ModelMate: A recommender for textual modeling languages based on pre-trained language models

Carlos Durá Costa
carlos.durac@um.es
Universidad de Murcia
Spain

José Antonio Hernández López
jose.antonio.hernandez.lopez@liu.se
Linköping University
Linköping, Sweden

Jesús Sánchez Cuadrado
jesusc@um.es
Universidad de Murcia
Spain

ABSTRACT

Current DSL environments lack smart editing facilities intended to enhance modeler productivity and cannot keep pace of current developments of integrated development environments based on AI. In this paper, we propose an approach to address this shortcoming through a recommender system specifically tailored for textual DSLs based on the fine-tuning of pre-trained language models. We identify three main tasks: identifier suggestion, line completion, and block completion, which we implement over the same fine-tuned model and we propose a workflow to apply these tasks to any textual DSL. We have evaluated our approach with different pre-trained models for three DSLs: Emfatic, Xtext and a DSL to specify domain entities, showing that the system performs well and provides accurate suggestions. We compare it against existing approaches in the feature name recommendation task showing that our system outperforms the alternatives. Moreover, we evaluate the inference time of our approach obtaining low latencies, which makes the system adequate for live assistance. Finally, we contribute a concrete recommender, named ModelMate, which implements the training, evaluation and inference steps of the workflow as well as providing integration into Eclipse-based textual editors.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; • **Computing methodologies → Machine learning**.

KEYWORDS

Recommendation, Meta-modeling, Model-Driven Engineering, Machine learning

ACM Reference Format:
Carlos Durá Costa, José Antonio Hernández López, and Jesús Sánchez Cuadrado. 2024. ModelMate: A recommender for textual modeling languages based on pre-trained language models. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3640310.3674089

MODELS '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0504-5/24/09
https://doi.org/10.1145/3640310.3674089

1 INTRODUCTION

The construction of software models of different kind (Ecore meta-models, UML models, DSL programs, etc.) is a central activity in Model-Driven Engineering (MDE) approaches. Therefore, improving modeling productivity is crucial for the successful implementation of a MDE approach [19]. However, this is not an easy task, since modeling effectively requires expertise at various levels. On the conceptual side, it is important to be able to choose the appropriate concepts (e.g., in a Petri net meta-model, typical concepts that may be modeled are Net, Place, Transition, etc.); at the syntax level, the modeler needs to construct models following the rules of the modeling language; and at the technological level, the modeler needs to use a concrete tool which should provide features to faciltate the different modeling tasks.

In this context, building recommender systems is a well-known approach to achieve the objective of improving developer productivity when using a tool or a language. Thus, integrated development environments (IDEs) make pervasive use of recommender systems of different types [24] and the trend is to use the latest developments in Artificial Intelligence (AI) and Machine Learning (ML) to boost such recommender systems. In the modeling setting, some recommender systems for modeling tools have been proposed and there is recently a renewed interest on them due to the availability of ML techniques [26]. However, the current state-of-the-art of recommender systems for modeling tools present a number of limitations. First, current recommender systems are built on top of small amounts of data (e.g., small datasets for training and/or evaluation) which limits their applicability. Also, most recommenders are conceived specifically for one kind of language (typically Ecore class diagrams) and it is not clear how to adapt them to other modeling languages. With respect to its integration into modeling environments, they have insufficient performance (or it has not been tested) to be used interactively and they do not integrate well with textual languages since they assume that the model is well-formed in memory. Moreover, the proposed models are not released in a way that could be easily deployed (e.g., a plug-in for a modeling language). In general, it can be claimed that current approaches are not mature enough to be used in real scenarios.

To address these shortcomings, in this paper we propose an approach to build recommender systems for textual modeling languages using pre-trained language models (PLMs). We identify three editing tasks which are supported by our system, namely identifier suggestion, line completion and fragment completion. We describe a general end-to-end process which can be applied to any textual modeling language, and we discuss the different elements involved in the construction of such system in practice, including

how to handle dataset construction, tokenization, inference, evaluation, etc. We apply this process to three particular DSLs: Ecore with Emfatic notation, Xtext and a custom DSL to describe domain entities. We fine-tune several open source, pre-trained models using the ModelSet dataset [13] and the MAR dataset [14, 15] to build models for these DSLs. We evaluate our proposal from several dimensions. First, we analyse the performance of each model in each of the identified tasks. Then, using the best models we compare against three other approaches in the literature, EcoreBert [31], MemoRec [6] and a variant of [4] by setting setup a benchmark for the "feature name" recommendation task in Ecore models. Moreover, we compare against the option of prompting OpenAI GPT3.5 obtaining comparable results. Finally, we assess whether our system is fast enough to be used in practice. Altogether, we make the following contributions:

- We propose an approach to build smart recommendation facilities for textual modeling languages based on pre-trained language models and identify three main editing tasks.
- We apply our approach on three DSLs using multiple state-of-the-art pre-trained models, achieving good results.
- We evaluate and compare our approach against previous proposals showing that it generalizes better.
- We contribute a tool called ModelMate[1], offering features for building textual datasets and for fine-tuning and evaluating models. Additionally, ModelMate can be integrated with Eclipse-based editors through a plugin and a web server.

Outline. The rest of the paper is organized as follows. In Section 2, we provide the background information. In Section 3, we present our approach to build recommender systems for textual modeling languages and Section 4 describes how ModelMate is used in practice. Section 5 reports the evaluation results. Section 6 discusses the related work and Section 7 concludes.

2 BACKGROUND

2.1 Pre-trained language models

Pre-trained language models (PLMs) are neural networks that follow the transformer architecture [28], and they are trained with a large corpora to perform a language modeling task *i.e.,* predicting a word given its surrounding context. Depending on their architecture, PLMs can be classified into three categories: encoder-only, decoder-only, and encoder-decoder [18]. In this paper, we only deal with decoder-only models as they are naturally designed to be used in the autocompletion tasks. This kind of PLMs are pre-trained in an autoregressive way, that is, predicting the next token given the preceding ones. Examples of decoder-only models include the GPT models [3, 23], CodeGPT [16], CodeGen models [20], among others.

Two methods exist for tuning a PLM to perform a specific task [18]. In-context learning [9] involves introducing a prompt into the PLM in order to steer it to solve the target task. However, this method is effective only for very large PLMs [30] (with several billions of parameters). The alternative method is fine-tuning [18], which entails further training the PLM using a dataset specific to the task at hand. Fine-tuning is particularly effective for smaller PLMs and is commonly employed when a task-specific dataset is accessible. The core of ModelMate involves fine-tuning small PLMs.

2.2 Recommendations in textual DSLs

To create models or meta-models, a modeler needs to use some kind of editor, which can typically be graphical (e.g., arrows and boxes or a tree editor) or textual. Thus, to enhance modeler productivity it is necessary to provide recommendation facilities integrated in such editors. In particular, the autocompletion facility is a key feature in modern IDEs. However, this is lacking in the case of modeling environments.

