS with a modelling tool. Moreover, it lacks support for exploration of the best strategies for preprocessing, or the performance of the selected RSs.

5 CONCLUSIONS AND FUTURE WORK

We have presented Droid, a tool to automate the construction of RSs for modelling languages. It provides a DSL to configure the language items to be recommended, the data preprocessing operations, the recommendation algorithms, and the evaluation metrics. The tool reports on preprocessing results and evaluates the chosen RSs against the selected metrics. The developer can select the most suitable RS for the language, which is deployed as a REST service. Droid supports off-the-shelf integration of the RS with the Eclipse tree editor, and it can be integrated with other technologies.

We are working on providing off-the-shelf integration of the RSs with other editors, e.g., based on Sirius or Xtext. We also plan to allow including other recommenders, e.g., based on neural networks, or built ad-hoc. For this, we will profit from the Eclipse extensibility mechanism defining suitable extension points in Droid.

REFERENCES

[1] G. Adomavicius and A. Tuzhilin. 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 (2005), 734–749.
[2] H. Agt-Rickauer, R. Kutsche, and H. Sack. 2018. Automated recommendation of related model elements for domain models *(CCIS, Vol. 991)*. Springer, 134–158.
[3] L. Almonte, E. Guerra, I. Cantador, and J. de Lara. 2022. Recommender systems in model-driven engineering: A systematic mapping review. *SoSyM* 21, 1 (2022), 249–280.
[4] L. Almonte, S. Pérez-Soler, E. Guerra, I. Cantador, and J. de Lara. 2021. Automating the synthesis of recommender systems for modelling languages. In *SLE*. 1–14.
[5] L. Burgueño, R. Clarisó, S. Li, S. Gérard, and J. Cabot. 2021. An NLP-based architecture for the autocompletion of partial domain models. In *CAiSE*. 91–106.
[6] T. Cerqueira, F. Ramalho, and L. B. Marinho. 2016. A content-based approach for recommending UML sequence diagrams. In *SEKE*. 644–649.
[7] M. C. de Oliveira, D. Freitas, R. Bonifácio, G. Pinto, and D. Lo. 2019. Finding needles in a haystack: Leveraging co-change dependencies to recommend refactorings. *Journal of Systems and Software* 158 (2019).
[8] J. Di Rocco, D. Di Ruscio, et al. 2022. MemoRec: a recommender system for assisting modelers in specifying metamodels. *SoSyM* (2022), in press.
[9] A. Dyck, A. Ganser, and H. Lichter. 2014. A framework for model recommenders - Requirements, architecture and tool support. In *MODELSWARD*. 282–290.
[10] M. Gasparic, T. Gurbanov, and F. Ricci. 2018. Improving integrated development environment commands knowledge with recomm. systems. In *ICSE-SEET*. 88–97.
[11] R. Haldar and D. Mukhopadhyay. 2011. Levenshtein distance technique in dictionary lookup methods: An improved approach. *ArXiv* abs/1101.1232 (2011).
[12] P. Mäder, T. Kuschke, and M. Janke. 2021. Reactive auto-completion of modeling activities. *IEEE Trans. Software Eng.* 47, 7, 1431–1451.
[13] P. Nguyen, J. di Rocco, D. di Ruscio, et al. 2019. FOCUS: a recommender system for mining API function calls and usage patterns. In *ICSE*. IEEE / ACM, 1050–1060.
[14] F. Ricci, L. Rokach, and B. Shapira. 2015. *Recommender Systems Handbook*.
[15] M. P. Robillard, R. J. Walker, and T. Zimmermann. 2010. Recommendation systems for Software Engineering. *IEEE Software* 27, 4 (2010), 80–86.
[16] C. Di Sipio, J. Di Rocco, D. Di Ruscio, and P. T. Nguyen. 2021. A low-code tool supporting the development of recommender systems. In *RecSys*. ACM, 741–744.
[17] M. Stephan. 2019. Towards a cognizant virtual software modeling assistant using model clones. In *NIER@ICSE*. IEEE / ACM, 21–24.