Software and Systems Modeling
https://doi.org/10.1007/s10270-024-01183-z

SPECIAL SECTION PAPER

ModelXGlue: a benchmarking framework for ML tools in MDE

José Antonio Hernández López[1] · Jesús Sánchez Cuadrado[1] · Riccardo Rubei[2] · Davide Di Ruscio[2]

Received: 12 May 2023 / Revised: 15 January 2024 / Accepted: 8 April 2024
© The Author(s) 2024

Abstract
The integration of machine learning (ML) into model-driven engineering (MDE) holds the potential to enhance the efficiency of modelers and elevate the quality of modeling tools. However, a consensus is yet to be reached on which MDE tasks can derive substantial benefits from ML and how progress in these tasks should be measured. This paper introduces ModelXGlue, a dedicated benchmarking framework to empower researchers when constructing benchmarks for evaluating the application of ML to address MDE tasks. A benchmark is built by referencing datasets and ML models provided by other researchers, and by selecting an evaluation strategy and a set of metrics. ModelXGlue is designed with automation in mind and each component operates in an isolated execution environment (via Docker containers or Python environments), which allows the execution of approaches implemented with diverse technologies like Java, Python, R, etc. We used ModelXGlue to build reference benchmarks for three distinct MDE tasks: model classification, clustering, and feature name recommendation. To build the benchmarks we integrated existing third-party approaches in ModelXGlue. This shows that ModelXGlue is able to accommodate heterogeneous ML models, MDE tasks and different technological requirements. Moreover, we have obtained, for the first time, comparable results for these tasks. Altogether, it emerges that ModelXGlue is a valuable tool for advancing the understanding and evaluation of ML tools within the context of MDE.

Keywords Benchmarking · Machine Learning · Model-Driven Engineering

1 Introduction

Machine learning (ML) is a fast-growing field with many applications in various domains and areas. For example, deep learning approaches have been successfully applied to solve complex problems in the Software Engineering domain (e.g., detecting code duplication [2], code analysis [51], or recommending relevant API function calls [39]). The integration of machine learning (ML) in software engineering has gained widespread attention in academia and industry. As a witness to this fact, the ASE'23 conference, as reported in [41], received an unprecedented number of 661 submissions, with over half dedicated to AI applications in software engineering. Many of these submissions explore the diverse applications of different ML approaches to enhance various software engineering tasks, with a focus on improving developer productivity and efficiency during coding.

In model-driven engineering (MDE), ML is increasingly being adopted to support different kinds of modeling tasks such as model classification [20, 54], clustering [9, 10], and providing real-time recommendations to modelers during modeling sessions [11, 20]. Thus, as suggested in [55], the trajectory of ML adoption in MDE is likely to parallel the trends observed in software engineering.

Selecting the appropriate ML tool for a particular task can be difficult, as different ML tools may have different strengths and weaknesses. Furthermore, many factors can affect the accuracy of the ML tools under consideration, such as the underlying models used, the quality of training data, and the available processing power. Therefore, benchmarking ML tools is essential for assessing their performance and suitabil-

Communicated by N. Bencomo, M. Wimmer, H. Sahraoui, and E. Syriani.

✉ Jesús Sánchez Cuadrado
jesusc@um.es

José Antonio Hernández López
jose.antonio.hernandez.lopez@liu.se

Riccardo Rubei
riccardo.rubei@univaq.it

Davide Di Ruscio
davide.diruscio@univaq.it

[1] Linköping University, Linköping, Sweden

[2] Università degli studi dell'Aquila, L'Aquila, Italy

Published online: 10 June 2024

ity for various scenarios with the goal of making progress in the field.

Currently, the modeling community lacks a unified understanding of which MDE tasks can benefit from ML and how to measure progress in these tasks. This is mainly due to the lack of a standardized framework for systematically evaluating and comparing different approaches. In contrast, most domains of ML application have a common understanding of the tasks that ML can address, and there are established benchmarks to evaluate new approaches. GLUE[1] and CodeXGlue benchmark[2] are examples of benchmarks for language understanding and for "deep code," respectively. These benchmarks provide the resources needed to evaluate new ML tools for specific tasks. However, in the case of MDE, the community is still missing a reference framework to evaluate new tools that solve modeling problems using ML.

This paper introduces a benchmarking framework called ModelXGlue, designed to support the comparison of ML-based systems for MDE tasks. Researchers can customize a benchmark for a specific ML/MDE[3] task by selecting a dataset, ML models (potentially created by different researchers), and relevant metrics to measure the performance of the ML models in the selected task.

The proposed framework is extensible and can accommodate new ML models without modifying the framework's source code. Each component can operate in a separate environment to prevent conflicts with libraries and enable the use of various technologies. Additionally, the system becomes completely automated once input configurations are provided. To assess the effectiveness of ModelXGlue, we integrated multiple ML/MDE tools proposed in the literature for three types of MDE tasks, namely *model classification*, *model clustering*, and *feature name recommendation*. We study the different requirements each ML tool poses in terms of execution platforms, dataset formats, encoding transformations, and more. We show how the framework's features enable us to address such requirements. We developed three reference benchmarks for these tasks and compared the outcomes of different ML models already published in the literature.

This paper extends and generalizes the methodology proposed in [32], and makes the following contributions:

- *Challenges for benchmarking ML tools for MDE tasks*: Distilling essential challenges from existing literature, we provide researchers with a consolidated understanding of the current landscape for benchmarking ML tools in the context of model-driven engineering (MDE) tasks;
- *A benchmarking framework for ML tools in MDE*: Presenting and discussing the main components of ModelXGlue, we showcase their applications through representative examples, offering a comprehensive benchmarking framework tailored for MDE;
- *Reusable benchmarks for three ML/MDE tasks*: This paper introduces the development of ML benchmarks for three crucial MDE tasks, including model classification, clustering, and feature name recommendation;
- *Use of* ModelXGlue *benchmarks in practice*: Demonstrating the practical application of the developed benchmarks, we compare state-of-the-art ML approaches for the considered MDE tasks. This facilitates the reuse of implementation idioms for others in their own ML/MDE tasks.

Both the developed framework and the benchmarks produced for this work are made openly available for future research[4]

Organization. Sect. 2 reviews related works in the ML and MDE field and discusses the motivation of this research based on the limitations found with respect to how such works are evaluated. Section 3 delves into existing challenges for benchmarking the application of ML models for MDE tasks. Next, in Sect. 4, we introduce the ModelXGlue framework using a comprehensive example. Section 5 discusses three MDE tasks that can be addressed through ML algorithms which we have used for the evaluation. We evaluate the ModelXGlue framework in Sect. 6. Finally, in Sect. 7, we conclude the paper and discuss future work.

2 Related work and motivation

This section discusses the significance of benchmarking, elucidating the challenges inherent in benchmarking ML tools and exploring how the ML community has addressed these issues (Sect. 2.1). This introduction sets the stage for a comparative analysis with the situation within the model-driven engineering (MDE) field. Subsequently, in Sect. 2.2, we scrutinize recent initiatives employing ML tools to tackle MDE problems. Our analysis includes an exploration of whether these works incorporate benchmarking elements such as metrics, comparisons against other models, and other relevant aspects.

[1] https://gluebenchmark.com/

[2] https://github.com/microsoft/CodeXGLUE.

[3] With the acronym ML/MDE we refer the application of machine learning to support the development of some MDE task.

[4] ModelXGlue is made available as an open source project in http://github.com/models-lab/modelxglue. Additionally, the benchmarks (including the required resources) presented in this paper are available at http://github.com/models-lab/modelxglue-mde-components.

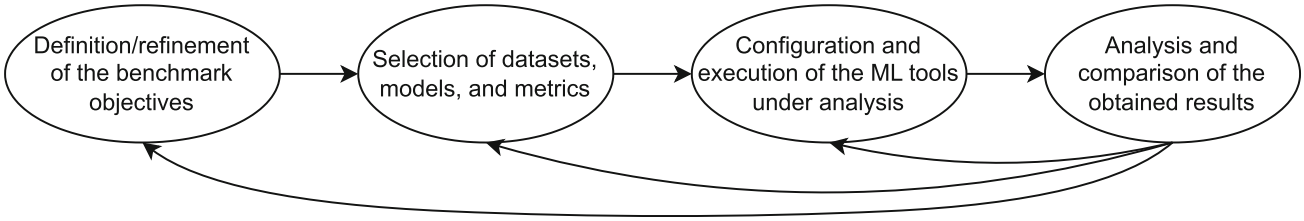

Fig. 1 A bird's eye view of a typical ML benchmarking process.

2.1 Overview of benchmarking ML tools

Benchmarking is a systematic process of evaluating and comparing machine learning tools based on various key performance indicators (KPIs) that reflect their quality and efficiency [40]. Benchmarking can help users to choose the most suitable tool for their needs and discover opportunities for improvement and optimization. However, benchmarking ML tools is a complex and challenging process, as it requires careful planning and execution of multiple steps.

As shown in Fig. 1, benchmarking ML tools encompasses an iterative process consisting of several steps, including the following ones [15]: *i)* defining the benchmarking goal and scope, *ii)* selecting the appropriate dataset, models, and metrics to use, *iii)* configuring and running the ML tools with the selected dataset, *iv)* analyzing and comparing the results obtained from the different tools. Each step involves several decisions and trade-offs. Therefore, a rigorous and consistent methodology is essential for conducting a fair and reliable benchmarking study [40].

The goal and scope of benchmarks are related to the type of task that is evaluated and how it will be used in practice. For instance, we might be interested in restricting the task to classifying numbers for the image classification task (i.e., assigning a label to an artifact). In contrast, if the model is intended to be used for driving assistance, then we might be interested in restricting the scope of the benchmark to traffic signs. According to the benchmark goals, we must choose the appropriate dataset among the available ones. In this context, different ML communities have created many referenced datasets. For example, there are general image recognition datasets like ImageNet [50] (one of the standard datasets) and COCO [26], which consists of extensive image collections, while task-specific datasets such as MNIST (for number classification) and the German Traffic Sign Recognition Benchmark (GTSRB) dataset [52] cater to specific evaluation needs.

The selection of ML models to be included in benchmarks depends on the task, scope, and selected dataset. For instance, the MNIST dataset is adequate for single-label classification models, but if we are interested in multi-class classification (i.e., predicting more than one label), then the dataset and the models need to provide specific support. Concerning the selection of the metrics used for benchmarking purposes, we could consider accuracy as a relevant metric to compare the proposed model to the current state-of-art algorithms on a task. Finally, we must build the infrastructure to train, execute and compare the results systematically.

Over the last years, the ML community has adopted different benchmark suites [34]. One of the most relevant tools to perform machine learning benchmarks both in training and inference is MLPerf [44]. It supports different datasets such as ImageNet, COCO and tasks like image classification and recommendation. At the same time, the software engineering community has established some common downstream tasks (such as program translation [27]) and their corresponding metrics (such as CodeBLEU [45]) for evaluating ML models in this domain. This has facilitated the creation of benchmarks like CodeXGLUE [33] to enable fair and consistent comparison of different ML models. The availability of such benchmarks has been crucial for advancing the field. However, as discussed below, there is currently a research gap in the modeling community with respect to the systematic evaluation of ML models applied to modeling problems.

2.2 Machine Learning in MDE

Machine learning has proven effective in addressing various challenges within model-driven engineering [6, 24]. In [12], the authors discussed the *cognification* of model-driven software engineering. The goal is to boost the performance of a process using knowledge. Furthermore, a comprehensive examination of the adoption of machine learning for managing modeling ecosystems is presented in [22]. Although there is a trend to apply machine learning to address MDE problems, there is not a well-established mechanism to evaluate such ML applications and compare them against previous works.