The techniques proposed in the literature to build recommender systems for non-textual DSLs cannot be easily adapted to be used for textual DSLs. The main reason is that such techniques typically assume that they have access to the complete model in memory, which is not the case in this scenario. In a textual editor when the user begins to write a new construct (e.g., class I), the DSL parser alone is not able to process the complete text and obtain an in-memory representation of the model. Although error correction techniques could be used to mitigate this issue, they are typically not available for DSLs. Instead, for textual languages we can take advantage of current advances in language models, which are able to understand text sequences of new languages if they are fine-tuned, which is the focus of our approach.

2.2.1 Xtext, Emfatic and Domain entities. In this work we apply our approach to three DSLs which showcase different features and challenges. These languages are Emfatic[2], Xtext[3] and a variant of the Domain entities DSL available in the Xtext distribution[4].

Xtext is a DSL to build textual DSLs. It is based on grammatical rules annotated with the meta-model elements that are manipulated (generated or assigned). The following listing shows an excerpt of the Xtext grammar to specify the syntax of the language to describe domain entities (i.e., the "Domain-Model example").

```
DomainModel: elements+=AbstractElement* ;
AbstractElement: PackageDeclaration | Entity ;
PackageDeclaration: 'package' name=QualifiedName
  '{' elements+=AbstractElement* '}';
Entity: 'entity' name=ValidID ('extends' superType=Entity)?
  '{' features+=Feature* '}';
```

Emfatic is a textual DSL to specify Ecore models. It is built on top of the Ecore meta-model and it provides transformations from .ecore files to the Emfatic syntax, and vice versa. The following listing is an excerpt of a Petri net meta-model specified with Emfatic. Emfatic distinguishes between attributes, references and containment references with the attr, ref and val keywords, respectively.

```
package petrinet;
class PetriNet {
  attr String[0..1] name;
  val Node[*] nodes;
  ...
}
class Node {
  ref PetriNet[1] net;
}
```

[1] ModelMate is available at https://github.com/models-lab/model-mate

[2] https://eclipse.dev/emfatic/

[3] https://eclipse.dev/Xtext/

[4] The "Domain-Model example" discussed in https://eclipse.dev/Xtext/documentation/102_domainmodelwalkthrough.html

The Domain entities DSL syntax ressembles Emfatic (as shown in the Xtext grammar shown above). The main difference is that features are defined with the syntax name : type and that it allows operations with a Java-like body.

2.2.2 Recommendation tasks in textual DSLs. A key aspect is to identify which are the recommendation tasks that make sense for a given DSL. In this work, we consider three recommendation tasks, which are illustrated in more detail in Fig. 1 using Emfatic.

(1) *Fragment completion.* Given the previous context (i.e., the text before the cursor), the system completes the current fragment by adding a sequence of tokens up to a certain place. In the example, the user has written class PetriNet { and the system proposes completing the rest of the class with several attributes and references. This is the most general type of completion, since it can be triggered anytime in the editing process.
(2) *Line completion.* In this case, the user is at the beginning of a line and the system generates a full line. In the example, the user is creating a class and the system generates a complete attribute which includes its type, cardinality and name. To be more effective, the editor should use this mode for language constructs which are typically described in a single line (i.e., when a line represent a complete construct in the target language). For instance, Emfatic features (attributes and references) are normally written in a line.
(3) *Identifier suggestion.* This task aims at helping the user in selecting proper identifiers (i.e., a naming task) and to select existing names to reference existing concepts. In the example, the system generates a list of proposed names for the type of the reference. Since the Place class has already been defined, it appears first in the list. Other suggestions like Transition have not yet been defined by the user, who may select them and later create the corresponding class.

These tasks can be mapped to other DSLs. For instance, in Xtext, the main construct is the grammatical rule. The user might be interested in obtaining recommendations about rule names (e.g., Entity), about property names or references to other rules (e.g., name and QualifiedName in name=QualifiedName) and about string literals (e.g., 'package'). In all cases the system should be able to recommend a specific token depending on the cursor position. Another kind of recommendation would be to predict a complete fragment. For instance, after writing Entity, the user might be interested on getting a completion for the rest of the rule (up to ';') or just a complete line.

In the following section we describe our approach to support these tasks in textual DSLs.

3 APPROACH

Our approach is based on the use of pre-trained language models to build a recommender system for textual modeling languages. First, we address the question of the availability of datasets. Afterwards, we explain the method used to fine-tune a PLM to understand the textual syntax of a new DSL. Then, we describe how the inference is performed and finally, we explain how to evaluate these models.

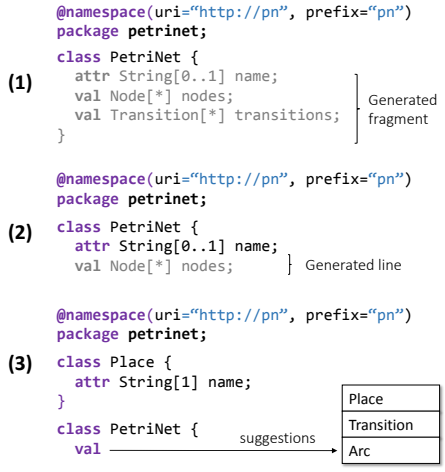

Figure 1: Recommendation types exemplified with Emfatic. Gray text is the generated completion. (1) Fragment completion, (2) Line completion, (3) Identifier suggestion.

3.1 Dataset availability scenarios

A key aspect of our approach is that, for each different DSL, a new model needs to be fine-tuned to learn the DSL syntax. However, the scope of DSLs is typically small since they are focused on a specific community of users whose size is way less than the communities of general purpose languages. Therefore, the amount of training data available is expected to be small as well. Moreover, when a new DSL is created it suffers from the well-known cold-start problem of recommender systems [24]: since the DSL has not been used yet, there are no users and therefore there is no training data to build a first version of the recommender system. The consequence is that most DSLs cannot profit from smart assistants from the beginning. In this context, given a DSL, we consider three different scenarios depending on the availability of datasets.

Textual dataset available. When there is a dataset of textual files already available, our approach is applicable almost directly without any intermediate steps (see Sect. 3.2). We evaluate this scenario using Xtext, which is popular enough to allow us to find an adequate amount of public files (i.e., in GitHub) to create a dataset.

Serialized dataset available. In this case, we do not have a textual dataset but there exists a dataset of serialized models (e.g., in XMI) for the target DSL. Thus, the dataset is fully compatible with the target textual syntax which is based on exactly the dataset metamodel. For example, notations like Emfatic or OclInEcore simply add a textual syntax to an existing modeling language (i.e., Ecore). In this case, we only need to use the serializing facilities of the tool, if available, or build a simple code generator.