In the following, we make an overview of relevant works that build ML tools to address problems in MDE, focusing on how such works are evaluated and made available for others to compare. In particular, we describe works, which support the MDE tasks of *model classification*, *model clustering*, and *modeling assistance*. Table 1 and Table 2 summarize how the analyzed works address the following dimensions:

Table 1 Outline of works related to MDE/ML and which facilities are provided for making experiments repeatable

		Task	Availability	Automation	Experiments	Comparison
Model Classification						
1	[36]	Single label inference	GitHub	No	Yes	No
2	[37]	Single label inference	GitHub	No	Partial	Yes
3	[38]	Single label inference	GitHub	No	Partial	Yes
4	[28]	Single/Multi-label inference	GitHub	Partial	Yes	Yes
Model Clustering						
5	[10]	Hierarchical clustering	No	No	N/A	No
6	[7]	Hierarchical clustering	Web	Partial	Yes	No
Modeling Assistance						
7	[23]	Feature name recommendation	GitHub	Partial	Yes	No
8	[20]	Feature name recommendation	GitHub	Partial	Yes	No
9	[54]	Feature name recommendation	GitHub/Zenodo	Partial	Yes	No
10	[11]	Feature name recommendation	GitHub	No	N/A	No
11	[16]	Feature name recommendation	GitHub	No	N/A	No
12	[1]	Next modelling operation	Zenodo	No	Yes	No
13	[14]	Feature name recommendation	No	No	N/A	No

Table 2 Outline of works related to MDE/ML focusing on which elements are used to produce and evaluate the ML models

		Dataset	Pre-processing	Training/ Inference	Evaluation	Metrics
Model Classification						
1	[36]	Ecore-555	Java	Python	Predicted label	Acc., prec., recall, F_1
2	[37]	Ecore-555	Java	Python	Predicted label	Prec., recall, F_1, acc., FPR, ROC, AUC
3	[38]	Ecore-555	Java	Python	Predicted label	Prec., recall, F_1, bal. acc., FPR, ROC, AUC
4	[28]	ModelSet	Java	Python	Predicted label(s)	Prec., recall, F_1, acc.
Model Clustering						
5	[10]	GitHub	Java	Java	Predicted cluster	Correlation
6	[7]	ATL Zoo	Java	R	Predicted cluster	Prec., recall, $F_{0.5}$
Modeling Assistance						
7	[23]	Ecore-555 [5], ModelSet [29], GitHub	Java, Python	Python	Element removal	SR, prec., recall, F_1
8	[20]	Ecore-555 [5], GitHub	Java	Java	Element removal	SR, prec., recall, F_1
9	[54]	MAR [30][31]	Java	Python	Element removal	Recall, MRR
10	[11]	Private	Java	Python	Element removal	Prec., recall
11	[16]	ModelSet [29]	Python	Python	Element removal	Prec., recall
12	[1]	Matlab dataset [17]	Matlab, Python	Matlab	Element removal	MRR, recall
13	[14]	GitHub Java corpus	-	-	Element removal	Prec., relevance

– *Availability:* The location where the source code of the tool, including the training and inference code, is available.
– *Automation:* Whether the authors provide scripts (or any other method) to automate the development process, which includes model training, inference, and re-execution of the original experiments. In this item, *Yes* means that the process is fully automated, *Partial* that some scripts are available and *No* that no automation is considered.
– *Experiments:* This category checks if the experiments are available, including the original data and other artifacts needed to re-run experiments.
– *Comparison:* Whether the original paper compares the proposed approach with baselines (e.g., previous approaches proposed in the literature).
– *Datasets:* The dataset(s) originally used to evaluate the approach.
– *Preprocessing:* The technology used to preprocess the dataset and convert it into a suitable form for the ML model.
– *Training/Inference:* The technology used to implement the training and inference of the ML model.
– *Evaluation:* Specifies the ground truth, which can be defined manually (e.g., a dataset manually annotated with labels) or computed (e.g., removing elements of a model and using them as the ground truth for recommendation).
– *Metrics:* The set of metrics used to measure the performance of the model in the given task.

Model classification. AURORA [36, 37] is a tool that exploits a feed-forward neural network to classify meta-models. The authors proved the tool's capability to classify Ecore models accurately. Similarly, a convolutional neural network (CNN) is used to build the MEMOCNN meta-model classifier [38]. The idea is to transform meta-models into particular 2D structures that the CNN can process. However, both AURORA and MEMOCNN may be biased due to the small size of the dataset (555 Ecore meta-models) and the presence of duplicates. These works corresponds to rows 1–3 in Tables 1 and 2. In one of them, the experiments are fully available, while in the others, crucial information is missing, specifically the comparison with the baselines. Furthermore, the source code released in those works is not in a form that is easily runnable. The dataset presented in [28] is accompanied by two exemplary applications of model clas-

sification (row 4). The first one is single-label inference, and the authors compare against [36], but to do so, the AURORA neural network is reimplemented from scratch. The second application is multi-label inference. The source code is available on GitHub and partially automated via scripts.

Model clustering. Basciani al. [10] proposed a tool to group meta-models. The tool leverages a hierarchical clustering technique to organize meta-models in a given repository automatically. Furthermore, the authors exploited different similarity measures, some conceived specifically for meta-models, to trigger the clustering. The replicability of this work is poor since the source code and the experiments are not available (see row 5). The SAMOS platform is used in [7, 8] for clustering. The approach is based on transforming models into graphs and extracting paths (in the form of $n-$grams). The source code and experiments are available in this case (row 6). However, the implementation is in R, which introduces additional complexity to re-execute the experiments. Although the authors provide a VirtualBox environment to facilitate the task, it is still difficult to use such an environment in an automated pipeline.

Modeling assistance. The authors in [18] propose the application of graph kernels to MDE, but no actual implementations are provided. In this context, Di Rocco et al. [21, 23] proposes MORGAN, a tool based on graph kernels to support the completion of both models and meta-models. The tool is conceived to recommend structural features to complete the models under construction. In particular, MORGAN can recommend classes, fields, and methods within a model class, metaclasses, and structural features like attributes and references. Another approach intended to recommend model features is MemoRec [20]. It is based on collaborative filtering to recommend class or structural feature names, depending on the current editing context. The EcoreBERT model [54] uses a randomly initialized RoBERTa model pretrained on all the Ecore meta-models of the MAR search engine. It is used to recommend feature names.

Burgeño et al. [11] proposed an NLP-based architecture to complete software models. The core of their proposal is to use two-word embedding models (e.g., GloVe [42], word2vec [35], etc.), one trained with general knowledge and the other with contextual knowledge. Both models are interpolated to perform the recommendations. In [16], a recommender for class names and feature names is proposed based on generating a prompt for querying OpenAI's GPT-3 API. An assistant for Simulink models is presented in [1]. The system can recommend new blocks to add to the model under specification (i.e., akin to predicting the next edit operation).

Table 1 summarizes in rows 7–13 the facilities provided by model assistance works. Although in all cases the implementations are available, there is typically a lack of facilities to automate the reproduction of the works (e.g., few scripts are provided, but just the code as is). Moreover, it is worth noting that these works do not compare against each other.

2.3 Assessment

From the results summarized in Tables 1 and 2, we can draw some conclusions about the need for better benchmarking infrastructures for ML and MDE. Most works make their implementations public, but they are barely usable for others to reuse since they are released "as is" (few or no scripts to automate simple tasks like training the model or doing inference). In some cases, even if there are scripts, the implementation is not designed to run on machines different from the one in which it was implemented (e.g., it is not rare to encounter absolute paths in the original source code, which makes it difficult to reuse the code in different machines or with different data).

In 6 out of the 13 analyzed works, the comprehensive data necessary to replicate the experiments is fully accessible. At the same time, the remaining studies provide only partial or no access to their data. Another noteworthy observation is that only three out of the 13 works assess their proposals against other state-of-the-art works or baselines. The similarity in the evaluation methodologies employed across these works, such as utilizing an element removal strategy in recommendation systems, suggests the feasibility of comparing different approaches. Nevertheless, the absence of such comparisons in practice reflects the challenging nature of this endeavor, often leading researchers to avoid paying more attention to it. However, achieving scientific progress in the application of ML to MDE necessitates the undertaking of such comparisons. Notably, each published work employs a distinct set of metrics, contributing to the challenge of comprehending the performance of each approach relative to others.

The goal of ModelXGlue is to provide MDE researchers with a framework to systematically evaluate their ML tools and facilitate the comparison against state-of-the-art approaches.

3 Challenges for benchmarking ML tools in MDE

The preceding discussion highlights a lack of consensus within the modeling community regarding the provision the adequate experimental details in ML/MDE works for comparative purposes. A majority of works refrain from comparing against prior approaches, complicating the assessment of each approach's relative merits. Additionally, the diversity of metrics employed in the evaluation of these works further impedes straightforward result comparisons. In essence, it can be asserted that the evaluation of ML/MDE works

lacks systematicity. Addressing this issue could involve the implementation of benchmarks that facilitate a systematic comparison of various ML tools applied to MDE tasks.

To support such a statement and introduce the challenges involved in benchmarking ML/MDE tasks, we discuss the *model classification* task, which has been subject to different proposals over the last few years [32, 37]. The task consists of assigning a meaningful label to a given model based on the labels observed in the training data. This problem has received increasing attention in recent years, as it can facilitate the exploration and analysis of extensive collections of models [30]. In particular, one of the challenges of reusing modeling artifacts from a repository is to ensure that they are properly categorized according to their characteristics. This categorization has traditionally been done by manually adding metadata to the artifacts, with different levels of detail depending on the specific problem. However, this manual process is tedious, time-consuming, and error-prone.

Challenges. Although model classification is a relatively simple task, the required pipeline includes most of the challenges typically found when combining ML and MDE [32]. For explanatory purposes, Fig. 2 illustrates some of the challenges using three concrete model classification approaches found in the literature: *feed-forward neural network* [36] (FFNN), a *graph neural network* [32] (GNN) and *k-nearest neighbors* (KNN) using a search engine [30]. The concrete challenges that we identify are the following:

- **Dataset selection, filtering and adaptation**. First of all, a *dataset* needs to be *selected* and all the approaches must be made compatible with such dataset to ensure that the results are comparable. This means that a common format for (modeling) datasets is desirable. Many times filtering and adaptation operations over the dataset are needed, like detecting and removing (quasi)-duplicate elements. As previously mentioned, it is important that such transformation is applied equally to all approaches.
- **Encoding and preprocessing**. The dataset needs to be *encoded* according to the input requirements of each ML model. For instance, if the selected model is a feed-forward neural network, a possible encoding is to extract a bag of words from the model (e.g., as 1-gram) and then use an embedding model (e.g., GloVe) for obtaining numerical vectors. On the other hand, a GNN requires transforming the models to graph with a specific format, whereas an approach like MAR can treat EMF models directly because it relies on a specific library which internally implements their own transformations. Since we are interested in enabling the use of different datasets, it is important that the tools do not couple this step with the training and inference.
- **Training and validation strategy**. Each ML *model* needs to be *trained* using a subset of the training data and later *tested* using another subset. There are different strategies for this (e.g., n-fold validation, train-test-validation splitting, etc.), but it is important that all compared tools follow the same strategy.
- **Evaluation metrics**. The evaluation of each tool must follow the same metrics, which in turn need to be selected according to the actual task. In practice, this requires that the tools provide the raw output so that it can be post-processed by the evaluation framework.
- **Heterogeneous environments**. The different steps of the process require heterogeneous execution environments. In practice, every approach handles the overall pipeline execution in different ways like creating execution scripts (which are difficult to change) or just documenting the commands that needs to be executed. This fact hampers the possibility of seamlessly benchmarking and comparing different approaches.

In summary, systematically benchmarking ML tools in MDE presents challenges, mainly due to the need for more standardization and reproducibility. Various tools may have different implementations, configurations, or versions that can impact their performance. Additionally, different datasets, models, or training strategies may cause variability and inconsistency in the results. As a result, it is crucial to use standardized and reproducible methods to ensure fair and reliable benchmarking.

4 Framework

In this section, we describe the ModelXGlue framework, which we have designed to facilitate benchmarking ML models specifically created to address MDE tasks. The ultimate goal is to allow modeling researchers to assess new models systematically by means of a clearly defined validation pipeline and, thus, to support and facilitate the advancement in the field.

4.1 Framework requirements

As discussed in the previous section, by considering the illustrative example of the model classification task, several shortcomings have hindered the systematic benchmarking and comparison of ML/MDE approaches. To address this issue, our framework needs to satisfy at least the following *requirements* that we elicited and generalized while working on the approach proposed in [32]:

- **Management of different datasets**. The availability of several datasets of different quality is a necessary element for advancing the discipline. Therefore, as new datasets arise, it is crucial to be able to use them to

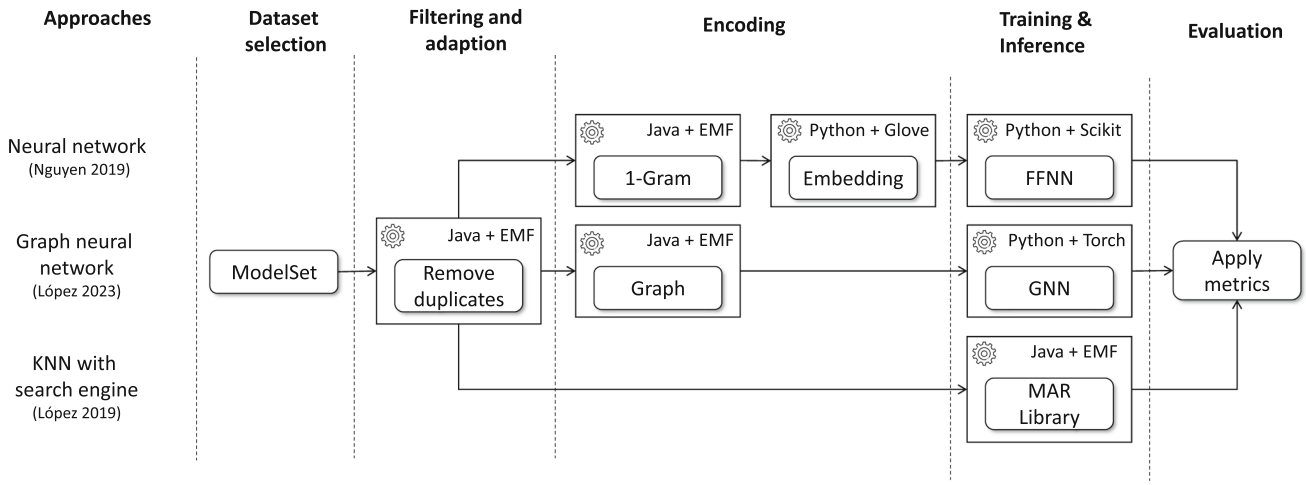

Fig. 2 Three approaches for model classification and what it is needed to execute and compare them. The geared icon means that the process needs to be executed in a specific execution environment.

compare existing models against new datasets. Hence, the framework must provide standard formats and data management mechanisms to easily use different datasets while executing comparison pipelines.

– **Management of different metrics**. The task of selecting and applying the appropriate metrics for a given ML model is an important step which is the subject of intense research in the ML community. A metric is a quantitative measure of how well a model performs on a specific task. Various metrics may be suitable for different tasks and models. Therefore, choosing and implementing the metric that best reflects the model's objective is crucial. Additionally, all models that are compared should be evaluated using the same metric. Therefore, for the sake of fair comparisons, the proposed framework must provide mechanisms to ensure that the same metrics are used for the ML models under analysis.
– **Management of execution environments**. Each pipeline element must be executed in its own environment with the needed dependencies automatically configured. The framework cannot assume a unique setting for all ML models, since each model will depend on specific versions of the exploited external libraries. In particular, in MDE, there is a plethora of technologies involved in the construction of tools, which may make it difficult to run experiments within a common framework. For instance, some tools may be implemented in Java, others in Python, or even depend on external services.
– **Agnosticity from the provenance of ML models.** The framework should be able to handle ML models from different sources and created by third parties (e.g., researchers who are not involved in the development of ModelXGlue). This means that when creating a benchmark, it must be possible to combine ML models whose provenance is different: a model created by a third party used in its original form, a model which is adjusted for its integration in the framework (e.g., to provide a fair comparison against other models), or even create new models from scratch.
– **Extensibility**. The framework should be modular so that new data transformations (e.g., filtering, encoding, etc.) and ML models can be plugged in without changing the framework's source code. For example, one should be able to add a new ML model for model classification (e.g., using a CNN) by simply interfacing with the framework. This is an essential requirement because we want to encourage the collaboration and improvement of the framework by getting contributions from MDE researchers who are working on ML and MDE and willing to share their novel tools for reuse, reproducibility, and comparison.
– **Automation**. The framework should automate benchmarking phases reducing human interventions. This includes allowing different configurations easily, including the specification of hyperparameters and the selection of the models to be compared.

4.2 ModelXGlue components and usage

Figure 3 provides an overview of the core components constituting the ModelXGlue framework. In our proposal, there are two main user roles: the ML expert and the benchmarker. The ML expert is typically a researcher who has built an ML model to address some MDE task. He or she is in charge of wrapping the model into a format that is readable by ModelXGlue (more details below). The other role is the benchmarker, whose task is to gather ML models published by ML experts (e.g., in GitHub repositories) and use the

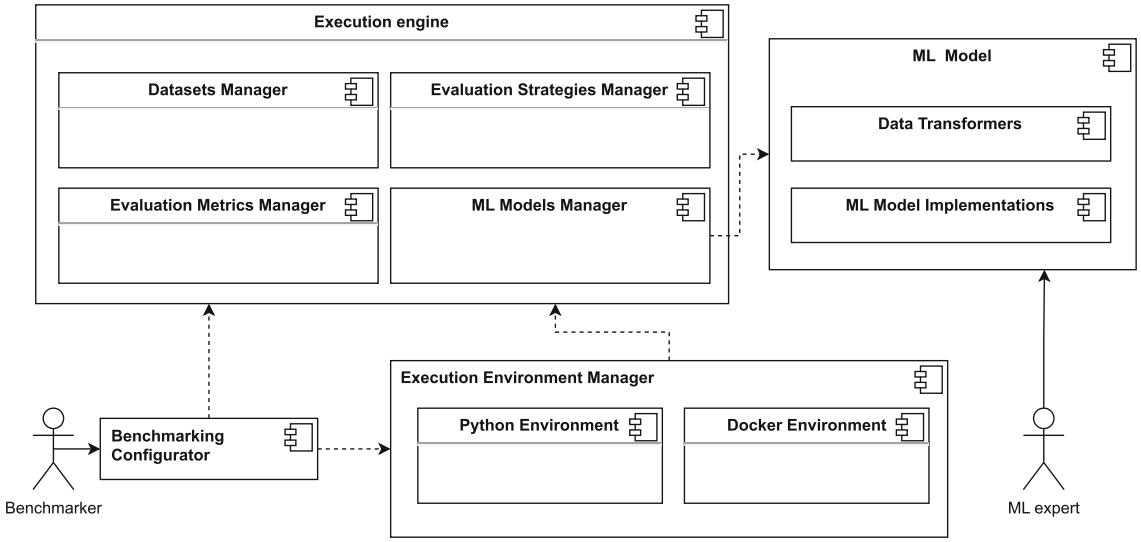

Fig. 3 Overview of the ModelXGlue components.

benchmark configuration facilities provided by ModelXGlue to set up a concrete benchmark (e.g., for model classification). These two roles can be played by the same person (or team) or by different people. To enable the interaction of the users, the framework is organized into four key components:

- *Execution engine*: This component is in charge of coordinating the execution of a benchmark by setting up its pipeline and executing the different steps. In particular, the execution engine uses the execution environment manager component to execute the evaluated ML models. To perform its job, it comprises dedicated subcomponents responsible for overseeing reusable datasets, the application of evaluation strategies (e.g., k-fold or train-test validations), the adoption of evaluation metrics, and the utilization of ML models available in the catalog contributed by ML experts, as discussed below.
- *ML Model Implementations*: ML experts can create and share new ML models within the benchmarking framework. All ML models are stored in a designated location (e.g., a GitHub repository), allowing the framework to discover and make them available. In addition to the actual ML model implementation (e.g., a neural network implemented in PyTorch), a ML model component also encompasses the transformations required to consume input datasets and encode them as needed. To integrate an ML model into a ModelXGlue, a manifest file must be provided. Such manifest includes the information about how to set up the execution environment of the model, so that it can be integrated as part of a concrete benchmark.
- *Benchmarking Configurator*: This component manages benchmark configurations, facilitating the execution of benchmarks that consume specified datasets to feed selected ML models. The evaluation is carried out in accordance with the chosen evaluation strategies and metrics.
- *Execution Environment Manager*: This component facilitates the execution of the ML components (i.e., the data transformations and the ML model implementations). Currently, the framework supports two different environments: Docker and Python.

This architecture allows us to satisfy the requirements stated above. In particular, the Execution engine component is able to load different datasets provided that they have a common interface (requirement *Management of different datasets*).

To satisfy the requirement about the *Management of different metrics* the Evaluation Metrics Manager provides a set of well-known metrics which can be selected by the benchmarker and the system ensures that they are applied consistently to all models of the benchmark.

The requirement related to *Management of execution environments* is satisfied by the fact that every ML model is executed by the Execution Environment Manager, which is in charge of providing the required dependencies and a clean execution environment.

The *Agnosticity from the provenance of ML models* is achieved by the fact that an ML model must always be wrapped as a ModelXGlue (i.e., it becomes an ML Model component as shown in Fig. 3). Such wrapping is typically easy since it implies providing a manifest file describing how to execute the implementation.

Regarding the *Extensibility* requirement, our architecture satisfies it, as new ML models are plugged in by benchmarkers simply by referring to them when configuring

benchmarks. The only requirement is that they comply with the interface required by the used ML model component.

The *Automation* requirement is achieved by the interaction of the different components: once a benchmark is configured, the execution engine uses the Evaluation Strategies Manager to setup a specific pipeline and apply it to all ML models in the benchmark.

The execution of the pipeline includes several steps which are managed automatically: loading the dataset, splitting it according to the selected evaluation strategy (e.g., k-fold), then using the ML models manager to load an ML model, applying the required data transformations to the original dataset (Data transformers) and then using the ML model implementation to train the model. After the model is trained, the model is evaluated using the corresponding test set, and the selected metrics are used for evaluation.

To summarize, a ModelXGlue benchmark is a specification to select and configure the machine learning models to be employed on a MDE task of interest. This specification also encompasses the chosen datasets, evaluation strategies and metrics used for the comparison. The machinery provided by ModelXGlue is able to automatically execute the given benchmark by training the models, applying them to the corresponding tests, and finally computing the corresponding evaluation metrics.

4.3 ModelXGlue in practice

To provide a concrete illustration of the elements of ModelXGlue, we consider a scenario in which a researcher wishes to conduct experiments related to the Ecore model classification task, where the goal is to compare various existing ML models designed for classifying Ecore models. Figure 4 provides an overview of the components of ModelXGlue in action, which can be split in two main elements: catalog of reusable ML models and the construction of benchmarks. The description of the catalog and the benchmarks is done with YAML configuration files, which are explained in more detail in Sect. 4.4. In the rest of the section, we present a high-level view of ModelXGlue by means of Fig. 4.

The **catalog of ML models** consists of sets of ML models to address specific tasks (model classification in the example) which have been made available by ML experts. In particular, the shown catalog consists of two available models for the explanatory model classification benchmark problem: the KNN approach using the MAR search engine [30] ❶ and the feed-forward neural network model proposed in the AURORA tool [36] ❷ . These models are "external" to ModelXGlue in the sense that they can be developed and made available in shared repositories independently of ModelXGlue.

To **construct a benchmark**, the user of ModelXGlue (the benchmarker) can select the actual models being benchmarked from the catalog by means of configurations as given in ❸ (ffnn.yaml and mar.yaml). Theses specification outlines which ML model will be evaluated and the specific settings for the hyperparameters that will be explored during benchmarking. The construction also involves defining the datasets that will be used as input for the benchmark ❹ and the description of the task to be benchmarked which includes information about the type of evaluation (train-test, k-fold, etc.) as well as the relevant evaluation metrics ❺ .

The lower part of Fig. 4 illustrates the execution of a benchmark and the steps involved in it. To launch an execution of ModelXGlue for a set of experiments, the user just indicates the folder in which the benchmark is located (parameter config-path), the name of the models to be evaluated (ffnn and mar in this case), the dataset to be used (like the one given in ❹) and the task to be executed (e.g., the one specified in ❺). The framework implements a pipeline that consumes such an input configuration by executing the following five steps:

(S1): First, a concrete dataset is loaded. It is possible to seamlessly change the dataset just be indicating a different one in the execution command.

(S2): The dataset typically needs to be transformed to extract features with which train an ML model. Therefore, this step is dependent on the selected ML model. The ML model description is interpreted to transform the features as needed. For instance, for the FFNN, a vectorization based on TF-IDF or GloVe is used.

(S3): The model is trained according to the selected sampling strategy (e.g., train-test, k-fold, etc.) and as many times as the hyperparameter selection requires.

To this end, ModelXGlue redirects the training to the concrete implementation, setting up the appropriate execution environment. In particular, we currently support Python environments and the most flexible solution which is to run Docker containers.