No dataset available. This is the most frequent scenario when a new DSL is being developed. To tackle this issue we propose to use a dataset of another modeling language, which is semantically compatible, and convert the models in the dataset to the required textual syntax. For instance, if one is developing a DSL for representing some kinds of business processes, it would be possible to

use datasets of UML Activity Diagrams or BPMN. Then, the requirement is to be able to construct a model-to-text transformation from the dataset to the DSL notation.

In this work we build one recommender for each of these cases. For the first scenario, we have built a recommender system for Xtext. For the second case, we have chosen Emfatic. For the third case, we reuse the Domain entities DSL from the Xtext tutorials.

3.2 Methodology

Given a modeling language with an associated textual notation, the methodology that we use to build a recommender system has the following steps (which are also illustrated in Fig. 2).

Dataset selection. A dataset with enough examples in textual syntax needs to be provided. Depending on the DSL, one of the three strategies described in the previous section needs to be followed.

Duplicate removal. It is important to remove duplicates from the dataset not only to avoid evaluation bias [2] but also to improve the training of the PLMs [12]. To this end, we implement a variant of the duplication detection algorithm by Allamanis [2] adapted to software models.

Text generation and tokenization. For every model, we need to create its textual representation. We employ the tokenizer of the DSL to extract the sequence of tokens from the original text (or a code generator if we use plain models) and add three special tokens to this sequence: <s> (start of sequence), </s> (end of sequence), and <EOL> (end of line). For example, the following listing illustrates the tokenized serialization (enhanced with the special tokens) of an Emfatic model. We perform this preprocessing for two primary reasons. First, similar preprocessing steps are conducted in popular code understanding benchmarks like CodeXGlue [16]. Second, from an implementation perspective, this format greatly facilitates running the inference and evaluation of PLMs. For instance, the generation of lines concludes upon encountering the <EOL> token.

```
<s> package petrinet ; <EOL> class PetriNet { <EOL>
attr String [ 0 ] name ; ... } </s>
```

Model selection. ModelMate is agnostic to the selected language model and the fine-tuning technique. However, in this work, we only consider models with less than one billion parameters. It is important to take into account that, in the same way as the development cost of a DSL needs to be modest [17], the training and inference cost of its associated recommender system probably also needs to be modest. The choice of relatively small pre-trained models have several implications like the possiblity of training them in commodity GPUs and the possibility of getting fast recommendations. The details of the chosen models are explained in Sect. 3.3.

Training. For training the models, we adopt a train-validation setup, leveraging the validation set to implement early stopping and mitigate potential overfitting of the PLM. Following each epoch, we assess the model's performance using the validation split. Should the model exhibit deteriorating performance on this split, training is stopped before reaching the maximum number of epochs (iterations over the training dataset) and the best model checkpoint is saved.

Evaluation. The final step is to evaluate the performance of the chosen models in order to understand if they perform well for the target DSL, for instance, to determine if additional training data is needed or which is the best model to be deployed. The details about how to perform this evaluation are explained in Sect. 3.5 and are applied in practice in Sect. 5.

3.3 Selected pre-trained language models

In our implementation we employ PLMs available in HuggingFace Hub, but conceptually it is possible to use any PLM. In particular, Table 1 presents the PLMs considered in this study, which include:

- GPT2 family [23]. Released by OpenAI, these models were trained using 40GB of text data (WebText). For this study, we focus on three checkpoints: GPT2 (124M parameters), GPT2-M (355M parameters), and GPT2-L (774M parameters).
- DistilGPT2 [25]. HuggingFace released this compressed version of GPT2, employing 82M parameters. It was pre-trained through knowledge distillation, with GPT2 serving as the teacher model. The dataset used in the training is an open-source version of the WebText corpus.
- CodeParrot-small[5]. This 110M parameter model was pre-trained using an extensive corpus extracted from GitHub, encompassing data from nine programming languages.
- Codegen family [20]. Salesforce released a series of models of 350M parameters. Codegen-nl was pre-trained on The Pile [10] containing both natural language and code. Codegen-multi was initialized with the Codegen-nl checkpoint and trained on a diverse corpus containing multiple programming languages from GitHub. Finally, Codegen-mono was initialized with the Codegen-multi checkpoint and trained using a Python corpus.

PLM	Parameters	Pre-training data
GPT2 [23]	124M	WebText [23]
GPT2-M [23]	355M	WebText [23]
GPT2-L [23]	774M	WebText [23]
DiltilGPT2 [25]	82M	OpenWebText [22]
CodeParrot-small-multi	110M	CodeParrot dataset
Codegen-nl [20]	350M	The Pile [10]
Codegen-multi [20]	350M	BigQuery [20]
Codegen-mono [20]	350M	BigPython [20]

Table 1: PLMs considered in ModelMate.

3.4 Inference

Given a fine-tuned model and an input preceding context, the new tokens are generated until a specific condition is reached. For example, if we are targeting Emfatic and we intend to recommend fragments (as depicted in the first case of Fig.1), token generation continues until the token } is obtained. For generating new lines (as in the second case of Fig.1), the termination token is the special <EOL> token. Finally, if the objective is to generate the next token, generation stops when a blank space is produced.

[5] https://huggingface.co/codeparrot/codeparrot-small-multi

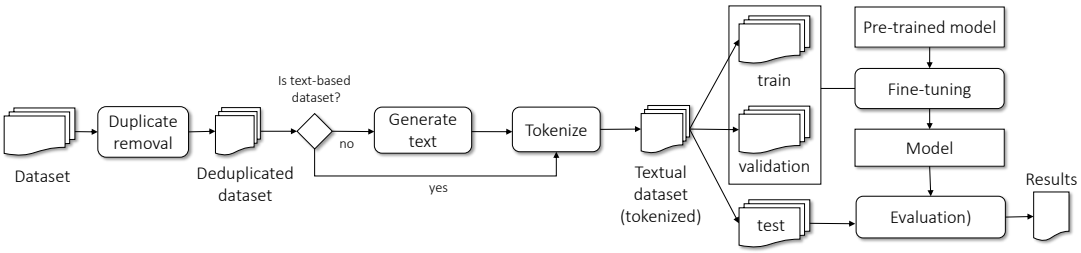

Figure 2: Training and evaluation workflow of ModelMate.

To provide more than one recommendation and to ensure deterministic inference, we use the beam search generation procedure [11], which approximately outputs the top-k most likely sequences according to the fine-tuned model given the input context. For token inference, we set $k = 5$ to generate five suggestions (which also allows us to compute the MRR over these suggestions). For lines and blocks, we use $k = 2$ because in practice we only require one suggestion and in our initial experiments we found no significant improvement with higher values of k (increasing k often results in the same first line/block). Moreover, a lower k increases inference speed.