(S4): The test phase (which may be interleaved with the training in some cases, e.g., with a k-fold strategy) works similarly as the training phase, by using the execution environment.

(S5): Finally, the results obtained by each ML model execution (according to the combinations of hyperparameters) are evaluated using metrics among those made available by ModelXGlue.

4.4 Main ModelXGlue features

In the following, the main features of the framework are discussed with respect to the features presented at the beginning of this section.

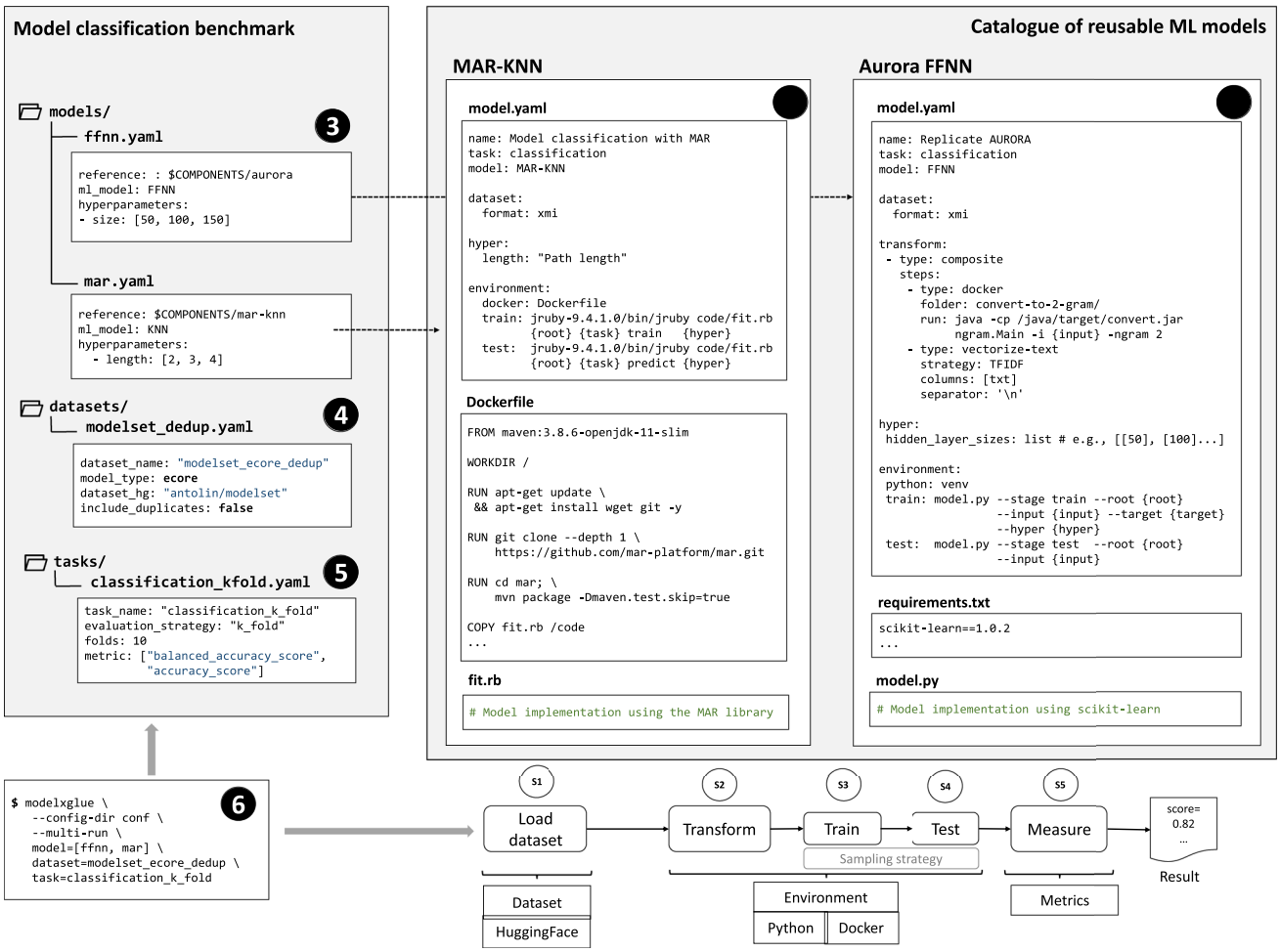

Fig. 4 Components of the framework. Researchers build catalogs of ML components for MDE tasks which can be integrated into benchmark. ❶ and ❷ shows the configuration, dependency specification and implementation of the two ML models (the MAR as a KNN classifier [30] and the FFNN of Aurora [36]). ❸ A benchmark is configured by referring to available ML models and specifying values for the hyperparameters of the models. ❹ One ore more datasets need to be configured, in this case ModelSet removing duplicate models. ❺ Description of the task to be performed (classification), along with the evaluation strategy (k-fold) and the relevant metrics. ❻ Command to execute a benchmark. ModelXGlue reads the configuration and for each experiment applies the configured pipeline and outputs a file with the metric scores.

4.4.1 Management of different datasets

To support different modeling datasets, ModelXGlue takes advantage of the dataset hub of HuggingFace.[5] In HuggingFace, a dataset consists of a table with a number of columns. In our case, a modeling dataset is defined in terms of the following columns:

- `id`: it is the unique identifier of the model in the dataset.
- `xmi`: it contains the XMI serialization of the stored model.
- `model_type`: it is a string indicating the type of model (e.g., Ecore, uml, etc.)
- `text` (optional): it contains the model serialized as 1-gram (i.e., the strings of the model).
- `graph` (optional): it contains the model serialized as a graph in networkx format.
- `labels`: for supervised learning datasets, it contains a comma-separated list of labels of the model.
- `is_duplicated` (optional): it indicates whether the model must be discarded in case of experiments that have to be executed without duplicates.

The only mandatory elements of a dataset is the *id*, the *xmi* text of the model and the *type of model*. The rest are optional. In particular, the *text, graph*, and *is_duplicated* elements are computed applying a preprocessing step of the dataset before uploading it to HuggingFace. We do this because such

[5] https://huggingface.co/docs/datasets/index

transformations are used very frequently in ML models, and therefore we have decided to avoid recomputing them by adding them as part of the dataset.

Using a specific dataset in ModelXGlue involves writing a configuration file like ❹ shown in Fig. 4. The property *dataset_name* is used when running a benchmark to refer to a physical dataset stored in the repository with such a name. The *dataset_hg* property represents the ID of the dataset in the HuggingFace hub. On the other hand, the *model_type* attribute is used to select only Ecore models from the physical dataset and prepare them for the benchmarks using the dataset called *modelset_ecore_dedup*. This mechanism allows for the creation of various views of the same physical datasets, depending on the purpose of the comparison.

4.4.2 Management of different metrics

The framework provides built-in support for a number of performance metrics, and new ones can be added if needed. However, an important aspect of applying metrics to evaluate the performance of ML models is to make sure that they are applied consistently for every evaluated model. To this end, in ModelXGlue, a benchmark is configured around the notion of *task*. A task includes an evaluation strategy (e.g., k-fold or train-test-validation) and a set of relevant metrics. The framework makes sure that all models are evaluated in the same way and that the metrics are computed equally in all cases. In practice, this means configuring a task as shown by ❸ in Fig. 4. Essentially, a task is given a name and a concrete evaluation strategy, which are used when running the benchmark. The metrics are configured by referring to the name of the metrics provided by the framework. Some relevant tasks along with the associated metrics are described in Sect. 5.

4.4.3 Management of execution environments

One of the distinctive features of ModelXGlue is its ability to simplify the process of setting up the environments required to run various ML models. Typically, this process involves dealing with dependencies and potential conflicts among third-party libraries that are needed for execution, which can be time-consuming and error-prone. ModelXGlue addresses this challenge by making explicit the configuration of the execution environment for each model under analysis and isolating such environment from the rest of the models, thus minimizing the risk of dependency clashes and eliminating the need to install unnecessary libraries that are not required for the specific models being tested. For instance, if a researcher is only interested in using a specific model, such as FFNN, they do not need to install dependencies for other models, such as GNNs, which can be a non-trivial task due to GPU-related libraries.

To address this issue the framework provides the notion of execution environment, so that each implementation of an ML task/model runs in an isolated environment with all its dependencies configured.

As a concrete example, model.yaml in ❶ specifies the requirements of the MAR model (as a KNN-based classifier). In particular, the `environment` property describes which is the execution environment (*docker* in this case) and how it will be invoked for training and test. The configuration of the environment is actually done in the corresponding Dockerfile.

In addition to Docker-based execution environments, ModelXGlue also provides support for Python-based execution environments through virtual environments[6] which is the case of AURORA (specified in ❷). In particular, the user specifies a requirements.txt files with the required Python packages and versions which are used to create the proper execution environment. Then, the concrete commands to execute *train* and *test* Python scripts must be given in the `environment` section of the YAML specification.

4.4.4 Agnosticity from the provenance of ML models

The framework permits to execute ML models independently from their origin. To this end, one approach is to wrap an existing model as a ModelXGlue component as we have done for MAR (e.g., see the item ❶ in Fig. 4) which includes, among other elements, a description of the model (metadata, its execution requirements, transformation pipeline, etc.). Alternatively, an existing approach can be reproduced by building the model from scratch (e.g., we have reimplemented AURORA using scikit-learn, a more modern approach) and encapsulating it into a component as specified in ❷ .

4.4.5 Extensibility

The framework prioritizes extensibility by decoupling the definition of an ML model designed for a specific task from its configuration in a specific benchmark (i.e., it separates ML models from its usage for running a set of experiments in a benchmark). As a result, each MDE task or model is implemented externally to the framework, but it specifies the necessary requirements for executing the task and its execution pipeline.

The framework serves as an interpreter that invokes the appropriate pipeline steps and transfers data through the pipeline while implementing the transformations requested by the model specification. This process is facilitated by the use of metadata files as for instance model.yaml in ❷ . In particular, in the ML model configuration it is possible to

[6] https://docs.python.org/3/library/venv.html.

apply transformations to the dataset. As a concrete example, let us suppose that we want to replicate some of the experiments in [36] in which different textual representations to encode meta-models are tried. The `transform` property of model.yaml in ❷ contains the configuration for a feed-forward neural network which uses a textual representation (based on 2-gram) and a TF-IDF vectorization. To achieve this, we use the ability of ModelXGlue to enhance the execution pipeline with custom transformations. The first transformation is executed in a Docker container which runs a Java program that performs the conversion by loading models in XMI and generating the textual representation. Then, a built-in vectorization transformation based on TF-IDF is applied. Using this approach, it is possible to integrate in the framework ML models with heterogeneous execution requirements for their pipelines.

To configure a set of experiments the user writes configuration files to point to the concrete ML models that she wants to run, plus information about concrete values of the hyperparameters to be used (see ❸). For instance, the `hyper` property of the same model specifies the execution of FFNN using concrete parameters. The framework automatically tries to find the best combination of hyperparameters by iterating over the different combinations.

4.4.6 Automation

The framework attempts to automate as much as possible the execution and configuration of the benchmarks. To this end, we rely on the configuration files presented above to automatically run experiments following this information. In practice, running a benchmark boils down to carrying out the following steps:

1. Implement new ML models that one wants to test for a specific task.
2. Pick up other ML models build by third parties to compare.
3. Configure the benchmark by establishing the hyperparameters of the selected ML models using configuration files as the ones shown in ❶ and ❷ .
4. Run the configured benchmarks using the command `modelxglue` as ❻ in Fig. 4
5. Inspect the results. They are saved as JSON files for ease of processing. For instance, Listing 1 shows an excerpt of the results obtained for a FFNN for model classification using a k-fold evaluation strategy. The mean_all_scores key contains, for each hyperparameter, the average of the results (for each configured metric) obtained in the 10 folds. Then, the results_best_hyperparameter simply stores the best result together with the actual value of the best hyperparameter. Finally, the configuration with which the model has been executed is also stored as metadata. In practice, we have developed some tooling to generate reports from sets of result files.

Listing 1 Excerpt of the results of a FFNN for model classification using k-fold.

```
{
  "results": {
    ...
    "mean_all_scores": {
      "{\"hidden_layer_sizes\": [50]}": [
        0.8258944354256854,
        0.8766834050693448
      ],
      "{\"hidden_layer_sizes\": [100]}": [
        0.8244323279377627,
        0.8786322333811574
      ], ...
    },
    "results_best_hyperparameter": {
      "score_folds_mean": [
        0.8293492965367966,
        0.8796102343376375
      ],
      "hyperparameter": {
        "hidden_layer_sizes": [
          150
        ]
      }
    }
  },
  "configuration": {
    "model": { ... },
    "hyperparameters": { ... },
    "dataset": { ... },
    "task": { ... }
  }
}
```

5 Catalog of ML/MDE tasks and datasets in ModelXGlue

The ModelXGlue framework is designed to support various MDE tasks by providing a configurable pipeline that allows users to perform custom transformations and apply different ML models in dedicated execution environments. The long-term objective is to establish a comprehensive catalog of ML/MDE tasks and associated benchmarks, covering a broad range of scenarios commonly encountered in the literature. In this section, we present the list of tasks and datasets for which we have developed benchmarks in the current version of ModelXGlue.

5.1 Datasets

As discussed earlier, ModelXGlue utilizes the HuggingFace datasets library to load modeling datasets, and it provides a common format for describing them. Therefore, the framework can support any dataset that is available on the HuggingFace hub. Presently, we have converted and uploaded two MDE datasets, namely Ecore-555 and ModelSet, to the HuggingFace hub. However, there are several

other relevant datasets that ModelXGlue could support. In the following sections, we discuss these datasets in detail.

Ecore-555.[7] This dataset was released by Babur [5] and contains 555 Ecore meta-models mined from GitHub in April 2017. They were manually labeled with their domains. Particularly, each meta-model belongs to one of these nine categories: bibliography, conference management, bug/issue tracker, build systems, document/office products, requirement/use case, database/sql, state machines, and petri nets.

The ModelSet dataset.[8] López et al. [29] considered a subset of the Ecore and UML models collected by the MAR search engine [30, 31] and labeled them. As a result, ModelSet was released in 2021. It contains 5,466 Ecore meta-models and 5,120 UML models labeled with its category as the main label (similarly to the Ecore-555 dataset but with more classes) plus additional secondary labels of interest.

The MAR dataset.[9] The MAR search engine [30, 31] has crawled and analyzed more than 500,000 models of different types. Particularly, the model repository managed by MAR is composed by Ecore models, UML models, BPMN models, Archimate models, to name a few. These models mainly come from GitHub, GenMyModel, and the AtlanMod Zoo.

The Lindholmen Dataset. The LindholmenDataset contains about 93,000 UML models [46] in different formats, like images, XMI and proprietary formats. The dataset is not labeled. The main shortcoming to use this dataset in our framework is that it needs to be processed to filter out models that are not compatible with standard modeling formats.

5.2 ML tasks

In this section, we describe three MDE tasks which can be addressed using ML algorithms, and for which we have already created reference benchmarks. The tasks are *model classification*, *model clustering* and *modeling assistance*. In particular, for each task, we provide a description, the related datasets, and possible evaluation metrics.

5.2.1 Model classification

Description. This task is intended to assign one (or more) descriptive labels to a given model. Typically, the label provides some semantic meaning to the model. This task has proved useful to facilitate the navigation of large model repositories by users. In particular, it has been used to implement faceted search in the MAR search engine [29].

Datasets. The dataset needs to be labeled with at least one label per model. Thus, the only datasets that could be considered for this task are the Ecore-555 and the ModelSet datasets.

Evaluation metrics. A good metric to assess the performance of the ML models is the balanced accuracy. It is a modification of the traditional accuracy to handle unbalanced datasets. In particular, given a category c, its recall is computed as:

$$\text{Recall}_c = \frac{\text{Corrected identified models of the category } c}{\text{Number of samples labeled with } c}.$$

The balanced accuracy is then computed as:

$$\text{balanced accuracy} = \text{AVG}_{c=1}^{C}\text{Recall}_c.$$

This metric is an adequate choice for ModelSet and Ecore-555 as both are highly unbalanced [32].

5.2.2 Model clustering

Description. The goal of the model clustering task is to identify groups of related models within a large set of models. These groups are intended to contain semantically similar models. This task is useful in scenarios which require having an overview of a given dataset of models. This has been used in the MDEForge repository as a way to organize and explore meta-models [10].

Dataset. This is an unsupervised learning technique and, therefore, the dataset does not need to be labeled. Thus, ML clustering algorithms could be applied to all the datasets previously presented. However, if the dataset contains labels, they can be used as ground truth and perform a more systematic evaluation.

Evaluation metrics. There are several ways of evaluating the output clusters. If we have access to the ground truth labels, we can consider metrics such as V-measure [47], Rand index [43], and NMI [53], among others. Otherwise, if there are no labels, one should consider to use metrics like silhouette coefficient [48] or Calinski-Harabasz Index [13] to evaluate the quality of the output clusters. Scikit-learn offers a wide range of clustering evaluation metrics[10] that can be easily integrated in our framework.

5.2.3 Modeling assistance

Description. This task consists of recommending relevant modeling concepts given an input context. A context is typically a set of model elements related to a given model element

[7] https://huggingface.co/datasets/antolin/ecore_555

[8] https://huggingface.co/datasets/antolin/modelset

[9] https://mar-search.org/

[10] https://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation

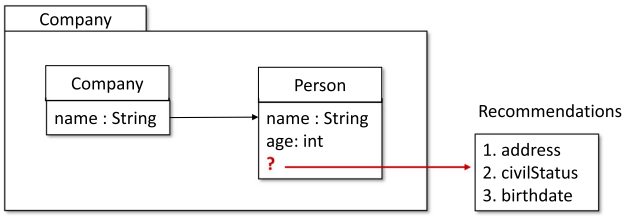

Fig. 5 Illustration of the feature name recommendation task.

that is being edited. ML models trained to solve this problem could easily be integrated in modeling environments to build autocompletion systems. It is important to note that the description of this task is very general and groups more specific tasks. For instance, in the Ecore meta-model domain, one can consider the task of recommending new EClasses for a given meta-model or recommending features for an existing EClass.

To better illustrate the modeling assistance task, let us consider the *feature name recommendation* problem for Ecore meta-models, which is shown in Fig. 5. Let us suppose that a modeler is building an Ecore model which represents companies and their employees. The EPackage is named *Company* and the developer is currently editing the *Person* EClass, adding new attributes. At this point, the user (or the system automatically) may invoke a recommender system to get a list of possible attributes for the class that it is currently being edited. In this case, the context could be the package name, the class name and the names of the attributes already added. With this information, the recommender system provides the user with a ranked list of potentially relevant attribute names.

Dataset. This task does not require a labeled dataset of models. Thus, all the datasets presented in the previous section can support the training of ML models to perform the modeling assistance task. The only requisite is that the dataset needs to be transformed to generate a training dataset by removing parts of the model (e.g., a feature) so that the model learns to predict them.

Evaluation metrics. A test sample can be constructed by removing a concept from a model and using the name of the removed concept as the ground truth. The recommendation system will take as input the model without the concept and will recommend a ranked list of k potential names, $\mathcal{S}_k = [s_1, \ldots, s_k]$. To evaluate the quality of the output list, we consider two metrics. One is the RR@k (Reciprocal Rank) which is computed as:

$$\text{RR@}k = \frac{1}{|\text{position of the ground truth within } \mathcal{S}_k|}.$$

This metric takes into account the position of the ground truth inside the ranked list and belongs to the interval [0, 1] (the higher the better). Given a set of test samples, the average of the reciprocal ranks are computed obtaining the MRR@k (mean reciprocal rank).

The other metric considered is the Recall@k. It is the fraction of the relevant concepts that are successfully retrieved. In this case, since there is only one relevant element, the metric will be 1 if the ground truth belong to the ranked list and 0 otherwise. Given a set of test samples, the average of the recalls is taken. In this situation (since there is only one relevant element per test sample), this average is also called Success Rate@k [20]. Note that this metric is less restrictive than the MRR as the position within the suggestion list is not taken into account.

6 Evaluation

In this section, we report the results of the evaluation of ModelXGlue. In particular, we aim to stress ModelXGlue with different benchmarking scenarios in order to experiment with its capabilities. To this end, we have used ModelXGlue to implement benchmarks for the three tasks described in the previous section. In this section, we will discuss the ML models we have chosen for each task. These models were sourced from previous works and have been incorporated into ModelXGlue. We will also evaluate whether ModelXGlue has the capability to support these ML models. Lastly, we will use ModelXGlue to conduct benchmarks and provide comprehensive reports on the outcomes.

6.1 ML models

To carry out the evaluation, we have selected several state-of-the-art ML approaches to solve three tasks in the context of MDE: model classification, model clustering, and feature name recommendation.

In Table 3, we have summarized the main features of the ML models we benchmarked. We have focused on the provenance of each model, whether it was used in its original form, fine-tuned, or built from scratch. Additionally, we have included information on the technology used to implement the model, and how it is made available (e.g., in the form of source code, a library or plain commands). Also, some models have constraints with respect to the kind of software models they support. It is important to consider these factors when designing benchmarks. Thus, the selection of the approaches shown in Table 3 has been done with the aim of applying ModelXGlue under different configurations based on the four dimensions shown in the table.

Table 3 Main features of the ML models adapted and benchmarked with ModelXGlue

	Provenance	Model implementation	Model availability	Constraints
Classification				
Scikit-learn models	[32]	Python / Scikit-learn	Source code	-
Lucene	[49]	Java	Source code	-
MAR	[30,31]	Java	Library	-
GNN	[32]	Python / PyTorch Geometric	Source code	-
Clustering				
SAMOS	[8]	R	Source code	-
Agglomerative clustering	From scratch	Python / Scikit-learn	N/A	-
K-MEANS	From scratch	Python / Scikit-learn	N/A	-
Recommendation				
MemoRec	[20]	Java	Command line	Minimum one attribute per class
EcoreBERT	[54]	Python / HuggingFace[PyTorch]	Source code + shell scripts	Maximum 15 classes in a single package
EcoreBERT fine-tuned	Fine-tune of [54]	Python / HuggingFace[PyTorch]	Source code + shell scripts	Maximum 15 classes in a single package
KNN-based approach	From scratch	Python	N/A	-

6.1.1 ML models for model classification

In order to establish a benchmark, we utilize the ML models outlined and provided in [32]. These models can be grouped into three distinct types.

- *Scikit-learn models*: We exploit the different models used in scikit-learn, as detailed in [32]. These models include the simple neural network (FFNN), support vector machine (SVM), k-nearest neighbor model (KNN), and Bayesian models (CNB, GNB, and MNB). Depending on the model's compatibility, we use either TF-IDF or GloVe vectorizations to feed these models.
- *Model search engines*: In the experiments, we also run a k-nearest neighbor model using off-the-shelf model search engines. In particular, we consider the use of the Lucene search engine [49] and the MAR search engine [30, 31].
- *Graph Neural Network (GNN)*: Finally, we consider the GNN model that has been used in [32] to perform the model classification task.

6.1.2 ML models for model clustering

Given a dataset of models, we aim to find meaningful semantic clusters inside that dataset. To tackle this task, several ML models are considered:

- *SAMOS* [8]: It is a tool that helps with model analytics and management. Its primary feature is hierarchical clustering, which allows for better organization and understanding of models. SAMOS works by mapping input models to vectors by extracting features such as unigrams, n-grams, graphs, or trees. These vectors are then used to compute a term-frequency-based vector space model (VSM) and perform hierarchical clustering.
- *Agglomerative clustering* and *K-means*: We consider the agglomerative clustering and k-means implementation of scikit-learn. As input for these clustering algorithms, TF-IDF or GloVe vectorizations are used.