3.5 Task evaluation procedure

The training of our models was performed using the complete models in tokenized, textual syntax. However, in order to evaluate all three of our tasks, we need to preprocess the original test set to create other tests sets suitable for evaluating each concrete task. In all cases the new test set is composed by a list of pairs formed by *context* and *expected*, which contain the input to the model and the expected ground truth, respectively. Next, we show how the test set is derived in order to evaluate the system in each task.

Fragment completion. For each DSL we need to identify the block delimiters. In Emfatic and the domain entities DSL they are { and }. In Xtext we use : as the block start (to have the rule name as context) and ; to indicate the end of the rule. Thus, for every block start token (e.g., {) in each model, we store as the context the prefix up to such token and the expected fragment consists of the sequence of tokens until the first block end token (e.g., }).

Line completion. For every <EOL> token, the context is the text before that (including the <EOL>) and the ground truth is the following text until the next <EOL>.

Identifier suggestion. We need to match in the text the identifiers that are of interest for each DSL. For instance, in Emfatic we are interested in identifiers that follow some keywords, like class <class-name> and attr <type-name>. Thus, we capture the context before the identifier (which includes the keyword) and the identifier is the ground truth. We also support regular expression matching to handle more complex cases like name : type in which the identification of the name depends on the appearence of : and type as tokens.

Next token prediction. We also consider an additional evaluation to have a general, rough view of the performance of the training model, independent of the concrete tasks. For every token, we capture the text up to a given token (excluded) as the context and that token is the expected predicted value. We ignore special tokens like <EOL>.

The new test sets are very large since they include all possible instances of each case for each sample in the test set. For instance, in the case of next token completion this means that the amount of tests is now the number of tokens per sample, in the case of line completion the number of lines, etc. Using the full versions of the datasets would make running evaluation impractical. Therefore, we introduce a parameter to sample the number of instances of each kind to reduce the size.

3.6 Evaluation metrics

Now we briefly present the evaluation metrics for each of the evaluation tasks that we have defined in the previous version.

Fragment completion metric. To evaluate the accuracy of generated fragments we use the smoothed BLEU score [21, 27]. This metric is widely used in machine translation and text generation tasks. It belongs to the interval $[0, 100]$ (the greater the better) and measures the similarity between the predicted text and the expected text by computing the n-gram overlapping. More precisely, BLEU is calculated as

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right),$$

where BP is a brevity penalty that lowers the score for very short completions, w_n is the weight given for the n-gram precision (it is normally set to $\frac{1}{N}$, $N = 4$), and p_n is the n-gram precision.

Line completion metric. In this case, we follow the same approach as in CodeXGlue [16] and we use Levenshtein distance and Exact Match (EM) as evaluation metrics. Levenshtein distance is an edit similarity metric that is calculated based on the minimum number of insertions, deletions, and substitutions needed to transform one string into another. It ranges between 0 and 100, and the higher this value is, the more similar the strings are. EM calculates the percentage of exact matches between predicted outputs and expected outputs.

The rationale for using different metrics for fragment and line completion is the following. A line is typically a small piece of text (e.g., 32 tokens at most in our experiments), whereas a fragment is normally composed of several lines. Therefore, measuring EM and edit similarity is more representative for line completion.

Identifier suggestion. In this task, we are generating 5 different predictions, so in order to take into account the position in which

the correct prediction is, we use the *MRR@k* [16] metric with k=5. Given a list of k suggestions ordered from highest to lowest relevance, if the expected suggestion (the ground truth) is in position i, it gets scored $\frac{1}{i}$, otherwise 0. The *MRR@k* is calculated as the average of all these scores for all the instances in the test set and the aggregated score is within the interval $[0, 100]$ (the greater the better).

Next token prediction. In this task we use the accuracy as evaluation metric, measured as the proportion of predictions matching the ground truth. This metric resembles the one used in the CodeXGlue [16] benchmark for token-level code completion.

4 MODELMATE IN PRACTICE

This section briefly describes the facilities provided by ModelMate and its implementation. To build datasets we provide two main facilities in the form of Java APIs. The first one is a simple API to detect duplicate models which is based on extracting relevant n-grams (i.e., strings) from each model in a dataset [2]. The second API allows us to implement simple model-to-text transformations which conforms to the CodeXGlue format.

For the training and evaluation of the models, it provides a Python library that allows the user to setup the training and evaluation pipeline. The library is built on top of Hydra (for configuration) and HuggingFace Transformers (for fine-tuning). Given a new DSL for which we want to train a model, the user just needs to create a configuration file for the dataset and run the training and evaluation commands.

To integrate ModelMate in textual editors we provide a dedicated web-server which loads the trained model and provides a REST API with endpoints to access the three recommendation tasks. The web server can be run locally or in a remote server with a more powerful GPU.

ModelMate is currently available for Eclipse-based editors through a plug-in. The plug-in provides built-in support for fragment completion for any textual editor by setting up a connection with the web server. Given a new DSL the only thing that needs to be done is to implement an extension point to register the file extension and to provide a tokenizer. For identifier suggestions it is needed to extend the DSL editors specifically since a special content assist processor for the DSL needs to be installed. To showcase this scenario, we have extended the Emfatic editor to suggest identifiers depending on the current cursor position.

Figure 3 shows the ModelMate plug-in for Eclipse and Emfatic. As the developer types, the plug-in sends the text before the cursor to a background thread which queries the web server. This is done asynchronously and only when the user has stopped typing for a few milliseconds. The suggested fragment is shown in ❶ and can be applied with a keywbord shortcut. In addition, by pressing Ctrl+Space, the default Eclipse content assist is shown but the identifier suggestions from ModelMate are added at the beginning (see ❷).

5 EVALUATION

In this section we evaluate our approach from several dimensions. First, we evaluate the performance of ModelMate for the three recommendation tasks with three different DSLs. Then, we compare

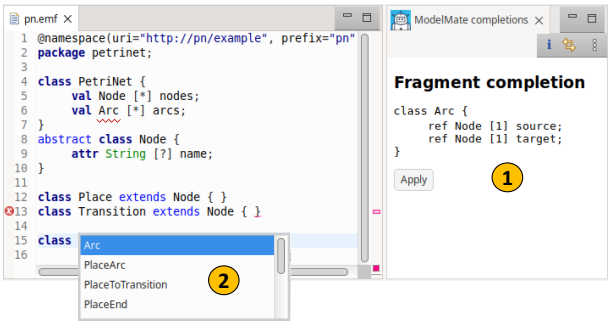

Figure 3: ModelMate plug-in for Emfatic. (1) Fragment completion suggestion. (2) Identifier suggestion.

ModelMate with other model recommenders on a common task. We also compare it against a propietary large language model (LLM), namely OpenAI GPT 3.5. Finally, we report the inference time of our approach. The ModelMate framework, including the Eclipse plugins, and the scripts and instructions to replicate the evaluation, are available at https://github.com/models-lab/model-mate.