6.1.3 ML models for feature recommendation

We consider an instance of the model assistance task, i.e., the feature name recommendation for Ecore meta-models. Given an Ecore meta-model, one feature (i.e., one EStructuralFeature) from some EClass is removed. This task aims to predict the removed feature name given the rest of the meta-model under development. To this end, the following ML models are compared:

- *MemoRec* [20]: It is an approach conceived to exploit collaborative filtering strategies to recommend valuable entities related to the meta-model under construction. The system leverages a graph representation to encode the relationship among meta-models artifacts. The recommendation engine is based on a collaborative filtering technique and can recommend suitable classes if the context is a package or structural features if the context is a class.
- *EcoreBERT* [54]: It is a RoBERTa model pre-trained on all the Ecore meta-models of the MAR search engine. The Ecore meta-models are first transformed into trees, then flattened into a string to be read by the RoBERTa model. We adapt the original model proposed in [54] for this task

by masking the feature name, the feature type, and merging the recommendations of EAttributes and EReferences.
- *EcoreBERT fine-tuned*: We have made a slight modification to the EcoreBERT input representation. Specifically, we have removed the distinction between EReferences and EAttributes. This change, combined with the task of predicting feature names and types, allows us to fine-tune EcoreBERT.
- *KNN-based approach*: This is a ML model that we have implemented from scratch. It uses as context the name of the EClass, the EPackage name and the names of the other class features. Then, we map these names to vectors using GloVe. Finally, the vectors are averaged, obtaining one vector representing the context. In the training phase, each vector has one ground truth associated (i.e., the removed feature name), and they are stored in a *kd*-tree with a pointer to the ground truth. In the testing phase, the context vector is obtained from the test sample and the *kd*-tree is queried to retrieve the most similar context vectors of the training data. The feature names associated with these vectors are the recommendations.

6.2 Building benchmarks with MODELXGLUE

As previously discussed, MODELXGLUE aims at facilitating the benchmarking of ML models applied to MDE tasks. Therefore, we are interested in analyzing to what extent the features provided by MODELXGLUE have the expressive power required for integrating existing ML models and creating new ones for the tasks described above. The process of integrating a single ML/MDE approach into MODELXGLUE has two main activities:

1. *Build transformations*: The dataset should be transformed according to the input expected by the target ML model. The transformations may need to be applied in the training phase, in the test phase or both.
2. *Integrate the ML model used by the approach under analysis*. It involves defining a concrete execution environment for the training and testing functions of the ML model, according to the original implementation technology.

Table 4 shows the transformations that we have implemented to support the ML models summarized in the previous section. In particular, the first column shows the models considered in the evaluation. The second column indicates the dataset format that is used as input of the pipeline. When it comes to classification and clustering, only one transformation is used. However, for recommendation, multiple transformations are applied in a sequence.

In the following, we give a detailed description of the developed transformations and their integration and adaptation processes for each model and task type.

6.2.1 Transformation functions

The transformations that we have implemented for the evaluation are described below.
Dump XMI to disk: This transformation takes the models as XMI strings and dumps them to disk as files. This is particularly helpful when the ML model or the next transformation requires the dataset to be on the disk, such as in Lucene, SAMOS, MemoRec, and other similar applications. To facilitate this process, we have integrated it as a built-in function in MODELXGLUE.
Vectorization: This transformation takes documents as input and produces numeric vectors using GloVe embeddings or a TF-IDF approach. We also have implemented this transformation as a built-in function inside MODELXGLUE.
Filter dataset: This transformation takes Ecore models as input and filters them according to the number of classes. It is applied for the recommendation task to respect the restrictions of the corresponding approaches and make them comparable. It is implemented as a Java function since it requires parsing the Ecore model. Thus, it runs in a Docker container.
Generate recommendation dataset: This transformation generates an augmented version of the input dataset. Each feature of each Ecore meta-model is removed, and a new version of the Ecore meta-model is constructed, annotating the name of the modified class and adding the removed feature as the target label. This is a key transformation for the benchmark since it modifies the original dataset to generate a new dataset which represents the actual task: predicting a feature name given a context (i.e., the class from which the feature has been removed). This function is implemented in Java and runs in a Docker container.
Model2tree: This transformation (implemented by the authors of EcoreBERT [54]) receives an Ecore meta-model as input and transforms it into a tree serialized as a string, which can be used, e.g., to feed the EcoreBERT model. It is composed of two functions, a Java function and a Python function, and runs in a Docker container.
Model2tree adapted: This transformation is a modified version of the previous one. The main difference is that the output tree does not distinguish between the EAttributes and EReferences.
Context generator for KNN: This transformation consists of a Java function that extracts all the feature names and their contexts for the KNN approach (i.e., the EPackage, EClass and EStructuralFeature names). This transformation runs in Docker container.

Table 5 provides an overview of the execution environments and implementation languages used for each dataset

Table 4 Dataset transformations for each ML model described in Table 3

	Dataset Format	Train Dataset transformations	Test Dataset transformations
Classification			
Scikit-learn models	text	Vectorisation	Same as the training dataset
Lucene	xmi	Dump XMI to disk	Same as the training dataset
MAR	xmi	-	Same as the training dataset
GNN	graph	-	Same as the training dataset
Clustering			
SAMOS	xmi	Dump XMI to disk	Same as the training dataset
Agglomerative clustering	text	Vectorisation	Same as the training dataset
K-MEANS	text	Vectorisation	Same as the training dataset
Recommendation			
MemoRec	xmi	Filter dataset Dump XMI to disk	Filter dataset Generate recommendation dataset Dump XMI to disk
EcoreBERT	xmi	-	Filter dataset Generate recommendation dataset Dump XMI to disk Model2tree
EcoreBERT fine-tuned	xmi	Filter dataset Generate recommendation dataset Dump XMI to disk Model2tree adapted	Same as the training dataset
KNN-based approach	xmi	Filter dataset Context generator for KNN Vectorisation	Filter dataset Generate recommendation dataset Context generator for KNN Vectorisation

Table 5 Execution environments and implementation languages for each dataset transformation

Transformation	Execution environment	Implementation language
Dump XMI to disk	built-in	Python
Vectorisation	built-in	Python
Filter dataset	docker	Java
Generate recommendation dataset	docker	Java
Model2tree	docker	Java & Python
Model2tree adapted	docker	Java & Python
Context generator for KNN	docker	Java

Table 6 Transformation functions. The lines of code (LOC) without blank lines are reported

Transformation	Java	Docker	Python
Filter dataset	128	5	-
Generate recommendation dataset	194	5	-
Model2tree	105	9	52
Model2tree adapted	105	9	42
Context generator for KNN	228	5	-

transformation. In addition, Table 6 presents the lines of code for each non-built-in dataset transformation, which can give an indication of the implementation effort required.

6.2.2 Integration of ML models

We have built various ML models as ModelXGlue components, which are summarized in Table 7. We have chosen a specific *execution environment* for each model based on its original implementation technology. Additionally, the *training approach* indicates whether the model needs to be trained from scratch, downloaded because it is pre-trained, or fine-tuned for this task. To integrate each ML model, we have followed different strategies, including *adapting* the original source code with modifications, *reimplementing* the approach based on the corresponding research paper, or *wrapping* a complete implementation using a scripting language. In the following section, we provide detailed information on how we integrated the ML models and the required effort for the three different MDE tasks.

ML models for model classification. Table 8 shows the classification ML models adapted to be ModelXGlue components and the corresponding lines of code required. The details involved in the creation of the models are the following:

- *Scikit-learn models*. The adaptation of the Python-based models, which were originally implemented using Scikit-learn, was easy. We have just moved the code and execution to a separate Python virtual environment. As shown in Table 8, the first five models (FFNN, CNB, GNB, MNB and SVM) have only 20 Python LOC, and all of them have the same structure since the only thing that changes is the name of the Scikit-learn model to use.
- *Lucene*. In the case of the Lucene approach, we had to reimplement most of it because it was not originally con-

Table 7 Execution environments, traning, and replication approaches for each model described in Table 3

	Model execution environment	Training approach	Replication approach
Classification			
Scikit-learn models	venv	Retrain	Adapt
Lucene	docker	Retrain	Reimplement
MAR	docker	Retrain	Wrapping (JRuby)
GNN	venv	Retrain	Adapt
Clustering			
SAMOS	docker	Retrain	Adapt
Agglomerative clustering	venv	Retrain	N/A
K-MEANS	venv	Retrain	N/A
Recommendation			
MemoRec	docker	Retrain	Wrapping (Bash)
EcoreBERT	venv	Download	Adapt
EcoreBERT fine-tuned	venv	Fine-tune	Adapt
KNN-based approach	venv	Retrain	N/A

ceived as a reusable library or program. This is because the approach has a non-negligible amount of Java LOCs. Furthermore, as it is implemented in Java, a Docker container was used as an execution environment.
- *MAR*. It was possible to use it as a Java library and therefore we have used JRuby to create a script that interacts with the library and implements the *k*-nearest neighbor.
- *GNN*. This model was originally implemented using PyTorch. In this case, the adaptation involved copy-pasting the original training algorithm and making sure that the dependencies (for PyTorch and PyTorch-Geometric) were properly configured.

Table 8 Classification models. The lines of code (LOC) without blank lines are reported

Model	JRuby	Java	Docker	Python
FFNN	-	-	-	20
CNB	-	-	-	20
GNB	-	-	-	20
MNB	-	-	-	20
SVM	-	-	-	20
GNN	-	-	-	172
Lucene	-	394	6	-
MAR	98	-	21	-

ML models for model clustering. Table 9 shows the clustering models that have been adapted to be components of ModelXGlue, along with the corresponding lines of code. Here are some details about how these models were created:

- *SAMOS*. The effort to include SAMOS in the framework involved mainly the installation of all the requirements in the environment and the adaptation of the main functions to be invoked by ModelXGlue. It is important to note that the installation of the requirements was not trivial as SAMOS combines R and Java. Thus, the use of a Docker container guarantees that dependency installation is now automated.
- *Agglomerative clustering* and *K-MEANS*. The adaptation of these models was similar to the Scikit-learn classification models.

ML models for feature recommendation. Table 10 shows the ML models that are used for the task of recommending feature names. These models have been modified to be ModelXGlue components, and their respective lines of code are also included. Further details about the creation of these models are provided below.

- *MemoRec*. This recommendation tool is made available as a set of command line programs which perform the preprocessing and the inference. Thus, a Docker container and shell scripts have been used to interact with the model.
- *EcoreBERT*. The integration of this model was moderately easy as we only had to implement the inference function. Since the model can be loaded through the HuggingFace, we used a Python virtual environment to run the inference.
- *EcoreBERT fine-tuned*. The main difficulty of integrating this model was to build the fine-tuning function. The training and inference functions was run in a Python virtual environment.
- *KNN-based approach*. The implementation of this model was relatively straightforward using the *kd*-tree implementation of Scikit-learn. The main difficulty was the implementation, in Java, of the transformation to generate the context vectors.

Altogether, we have been able to adapt and integrate existing ML models into ModelXGlue with a moderate effort.

Table 9 Clustering models. The lines of code (LOC) without blank lines are reported

Model	R	Java	Docker	Python
SAMOS	20	315	15	-
Agglomeraive clustering	-	-	-	16
K-MEANS	-	-	-	16

Table 10 Recommendation models. The lines of code (LOC) without blank lines are reported

Model	Java	Docker	Python	Shell
MemoRec	-	11	-	41
EcoreBERT	-	-	117	-
EcoreBERT fine-tuned	-	-	183	-
KNN-based approach	-	-	49	-

6.3 Running benchmarks

Once all the selected ML models were built as ModelXGlue components and organized in the form of three benchmarks, we used the framework to run the benchmarks in order to evaluate the performance of the models and compare them. In the remaining of this section, we present and discuss the results.

6.3.1 Model classification

Datasets and evaluation metrics. We have considered the ModelSet dataset without duplicates and the Ecore-555 dataset. We report two evaluation metrics: the balanced accuracy and the traditional accuracy.

Dataset split. As it is done in previous works [32, 36], we run a 10−fold evaluation strategy. First, the dataset is split into 10 independent test sets. At each time step, one fold is used as test set and the other nine folds as train set. Thus, each model is trained 10 times, and the average of the evaluation metrics is used. The balanced accuracy is used to select the best hyperparameter, and only the results of the best combination of hyperparameters are reported. To run the experiments, we removed categories in both datasets with less than 10 elements to ensure that, at least, there is one representative of each category in all the test folds.

Results. Tables 11 and 12 show the results with ModelSet (without duplicates) and the results using Ecore-555 (with duplicates).

The performance results using the Ecore-555 dataset are much higher because of two reasons: *i)* the number of categories is much lower than ModelSet which makes the classification problem easier, and *ii)* we have considered the Ecore-555 dataset with duplication which may cause an overlap between the train and test sets. We employ the Ecore-555 dataset because of two reasons. Firstly, to show that our framework actually allows the use of different datasets seamlessly. And secondly, to use the same dataset as [36] and notice that the results obtained here are similar to the ones reported in [36] for the FFNN with TF-IDF model.

Table 11 Classification task with the ModelSet dataset (without duplicates) using k-fold as sampling strategy

Model	Features	Balanced Accuracy	Accuracy
FFNN	TF-IDF	0.829	0.880
SVM	TF-IDF	0.816	0.873
SVM	GloVe	0.792	0.848
LUCENE	-	0.785	0.834
GNN	graph	0.784	0.844
FFNN	GloVe	0.778	0.837
MAR	-	0.773	0.823
KNN	TF-IDF	0.769	0.816
CNB	TF-IDF	0.725	0.822
MNB	TF-IDF	0.722	0.828
KNN	W2V	0.721	0.769
GNB	TF-IDF	0.615	0.680
SVM	kernel	0.602	0.716

Table 12 Classification task with the Ecore-555 dataset (with duplicates) using k-fold as sampling strategy

Model	Features	Balanced Accuracy	Accuracy
FFNN	TF-IDF	0.963	0.971
SVM	TF-IDF	0.948	0.960
FFNN	GloVe	0.941	0.953
LUCENE	-	0.939	0.945
GNN	graph	0.938	0.947
SVM	GloVe	0.933	0.943
CNB	TF-IDF	0.920	0.951
MNB	TF-IDF	0.920	0.947
MAR	-	0.919	0.925
KNN	TF-IDF	0.913	0.929
GNB	TF-IDF	0.900	0.900
KNN	W2V	0.898	0.914
SVM	kernel	0.818	0.869

The results obtained in Table 11 are very similar to the ones obtained in [32], showing that ModelXGlue can support the replicability of ML/MDE approaches. To evaluate this fact numerically, we apply the Kendall rank correlation coefficient to both rankings (Table 11 in this paper and Table 5 in [32]). The coefficient obtained is 0.92. Thus, there is an strong agreement between the induced ranks. The main difference found is that the results for the GNN are slightly different, probably because of the data splits being slightly different and a different seed is used (i.e., GNNs are not deterministic).

6.3.2 Model clustering

Datasets and evaluation metrics. For the sake of shortness, as target dataset, we have only considered the ModelSet dataset without duplicates. We use the labels as the ground

Table 13 Clustering with ModelSet

Model	Features	V-score
K-MEANS	TF-IDF	0.691
Agglomeraive clustering	TF-IDF	0.657
K-MEANS	GloVe	0.607
Agglomeraive clustering	GloVe	0.525
SAMOS	-	0.481

Table 14 Feature recommendation task (without duplicates)

Model	MRR	Success rate
EcoreBERT fine-tuned	0.400	0.508
EcoreBERT	0.366	0.504
MemoRec	0.358	0.419
KNN-based approach	0.328	0.399

truth clusters and report the V-measure score. This metric is defined as the harmonic mean between homogeneity and completeness. A clustering assignment satisfies the homogeneity property if each cluster contains only members of a single class, and it satisfies the completeness property if all members of a given class are assigned to the same cluster. The V-measure scores are between 0.0 and 1.0, and the higher the better [47].

Dataset split. To perform the clustering experiments, we do not split the dataset and run directly the clustering algorithms over the full dataset.

Results. The clustering model V-scores are displayed in Table 13. It is evident that our implemented simple approaches perform better than SAMOS.

6.3.3 Feature name recommendation

Datasets and evaluation metrics. For the sake of brevity, as the target dataset, we have only considered ModelSet without duplicates. As evaluation metrics, we use *mean reciprocal rank* and *success rate* with a fixed cut-off value $k = 5$ of the size of the recommendation list.

Dataset split. We use a train/validation/test strategy to perform the recommendation experiments. The rationale is that training transformers networks is costly and sampling alternatives, such as $k-$fold, require to train the models several times.

Results. Table 14 shows the MRR@5 and the Success rate@5 of the considered models. The models that achieve the best performance are the EcoreBERT models. This is because EcoreBERT was pre-trained using the MAR dataset, and ModelSet is a subset of MAR. Thus, it is likely that EcoreBERT has already seen the test models in the pre-training phase. Moreover, the fine-tuning procedure helps EcoreBERT to achieve greater performance. Notably, the gap between the fine-tuning and pre-trained versions is more prominent in the MRR. An added value of this work is that we have adapted the EcoreBERT's work to be fine-tuned, and we have achieved the best performance in this recommendation task.

The worst model is the KNN-based approach. This is not surprising as the KNN model only has access to a small portion of the full context (EClass and EPackage names and feature names). The results obtained for EcoreBERT and MemoRec are aligned with the results obtained in their original works.

6.4 Comparison of ModelXGlue with ML benchmarks

In this section, we compare the ModelXGlue benchmark with three other relevant benchmarking frameworks, considering the criteria outlined in Sect 4. The comparison is presented in Table 15. The rows encompass such criteria, including additional information like the programming language in which the benchmarking framework is implemented, the supported ML models, and the kind of input data that the framework provides support to. Meanwhile, the columns represent the four benchmarking frameworks selected for this comparative analysis.

CodeXGlue [33] is a widely used benchmark for code intelligence. That is, its aim is to compare the performance of NLP deep learning models in several code tasks. It includes 14 datasets for 10 diversified code intelligence tasks. The framework is written in Python and only PyTorch models can be run on top of it. Furthermore, it is not possible to add another type of ML model that is not PyTorch neural networks (e.g., KNN-based approaches, SVM, etc.) without changing the source code.

AMLB [25] is a benchmark for AutoML systems. AMLB internally trains multiple models with various hyperparameters, utilizing heuristics to construct or select the optimal ML model. Given the plethora of AutoML frameworks with diverse dependencies and programming languages, AMLB facilitates the management of environments when executing these systems. However, it lacks support for this functionality during data preprocessing, as the target frameworks typically accommodate standardized input data formats (e.g., parquet, arff, or csv) or Python objects like numpy arrays and pandas dataframes.

The Droid framework has been proposed by Almonte et al. [3, 4]. Its goal is to achieve the automation of the configuration, evaluation and synthesis of recommender systems for modeling languages. The key difference of Droid with respect to CodeXGlue, ModelXGlue and AMLB is that is Droid is specifically designed for recommender systems for modeling languages. In this respect, Droid is not exactly a

Table 15 Comparison of different ML benchmarks

Requirement	ModelXGlue	CodeXGlue [33]	AMLB [25]	Droid [4]
Management of different datasets	Yes	Yes	Yes	Yes
Management of different metrics	Yes	Yes	Yes	Yes
Exec. environments (for ML models)	Yes	No	Yes	No
Exec. environments (for pre-processing)	Yes	No	No	No
Agnosticity from the provenance of ML models	Yes	No	Yes	No
Extensibility	Yes	No	Yes	Yes
Automation	Yes	Yes	Yes	Yes
Language	Python	Python	Python	Java
Target ML models	Any	PyTorch neural networks	AutoML systems	Any
Target data	Software models	Code (token sequences)	Images and tabular data	Software models

benchmarking framework (although it can be used for this purpose) but a Low-Code platform in which the developer is guided through different choices to build the best recommender system which fits all his requirements. Thus, Droid allows the user to perform evaluations on the evaluated recommender systems.

Therefore, the most similar framework to ModelXGlue is AMLB, but their target is different. AMLB is focused on images and tabular data while we focus on software models, which requires that ModelXGlue also provides support for different execution environments for the preprocessing phase. Moreover, AMLB is designed specifically for AutoML models, whereas ModelXGlue support any kind of ML model.

6.5 Discussion

In this section, we provide a critical assessment of ModelXGlue based on our experience using it to build the benchmarks presented in the previous section. The discussion is organized according to the requirements presented in Sect. 4.

Management of different datasets. We have been able to seamlessly use two different datasets in the experiments since we rely on a common format. We began using only ModelSet, but then we added the Ecore-555 dataset with little extra effort.

Management of different metrics. We have implemented the most common evaluation metrics, so that the experiments consistently use the same metrics for all the models. As a concrete example, in the original work of MemoRec [20], success rate is used as the main metric, whereas in EcoreBERT [54] MRR is used. This makes both works barely comparable. In our experiments, we have been able to compare both works with the same metric (actually we use both precision and MRR).

A limitation of our current implementation is that the metrics are hard-coded in the framework. In the future, we would like to provide some mechanism for users to provide their own metrics (e.g., in the form of libraries as in HuggingFace evaluate[11]).

Management of execution environments. The task of integrating approaches in ModelXGlue is facilitated by the possibility of encapsulating the runtime environment (i.e., the dependencies) in a Docker container or a Python virtual environment. In particular, Docker should be used for approaches which rely on runtimes different from Python. As a concrete example, loading and manipulating EMF models in ecosystems different from Java is often very difficult or even impossible.[12] Thus, the fact that we can run Java projects as part of the pipeline and then combine the results with other parts of the pipeline implemented in Python is a crucial feature to support ML for MDE.

At the same time, the fact that each pipeline runs in an isolated environment also has several advantages. One of them is that there are no version conflicts. Another advantage is that the dependencies required for a specific model are only installed when it is executed. This is important when the models need CUDA support (for GPU). For instance, our GNN implementation is based on PyTorch and PyTorch-Geometric, which some machines may not support. Therefore, when benchmarking, one may decide which models to execute according to the features of the target machine. Another advantage is that if the execution of one of the benchmark models fails, the rest of the benchmark may proceed, and partial results can be obtained.

All of these allowed us to execute the performed experiments in different machines automatically, without the need of any particular setup process.

However, a number of MDE tools are based on the EMF, and their implementations are often tight to the Eclipse platform. This means that they are only executable from within the Eclipse IDE. This is the case of SAMOS, for which we had to adapt the code to make it deployable through Maven manually. Therefore, we want to investigate ways to use Eclipse-plugins directly in our framework.

[11] https://huggingface.co/docs/evaluate/index

[12] There are solutions to use EMF in dynamic languages (PyEcore or RubyTL [19]), but they are not fully compatible.

Another point of improvement for ModelXGlue is to have debugging facilities. Since the models run in isolated environments it is sometimes complicated to use standard debugging facilities (e.g., the IDE debugger).

Agnosticity from the provenance of ML models. The ModelXGlue framework enables the use of various machine learning models in a consistent way, regardless of their source. Whether one is integrating a third-party model or creating his or her own, the use of Docker or a Python environment provides a convenient way to package dependencies and wrap the models in a component directly executable by ModelXGlue. In particular, we have integrated 15 models of different types and provenances, thus covering the three main scenarios that can be found in practice: wrapping an existing components (11 models), building from scratch (3 models) and fine-tuning and wrapping an existing model (1 model). Moreover, we have shown that the results obtained running the models using ModelXGlue are aligned with the ones obtained by the original authors using ad-hoc execution environments.

Extensibility. The framework implementation is completely decoupled from the benchmarks that we have developed, that is, they actually belong to different code repositories. This permits creating and evolving benchmarks without the need of modifying the source code of ModelXGlue.

Another advantage of the extensibility of the framework, is that it is possible to build a catalog of reusable data transformations for ML/MDE benchmarks. The use of a common set of transformations is sometimes important for ensuring that the ML models can be compared faithfully. In our case, we needed to make sure that the recommendation models consistently applied the same transformations. Moreover, the fact that ModelXGlue promotes the separation between data transformations and the actual ML models may help researchers to organize their approaches into a set of well-defined modules. In the future, we expect catalogs of useful data transformations which could be seamlessly reused to create new approaches.

The main issue with the extensible pipeline is that each stage defines input and output data which needs to be moved around. In our current implementation, we use the file system to serialize data and ML models. This may be inefficient and also requires to be able to serialize the ML models to disk. While we have not yet found actual issues with this approach, this may need to be tackled to scale the system to larger datasets.

In addition, the fact that the pipelines are specified as sequence of transformations is a conceptually clean abstraction, but in some cases it may cause some inefficiencies in particular when both the training and test pipelines need to load the same element twice (e.g., GloVe embeddings). In other words, ModelXGlue does not have a way to share common execution environments and globally load shared resources.

Automation. The benchmark specification is fully automated through configuration files. A benchmark is composed of three types of configuration files: datasets, tasks, and ML models. In particular, once the configuration file for the dataset and a task have been created, it is shared for the execution of each ML model under analysis. Moreover, the configuration of a model consists of specifying its hyperparameters. This approach allows us to quickly iterate over the benchmark by adding new models, changing hyperparameters, and testing results, supporting the typical iterative process shown in Fig. 1.

By developing the three reference benchmarks, we have systematically evaluated state-of-the-art approaches for model classification, model clustering, and feature name recommendation. In practice, other researchers could continue extending the benchmarks with better models. We also made tools to analyze the results and generate summaries, including the tables in this paper, which were automatically created.