5.1 Performance in recommendation tasks

We evaluate our approach from the point of view of the performance of the trained models with respect to the tasks presented in Section 2.2. To this end, we have fine-tuned the models described in Section 3.3 using the procedure described in Section 3.2. This evaluation corresponds to what a DSL developer would do to determine which pre-trained model to use for a given DSL and to understand the expected performance of such models.

The models have been fine-tuned in a machine with an NVIDIA RTX A5000 GPU with 24 GB of memory in about 2 hours each, on average. The main training parameters include context length, which we set to 512 tokens to make sure that all models fits in the GPU memory, the learning rate is set to 5×10^{-5} and we train up to 10 epochs.

The datasets used for training the models for each DSL are shown in Table 2. Each dataset is explained in the corresponding section.

Dataset	#Models (Raw)	#Models (Dedup)	#Models (Filtered)	Applied to
MAR/Ecore	67, 322	8, 541	8, 541	Emfatic
MAR/Xtext	5, 061	2, 762	2, 762	Xtext
ModelSet	10, 595	5, 568	1, 204	Domain DSL

Table 2: Datasets used for training the recommenders for the three DSLs.

5.1.1 Emfatic. To apply our training pipeline to Emfatic models we have used the dataset of Ecore models available in the MAR search engine [15], specifically the ones crawled from GitHub. After removing duplicates the dataset contains 8, 541 models. We have converted these models to text using a custom serializer. The reason is that the default Emfatic serializer tends to generate too many

keywords for default values which clutter the input and it is not representative of real Emfatic texts.

The results of the evaluation are shown in Table 3. We use the accuracy for next token prediction (column 2) as the primary metric to sort the table. For the *identifier* recommendation task we consider class names, super class names (after extends), feature types which can be for attributes (attr), plain references (ref) and containment references (val), and feature names (in Emfatic they go after the cardinality). For the *line* completion task, we consider the completion of full lines (e.g., the contents from one line break to the next line break). For the *fragment* completion task, we consider the completion of the body of a class (e.g., class PetriNet { block-content }).

As can be observed the values are relatively high, meaning that in many cases the system provides relevant suggestions. For attribute types the MRR is higher than for ref and val because the amount of primitive types is limited. For super class names the MRR is also high because abstract classes tend to be added at the beginning of the file (i.e., they appear in the context of the completion). For line and fragment completion the results indicate a good performance [27].

Regarding the performance of the pre-trained models, the codegen family outperforms the GPT2 family. We can also observe that in the GPT2 family, the larger the model in terms of number of parameters the better the results. However, the model that stands out is codeparrot since it is a relatively small model which is on par with the larger models.

5.1.2 Xtext. To build a dataset for Xtext we took the .xtext files available in MAR, which we deduplicated to obtain a dataset of 2.762 Xtext files. Then, we applied the Xtext tokenizer to render the final textual dataset.

For Xtext we evaluate next token prediction accuracy, line completion and fragment completion, but we do not apply identifier suggestion because Xtext has few keywords over which "anchor" the evaluation of specific tokens. Instead, Xtext rules are just a sequence of grammar elements and therefore the basic, next token prediction task is more useful in practice (see Xtext example in Section 2.2).

For the sake of brevity, we only report the results of one model of each family. The results are shown in Table 4. In this case, the results are lower than in Emfatic because in Xtext the amount of non-keyword tokens that the model must predict is very high.

5.1.3 Domain entities DSL. This is a simple language to describe domain entities in a software project. Although in the surface it is similar to Emfatic, it showcases some differences worth studying. On the one hand, the goal of the DSL is not meta-modeling, but software modeling and therefore it should be trained on a dataset containing domain models. On the other hand, class features are specified with the syntax name : type and there is no special keyword to introduce the kind of feature (attribute or reference). This means that inference becomes more difficult for features names.

The construction of the dataset for this DSL is slightly more involved. We use both Ecore and UML from the ModelSet dataset [13]. After removing duplicates, for Ecore we only keep the models labelled with *domain-model* and for UML we only used the models containing diagrams with at least 5 classes. The rationale is to make sure that the final dataset is about domain modeling rather than meta-modeling (as it is the case of the Emfatic dataset). After filtering, the final dataset only contains 1,204 models. Then, we convert these models to text using a custom serializer. Since this DSL support operations with a Java-like body, we generate default return values for the operations according to the return type (e.g., return 0 if the return type is an integer).

The results are shown in Table 5. The accuracy in next token prediction is higher than in Emfatic, possibly because the body of the operations is very repetitive (this also explains the high value of the BLEU metric). On the other hand, the MRR for entity names is close to the results for Emfatic class names. For operation names, the MRR is a bit low because the models do not follow a consistent naming convention. Regarding feature names and types, it is noteworthy that the results now show higher values for types compared to names, which is contrary to Emfatic.

5.2 Comparison with model recommenders

The goal of this evaluation is to understand whether ModelMate is able to outperform existing model recommenders in a task which could be addressed by all of them. Unfortunately, we have not found any model recommender focused on the *fragment* and *line* completion tasks. Instead, as shown in Table 9, current recommenders are focused on predicting identifiers. Therefore, we use an adaptation of our *identifier suggestion* task.

We have selected model recommenders published in the literature which can be used directly or with minor adaptations to perform the identifier recommendation task. In particular, we select three works to compare with: EcoreBert [31], MemoRec [6] and the approach in [4] (named here KNN/Glove), which we have reimplemented and adapted since the original tool is not replicable. We apply these recommenders to the task of suggesting feature names in Ecore meta-models because it is well supported by all of them. The task is as follows: given an Ecore meta-model with one feature removed, predict the name of the removed feature. This means that the tools are allowed to use the rest of the model as context.

We reused the original EcoreBert model "as is", which was trained with the MAR/Ecore dataset. Hence, we train MemoRec and KNN/-Glove with the MAR dataset to make sure that they have seen the same training data as EcoreBert. For ModelMate we used as target syntax the one of Domain entities DSL since it adapts well to the task of only predicting the feature name. We used the codeparrot model for this evaluation since it achieved nearly the best results in the previous evaluations, despite being only a third the size of the top-performing model in the previous evaluations.

We perform the evaluation with two datasets: the Ecore models from ModelSet and the Ecore models from GenMyModel available in MAR. For both datasets we apply a filtering pipeline in which we first remove duplicates, then remove models with less than 2 classes (i.e., small models) and finally remove models which are likely not using English (to perform a fair comparison with MemoRec which cannot generalize to non-english languages). The ModelSet dataset is a subset of the MAR/Github dataset and thus it is expected that all tools perform well over this dataset. However, the models provided by GenMyModel have not been seen in the training and allows us to study the generalization capabilities of the models.

		Identifier Suggestion						Line		Fragment
Model	Accuracy	class name	super name	attr. type	ref. type	val. type	feature name	EM	Edit Similarity	BLEU
codegen-mono	69.10	45.03	78.28	77.24	52.91	39.22	59.35	14.27	50.51	31.21
codegen-multi	69.07	45.19	78.33	77.73	53.45	39.03	59.36	14.37	50.81	31.32
codeparrot	68.64	44.25	77.65	77.72	50.84	37.79	58.35	13.90	48.63	30.67
gpt2-large	68.52	42.29	75.41	76.60	50.58	36.87	57.51	13.24	48.53	29.93
codegen-nl	68.02	41.55	76.31	75.75	50.21	35.87	57.23	12.06	46.04	29.11
gpt2-medium	67.65	39.97	74.24	75.91	48.34	33.66	54.83	11.63	44.36	28.39
gpt2	65.83	35.19	70.78	74.49	42.15	28.77	51.07	9.16	41.76	26.15
distil-gpt2	64.02	29.08	66.30	73.15	36.65	23.58	47.02	6.48	37.19	24.09

Table 3: Results of the evaluation of different models for Emfatic.

		Line		Fragment
Model	Accuracy	EM	Edit Similarity	BLEU
codeparrot	60.13	22.38	43.90	23.07
codegen-multi	58.75	21.41	42.02	22.81
gpt2	54.99	20.32	38.04	17.57

Table 4: Evaluation results for Xtext.

For each sample, the recommenders output up to 5 suggestions. We use as metrics the Mean Reciprocal Rank (MRR@5) and the Success Rate (SR@5), which is similar to accuracy but allowing to match any of the 5 results.

Table 6 shows the results obtained in the *feature name recommendation* task with the four approaches. For ModelSet both MemoRec and KNN/Glove obtain very good results, but ModelMate is relatively close. However, using the GenMyModel dataset (whose models have not been seen in training) the results show that ModelMate outperforms the other approaches by more than 100%. The rationale for this discrepancy in the results is that ModelMate is able to generalize, which is a key property for a recommender system.

5.3 Comparison with LLMs

Large Language Models (LLMs) are PLMs with several billions of parameters. LLMs exhibit in-context learning capabilities, which means that given some examples of the target syntax in the prompt, they may be able to generalize to other examples. Therefore, an alternative approach to our proposal is to directly use an LLM by showing some examples in the prompt.

In this part of the evaluation we want to compare the performance of an LLM in the three proposed tasks against our approach. In our case, we show in the prompt three meta-models selected randomly from the dataset in Emfatic format and the task proposed to the LLM is to complete a partial meta-model. We select OpenAI GPT-3.5-turbo-instruct which is the best model for text completion tasks at the time of writing this article[6]. We set the temperature to 0 and the number of generated completions to 1. Depending on the type of task, we post-process the result to obtain the following token, line or block respectively. Given that this evaluation involves API usage costs, we created random sample of the test set by fixing the number of samples to 1,000 in the case of fragments and lines, and to 200 for each kind of identifier completion. We also re-run ModelMate over the same sampled dataset with three of the most significant models.

Table 7 shows the results of the evaluation. As it can be observed, ModelMate and GPT 3.5 perform similarly in the identifier suggestion task. For line completion, GPT 3.5 outperforms ModelMate, whereas ModelMate surpasses GPT 3.5 in fragment completion.

5.4 Inference time

We aim at evaluating whether the performance of our approach in terms of the inference time is adequate for online assistance. This is a key aspect for the use of a recommendation system in practice. If the latency of a recommendation is large, then the system cannot be properly integrated in a modeling tool.

We choose a small model (codeparrot), a medium model (codegen) and a large model (gpt2-large). We run 1,000 predictions for each kind of task on a NVIDIA RTX A5000 GPU. Table 8 shows the average time and the standard deviation. In the identifier recommendation task, all models are typically below half a second, even though we request five recommendations. Regarding line recommendation, the latency remains consistently low, notably with codeparrot. In fragment recommendation, codeparrot still shows low latency but it increases for larger models which also have a large standard deviation. The variability can be attributed to both the diversity of fragment sizes and the quadratic time complexity of the transformer with respect to the input length.

5.5 Discussion

In this section, we discuss more in-depth several aspects of our experiments, limitations and the applicability of ModelMate.

Overal performance assessment. We obtain consistently good results in the three DSLs evaluated. For the basic next token prediction task, the accuracy ranges between 60% and 70% for Emfatic and Xtext depending on the PLM. In the case of the Domain entities DSL is higher but it might be biased due to repetitive patterns in the body of the operations.

Impact of DSL syntax on performance. For the identifier suggestion task we also obtain consistent results (in Emfatic and Domain entities). The main difference is the fact that in Emfatic the MRR for reference (i.e., ref and val) types is lower whereas for feature names is higher. This is due to the fact that a feature in Emfatic

[6]For many basic tasks, the difference between GPT-4 and GPT-3.5 is not significant, as noted in https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo.

		Identifier Suggestion				Line		Fragment
Model	Accuracy	entity name	op. name	feature name	feature type	EM	Edit Similarity	BLEU
codegen-multi	77.39	40.86	19.49	33.35	78.16	36.99	59.21	71.78
codeparrot	76.31	41.38	20.07	31.39	79.20	34.38	55.92	72.19
gpt2	69.58	28.34	11.87	20.14	70.53	22.21	43.57	65.08

Table 5: Evaluation results for the Domain entities DSL

	ModelSet		MAR/GenMyModel	
	MRR@5	SR@5	MRR@5	SR@5
ModelMate	0.52	0.64	**0.18**	**0.25**
EcoreBert	0.34	0.47	0.09	0.13
MemoRec	**0.72**	**0.73**	0.10	0.12
KNN/Glove	0.70	0.75	0.06	0.08

Table 6: Comparison of ModelMate with other ML-based recommenders in the feature name recommendation task.

has the syntax e.g., `val Node[*] nodes`. Thus, to predict the type (e.g., `Node`) the model only has information about whether is `attr`, `ref` or `val`, whereas to predict the feature name the PLM has already seen the type. This shows that the syntax of the DSL is important for the performance of the recommender system and should be taken into account when designing new DSLs.

It is also interesting to analyse why the inference of some kinds of identifiers has better precision than others. In Emfatic, the type of `val` references (containment) has lower MRR than `ref`. This is because meta-modelers typically begin creating the container classes (e.g., `PetriNet`) and then add the contained classes (e.g., `Node`). This means that the context for the inference does not include the referenced type. In the case of `ref` references, it is the other way around, they typically refer to names that have already been written. A similar issue occurs with the super classes, which are typically added at the beginning of the file. This fact shows a limitation of our approach, which do not take advantage of pieces of text which might be defined below the cursor position. This requires special handling at the tokenizer and grammar level and is left for future work.