7 Conclusion and future work

Machine learning (ML) is a rapidly growing field with a wide range of applications in various domains, including MDE. However, selecting the appropriate ML tool for a specific task can be challenging, as different tools may have distinct strengths and weaknesses. To address this issue, we introduced the ModelXGlue framework, which facilitates benchmarking ML models specifically created for MDE tasks. The framework can manage different datasets, metrics, and execution environments, automating the benchmarking process and simplifying comparisons while reducing the burden of managing various artifacts involved in the process. We have built an initial catalog of MDE/ML benchmarks and demonstrated their execution within the framework.

In addition to the presented work, future plans for ModelXGlue include the development of domain-specific languages and supporting tools that simplify the specification of input configurations needed for executing benchmarks (e.g., type checking inputs and outputs in the pipeline). Moreover, we would like to address some of the shortcomings and inefficiencies of the current prototype, as discussed previously. Finally, we are interested in fostering the usage of ModelXGlue within the MDE community to grow the existing catalog of benchmarks.

Acknowledgements This work has been partially supported by the following grants and projects:
• Grant TED2021-129381B-C22 (SATORI project) funded by MCIN/AEI/10.13039/501100011033 and NextGenerationEU/PRTR,
• Grant PID2022-140109NB-I00 (AIM project) funded by MCIN/AEI/10.13039/501100011033 and FEDER/UE,
• Grant CNS2022-135578 (LowSheets project) funded by MICIU/AEI/10.13039/501100011033 and NextGenerationEU/PRTR.

• EMELIOT national research project, which has been funded by the MUR under the PRIN 2020 program (Contract 2020W3A5FY).
• European Union–NextGenerationEU through the Italian Ministry of University and Research, Projects PRIN 2022 PNRR *"FRINGE: context-aware FaiRness engineerING in complex software systEms"* grant n. P2022553SL.
• The Italian "PRIN 2022" project TRex-SE: *"Trustworthy Recommenders for Software Engineers,"* grant n. 2022LKJWHC.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Adhikari, B., Rapos, E.J., Stephan, M.: Simima: a virtual simulink intelligent modeling assistant: Simulink intelligent modeling assistance through machine learning and model clones. Softw. Syst. Model. pp. 1–28 (2023)
2. Allamanis, M.: The adverse effects of code duplication in machine learning models of code. In: Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, pp. 143–153 (2019)
3. Almonte, L., Cantador, I., Guerra, E., de Lara, J.: Towards automating the construction of recommender systems for low-code development platforms. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 1–10 (2020)
4. Almonte, L., Guerra, E., Cantador, I., De Lara, J.: Building recommenders for modelling languages with droid. In: 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–4 (2022)
5. Babur, Ö.: A Labeled Ecore Metamodel Dataset for Domain Clustering. https://doi.org/10.5281/zenodo.2585456
6. Babur, Ö., Chaudron, M.R., Cleophas, L., Ruscio, D.D., Kolovos, D.: Preface to the first international workshop on analytics and mining of model repositories. In: 2018 MODELS Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS, MODELS-WS 2018, pp. 778–779. CEUR-WS. org (2018)
7. Babur, Ö., Cleophas, L.: Using n-grams for the automated clustering of structural models. In: International Conference on Current Trends in Theory and Practice of Informatics, pp. 510–524. Springer (2017)
8. Babur, Ö., Cleophas, L., van den Brand, M.: Samos-a framework for model analytics and management. Sci. Comput. Program. **223**, 102877 (2022)
9. Babur, Ö., Cleophas, L., Brand, M.v.d.: Hierarchical clustering of metamodels for comparative analysis and visualization. In: European conference on modelling foundations and applications, pp. 3–18. Springer (2016)
10. Basciani, F., Rocco, J.D., Ruscio, D.D., Iovino, L., Pierantonio, A.: Automated clustering of metamodel repositories. In: International conference on advanced information systems engineering, pp. 342–358. Springer (2016)
11. Burgueño, L., Clarisó, R., Gérard, S., Li, S., Cabot, J.: An nlp-based architecture for the autocompletion of partial domain models. In: International Conference on Advanced Information Systems Engineering, pp. 91–106. Springer (2021)
12. Cabot, J., Clarisó, R., Brambilla, M., Gérard, S.: Cognifying Model-Driven Software Engineering, pp. 154–160 (2018). https://doi.org/10.1007/978-3-319-74730-9_13
13. Caliński, T., Harabasz, J.: A dendrite method for cluster analysis. Commun. Stat.-Theory Methods **3**(1), 1–27 (1974)
14. Capuano, T., Sahraoui, H., Frenay, B., Vanderose, B.: Learning from code repositories to recommend model classes. J. Object Technol. **21**(3), 3 (2022)
15. Caruana, R., Niculescu-Mizil, A.: An empirical comparison of supervised learning algorithms. In: Proceedings of the 23rd international conference on Machine learning - ICML '06, p. 161–168. ACM Press, Pittsburgh, Pennsylvania (2006).https://doi.org/10.1145/1143844.1143865
16. Chaaben, M.B., Burgueño, L., Sahraoui, H.: Towards using few-shot prompt learning for automating model completion. In: 2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), pp. 7–12. IEEE (2023)
17. Chowdhury, S.A., Varghese, L.S., Mohian, S., Johnson, T.T., Csallner, C.: A curated corpus of simulink models for model-based empirical studies. In: Proceedings of the 4th International Workshop on Software Engineering for Smart Cyber-Physical Systems, pp. 45–48 (2018)
18. Clarisó, R., Cabot, J.: Applying graph kernels to model-driven engineering problems. In: Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis, pp. 1–5 (2018)
19. Cuadrado, J.S., Molina, J.G., Tortosa, M.M.: Rubytl: A practical, extensible transformation language. In: Model Driven Architecture–Foundations and Applications: Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006. Proceedings 2, pp. 158–172. Springer (2006)
20. Di Rocco, J., Di Ruscio, D., Di Sipio, C., Nguyen, P.T., Pierantonio, A.: Memorec: a recommender system for assisting modelers in specifying metamodels. Softw. Syst. Model. pp. 1–21 (2022)
21. Di Rocco, J., Di Sipio, C., Di Ruscio, D., Nguyen, P.T.: A gnn-based recommender system to assist the specification of metamodels and models. In: 2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 70–81. IEEE (2021)
22. Di Ruscio, D., Nguyen, P.T., Pierantonio, A.: Machine Learning for Managing Modeling Ecosystems: Techniques, Applications, and a Research Vision, pp. 249–279. Springer International Publishing, Cham (2023). https://doi.org/10.1007/978-3-031-36060-2_10
23. Di Sipio, C., Di Rocco, J., Di Ruscio, D., Nguyen, P.T.: Morgan: a modeling recommender system based on graph kernel. Software and Systems Modeling pp. 1–23 (2023)
24. Gérard, S., Burgueño, L., Burdusel, A., Gerard, S., Wimmer, M.: Preface to MDE Intelligence 2019: 1st Workshop on Artificial Intelligence and Model-Driven Engineering. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 168–169. IEEE, Munich, Germany (2019).https://doi.org/10.1109/MODELS-C.2019.00028. https://hal-cea.archives-ouvertes.fr/cea-02572659

25. Gijsbers, P., Bueno, M.L., Coors, S., LeDell, E., Poirier, S., Thomas, J., Bischl, B., Vanschoren, J.: Amlb: an automl benchmark. arXiv preprint arXiv:2207.12560 (2022)
26. Lin, T.Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., Zitnick, C.L.: Microsoft coco: Common objects in context. In: Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13, pp. 740–755. Springer (2014)
27. Liu, F., Li, J., Zhang, L.: Syntax and domain aware model for unsupervised program translation (arXiv:2302.03908) (2023). Accepted for publication at ICSE 2023
28. López, J.A.H., Cánovas Izquierdo, J.L., Cuadrado, J.S.: Modelset: a dataset for machine learning in model-driven engineering. Softw. Syst. Model. pp. 1–20 (2021)
29. López, J.A.H., Cánovas Izquierdo, J.L., Cuadrado, J.S.: Modelset: a dataset for machine learning in model-driven engineering. Softw. Syst. Model. pp. 1–20 (2021)
30. López, J.A.H., Cuadrado, J.S.: Mar: A structure-based search engine for models. In: Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems, pp. 57–67 (2020)
31. López, J.A.H., Cuadrado, J.S.: An efficient and scalable search engine for models. Softw. Syst. Model. pp. 1–23 (2021)
32. López, J.A.H., Rubei, R., Cuadrado, J.S., Di Ruscio, D.: Machine learning methods for model classification: a comparative study. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, pp. 165–175 (2022)
33. Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al.: Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664 (2021)
34. Madan, M., Reich, C.: Comparison of benchmarks for machine learning cloud infrastructures. Cloud Comput. **50** (2021)
35. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)
36. Nguyen, P.T., Di Rocco, J., Di Ruscio, D., Pierantonio, A., Iovino, L.: Automated classification of metamodel repositories: A machine learning approach. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 272–282. IEEE (2019)
37. Nguyen, P.T., Di Rocco, J., Iovino, L., Di Ruscio, D., Pierantonio, A.: Evaluation of a machine learning classifier for metamodels. Softw. Syst. Model. **20**(6), 1797–1821 (2021)
38. Nguyen, P.T., Di Ruscio, D., Pierantonio, A., Di Rocco, J., Iovino, L.: Convolutional neural networks for enhanced classification mechanisms of metamodels. J. Syst. Softw. **172**, 110860 (2021)
39. Nguyen, P.T., Rocco, J.D., Sipio, C.D., Ruscio, D.D., Penta, M.D.: Recommending API function calls and code snippets to support software development. IEEE Trans. Software Eng. **48**(7), 2417–2438 (2022). https://doi.org/10.1109/TSE.2021.3059907
40. Olson, R.S., La Cava, W., Orzechowski, P., Urbanowicz, R.J., Moore, J.H.: Pmlb: a large benchmark suite for machine learning evaluation and comparison. BioData Mining **10**(1), 36 (2017). https://doi.org/10.1186/s13040-017-0154-4
41. Ozkaya, I.: The next frontier in software development: Ai-augmented software development processes. IEEE Softw. **40**(4), 4–9 (2023). https://doi.org/10.1109/MS.2023.3278056
42. Pennington, J., Socher, R., Manning, C.D.: Glove: Global vectors for word representation. In: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), pp. 1532–1543 (2014)
43. Rand, W.M.: Objective criteria for the evaluation of clustering methods. J. Am. Stat. Assoc. **66**(336), 846–850 (1971)
44. Reddi, V.J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., et al.: Mlperf inference benchmark. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pp. 446–459. IEEE (2020)
45. Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., Ma, S.: Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297 (2020)
46. Robles, G., Ho-Quang, T., Hebig, R., Chaudron, M.R., Fernandez, M.A.: An extensive dataset of uml models in github. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 519–522. IEEE (2017)
47. Rosenberg, A., Hirschberg, J.: V-measure: A conditional entropy-based external cluster evaluation measure. In: Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL), pp. 410–420 (2007)
48. Rousseeuw, P.J.: Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. J. Comput. Appl. Math. **20**, 53–65 (1987)
49. Rubei, R., Di Rocco, J., Di Ruscio, D., Nguyen, P.T., Pierantonio, A.: A lightweight approach for the automated classification and clustering of metamodels. In: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 477–482. IEEE (2021)
50. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al.: Imagenet large scale visual recognition challenge. Int. J. Comput. Vision **115**, 211–252 (2015)
51. Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Sarro, F.: A survey on machine learning techniques for source code analysis. CoRR (2021). arXiv:2110.09610
52. Stallkamp, J., Schlipsing, M., Salmen, J., Igel, C.: The german traffic sign recognition benchmark: a multi-class classification competition. In: The 2011 international joint conference on neural networks, pp. 1453–1460. IEEE (2011)
53. Vinh, N.X., Epps, J., Bailey, J.: Information theoretic measures for clusterings comparison: is a correction for chance necessary? In: Proceedings of the 26th annual international conference on machine learning, pp. 1073–1080 (2009)
54. Weyssow, M., Sahraoui, H., Syriani, E.: Recommending metamodel concepts during modeling activities with pre-trained language models. Software and Systems Modeling pp. 1–19 (2022)
55. Yellin, D.M.: The premature obituary of programming. Commun. ACM **66**(2), 41–44 (2023). https://doi.org/10.1145/3555367

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.