PLM selection. With respect to whether larger models perform better, we observe that the performance of the models is not only determined by the number of parameters but also by the training data and the architecture of the model. In this sense, the *codegen* family outperforms the GPT2 family in general. However, *codeparrot* is consistently performing very well in the three DSLs evaluated despite being a small model. In this respect, it is worth noting that DSLs are small scope languages. This means that the cost associated to build a DSL needs to be modest, which applies to its construction (e.g., using a language workbench) but also to the training and deployment of ML-based recommender systems. Therefore, being able to pick the proper model for a DSL is an important aspect to take into account. Our evaluation suggests that *codeparrot* can be a good alternative for DSLs since it has a low training cost, good accuracy and fast inference time.

ModelMate outperforms existing model recommenders. The comparison of ModelMate against other model recommenders has shown that some recommenders (MemoRec and KNN/Glove) may obtain very good results when the test dataset overlaps the training data. This is so because they memorize. In our case, we apply early stopping in the training to avoid overfitting and to make sure that the model generalizes. Also, it is important to take into account that the evaluation using the GenMyModel dataset can be considered a "hard problem" since there is a important distribution shift with respect to the dataset on which the models were trained. Thus, the results obtained by ModelMate can be considered good. It is also worth noting that the design of this task is oriented to non-textual editors (e.g., when a user adds a feature to a class, the graphical editor automatically sets the name). This evaluation shows that ModelMate can be adapted to non-textual editors as well and can even outperform approaches which have been tailored for this kind of editors.

Generalisation and adaptability. We do not expect major issues for ModelMate to generalise to new DSLs. For the evaluation we have purposely chose two very different DSLs: Xtext and Emfatic to show that the system is able to perform well for unrelated DSLs. In this sense, the main entry barrier to apply ModelMate could be the absence of datasets. If a good dataset for the target DSL already exists, then the construction of a recommender should be straightforward regardless of the style of DSL. We introduced the notion of "semantically compatible dataset" as a way to refer to two ideas: a) the transformation makes sense (e.g., transforming from UML to Ecore, but not from Petri Net to Ecore) and b) the actual dataset contains data that is useful for the users of the target DSL. For instance, a UML dataset probably contains only domain models and it might not be useful to train an Emfatic recommender which is expected to be used to build meta-models. In this sense, devising techniques to synthetically build datasets for DSLs (e.g., using LLMs) is a new research direction that we expect to be important in the future and would boost our approach [29, 32].

ModelMate vs. GPT3.5. With respect to the comparison with a state-of-the-art LLM, we observe that ModelMate produces results comparable to GPT-3.5. A key difference between both approaches is the size of the models, since GPT-3.5 has several orders of magnitude more parameters than the PLMs that we have considered. This is a key aspect since the cost of using an LLM is high (in terms of API costs, computational resources and data privacy). In contrast, ModelMate can be executed in modest computers with good performance and without additional costs or data privacy concerns.

Practical usage. In our experience, the integration of a new DSL in ModelMate is relatively simple. The main effort is to create a dataset for the DSL, which can be done with a code generator. To give a concrete example, the time needed to create and debug the code generator for the Entities DSL dataset was about 4 hours. However,

		Identifier Suggestion						Line		Fragment
Model	Accuracy	class name	super name	attr. type	ref. type	val. type	feature name	EM	Edit Similarity	BLEU
GPT-3.5	**80.58**	**42.00**	79.00	72.50	**58.50**	**51.5**	54.00	**22.60**	**65.80**	23.73
codegen-multi	69.24	41.23	78.93	80.49	52.21	37.71	**62.87**	14.20	48.84	**31.65**
codeparrot	68.71	41.83	**80.58**	79.78	50.23	38.60	62.72	13.10	46.54	30.87
gpt2-large	68.52	40.95	78.46	**81.35**	47.41	39.05	61.25	12.60	45.12	30.42

Table 7: Comparison of ModelMate/Emfatic against prompting GPT-3.5 on a sampled version of our test dataset (1,000 samples for tokens, lines and fragments and 200 for each kind of identifier).

	Identifier	Line	Fragment
codeparrot	84.34 ± 40.0	130.2 ± 43.8	420.0 ± 714.0
codegen-multi	232.2 ± 115.6	341.9 ± 130.0	1490.7 ± 2311.6
gpt2-large	365.4 ± 217.8	425.6 ± 191.6	1877.8 ± 2801.5

Table 8: Inference time of ModelMate/Emfatic. Average time (ms) of 1,000 samples per task ± the standard deviation.

this time is highly related to the expertise of the modeler in the considered language. Regarding the editing experience, we found it generally smooth and the suggestions are relevant, although this must be formally validated with a user study in future work. Moreover, ModelMate is robust since it is able to provide suggestions even in the presence of errors. This can be observed in Fig. 3 which shows an Emfatic snippet with a syntax error (line 13) and a semantic error (line 6, because Arc is not defined). The architecture of ModelMate makes it possible to deal with incomplete models and can provide suggestions even in the presence of errors. Moreover, our evaluation has shown that the inference time is adequate for online assistance.

Altogether, the results of the evaluation show that ModelMate is a promising approach for enhancing a DSL with a smart recommender system. Moreover, to the best of our knowledge, this is the first recommender system specifically built for textual DSLs.

6 RELATED WORK

This section conducts a review of previously proposed recommender systems and places our work in context. Table 9 summarizes the discussed recommender systems on several dimensions: the recommended task, the underlying technique, the operational level of the recommendation engine (i.e., whether it processes the AST or the textual syntax as input), the targeted artifacts, the training and evaluation datasets used, and whether it's integrated into a modeling environment. The initial two rows pertain to PLM-based recommender systems (Sect.6.1), while the subsequent rows relate to traditional ML-based recommender systems (Sect.6.1).

To the best of our knowledge no work has focused on providing recommendations specifically tailored to textual languages. Existing approaches assume that the model to be completed is structurally valid, but this assumption does not work for models edited textually. Therefore, they cannot be integrated in textual languages.

6.1 PLM-based MDE recommender systems

Language models have been used in MDE to build some recommender systems. For instance, Weyssow et al. [31] introduce EcoreBERT, a RoBERTa model trained from scratch on the full MAR/Ecore dataset [14, 15]. By transforming Ecore models into serialized trees, the model predicts masked names (class name, feature name, etc.). The prediction is limited to one subword, this means that a recommendation could be potentially a partial word. To mitigate (but not fully eliminate) this problem, the authors trained the subword tokenizer and the RoBERTa model from scratch. However, this implementation choice hurts the generalizability of the model as the pre-training phase encompasses a relatively small dataset. Chaaben et al. [5] use GPT-3 and a few-shot approach for recommending class and feature names. One shortcoming of this approach is that it relies on GPT-3 which is only accessible via an API.

6.2 ML-based MDE recommender systems

Several recent recommender system proposals opt for more traditional machine learning techniques. For instance, MORGAN [7, 8] leverages graph kernels to support model and meta-model completion, focusing on structural features within models. However, scalability becomes an issue due to the lack of an index for performing k-nearest neighbor matches with respect to the training set, hindering performance with larger datasets. MemoRec [6], employs a collaborative filtering approach for recommending class or structural feature names. The evaluations of MORGAN and MemoRec do not take dataset duplication into account which usually inflates the results in similarity-based approaches [2]. Burgueño et al.[4] propose an NLP-based architecture for completing software models, utilizing two-word embedding models trained with general knowledge and project documentation. These models are then interpolated to provide recommendations. Lastly, Adhikari et al.[1] introduce SimIMA, a recommender system tailored for Simulink models, comprising SimGESTION for suggesting single-step operations via ensemble learning and SimXAMPLE for offering similar models to developers for inspiration.

7 CONCLUSION

This paper tackles the construction of smart editing facilities for textual DSLs, which is an important gap in current DSL environments. To this end, we propose a complete workflow to fine-tune PLMs to tailor them to textual DSLs. ModelMate supports three kinds of editing tasks: identifier suggestion, line completion, and fragment completion. We have evaluated our proposal with various PLMs and DSLs, including Emfatic, Xtext, and a Domain entity DSL.

Approach	Task	Technique	Level	Target artifacts	Training dataset	Evaluation dataset	Integration
Chaaben et al. [5]	Identifier	GPT-3	AST	UML models	N.A.	ModelSet	No
EcoreBERT [31]	Identifier	Training RoBERTa	AST	Ecore models	MAR	MAR	No
MORGAN [8]	Identifier	k-nn + graph kernels	AST	Any	ModelSet BigQuery JSON	ModelSet BigQuery JSON	No
MemoRec [6]	Identifier	Collaborative filtering	AST	Ecore models	Ecore555 Ecore GitHub	Ecore555 Ecore GitHub	No
Burgueño et al. [4]	Identifier	Word embedding + k-nn	AST	UML models	Propietary dataset	Propietary dataset	No
SimIMA [1]	Fragment	Ensemble learning	AST	Simulink	Simulink datatset	Simulink datatset	Yes
ModelMate	Identifier Line Fragment	Fine-tuning PLMs	Textual syntax	Any with textual syntax	MAR	MAR ModelSet	Yes

Table 9: Comparative table of the state-of-the-art recommender systems in the context of MDE.

The results show that ModelMate consistently achieves good performance. In particular, ModelMate outperforms existing alternatives in the feature name recommendation task and it is comparable to OpenAI GPT 3.5. Moreover, we show that the inference time is suitable for real-time assistance scenarios. On the practical side, we contribute a framework to build recommender systems for other DSLs as well as an Eclipse plug-in for integrating them into textual editors and we make available concrete plug-ins for Emfatic and Xtext.

As future work we plan to investigate how to extend our approach for open source LLMs without compromising too much the training and inference speed. We also want to look into how to enrich datasets, for instance, merging several sources as well as analysing its quality. Finally, we are interested in adapting ModelMate to non-textual editors for other tasks like name recommendation in general (not only feature names) and predicting the next editing operation.

Acknowledgments. Work supported by projects TED2021-129381B-C22 (MCIN/AEI/10.13039/501100011033 and NextGenEU/PRTR), PID2022-140109NB-I00 (MCIN/AEI/10.13039/501100011033 and FEDER/UE) and CNS2022-135578 (MICIU/AEI/10.13039/501100011033 and NextGenEU/PRTR).

REFERENCES

[1] Bhisma Adhikari, Eric J Rapos, and Matthew Stephan. 2023. SimIMA: a virtual Simulink intelligent modeling assistant: Simulink intelligent modeling assistance through machine learning and model clones. *Software and Systems Modeling* (2023), 1–28.

[2] Miltiadis Allamanis. 2019. 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*. 143–153.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[4] Loli Burgueño, Robert Clarisó, Sébastien Gérard, Shuai Li, and Jordi Cabot. 2021. An NLP-based architecture for the autocompletion of partial domain models. In *International Conference on Advanced Information Systems Engineering*. Springer, 91–106.

[5] Meriem Ben Chaaben, Lola Burgueño, and Houari Sahraoui. 2023. 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)*. IEEE, 7–12.

[6] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T Nguyen, and Alfonso Pierantonio. 2022. MemoRec: a recommender system for assisting modelers in specifying metamodels. *Software and Systems Modeling* (2022), 1–21.

[7] Juri Di Rocco, Claudio Di Sipio, Davide Di Ruscio, and Phuong T Nguyen. 2021. 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)*. IEEE, 70–81.

[8] Claudio Di Sipio, Juri Di Rocco, Davide Di Ruscio, and Phuong T Nguyen. 2023. MORGAN: a modeling recommender system based on graph kernel. *Software and Systems Modeling* (2023), 1–23.

[9] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (2022).

[10] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027* (2020).

[11] Alex Graves. 2012. Sequence transduction with recurrent neural networks. *arXiv preprint arXiv:1211.3711* (2012).

[12] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533* (2022).

[13] José Antonio Hernández López, Javier Luis Cánovas Izquierdo, and Jesús Sánchez Cuadrado. 2021. ModelSet: a dataset for machine learning in model-driven engineering. *Software and Systems Modeling* (2021), 1–20.

[14] José Antonio Hernández López and Jesús Sánchez Cuadrado. 2020. MAR: A structure-based search engine for models. In *Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems*. 57–67.

[15] José Antonio Hernández López and Jesús Sánchez Cuadrado. 2021. An efficient and scalable search engine for models. *Software and Systems Modeling* (2021), 1–23.

[16] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[17] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[18] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. 2023. Recent advances in natural language processing via large pre-trained language models: A survey. *Comput. Surveys* 56, 2 (2023), 1–40.

[19] Gunter Mussbacher, Benoit Combemale, Jörg Kienzle, Silvia Abrahão, Hyacinth Ali, Nelly Bencomo, Márton Búr, Loli Burgueño, Gregor Engels, Pierre Jeanjean, et al. 2020. Opportunities in intelligent modeling assistance. *Software and Systems Modeling* 19 (2020), 1045–1053.

[20] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[21] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[22] Joshua Peterson, Stephan Meylan, and David Bourgin. 2019. Open clone of openai's unreleased webtext dataset scraper.

[23] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[24] Martin Robillard, Robert Walker, and Thomas Zimmermann. 2009. Recommendation systems for software engineering. *IEEE software* 27, 4 (2009), 80–86.

[25] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).

[26] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, Indira Vats, Hadi Moazen, and Federica Sarro. 2024. A survey on machine learning techniques applied to source code. *Journal of Systems and Software* 209 (2024), 111934.

[27] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th international conference on software engineering*. 1597–1608.

[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[29] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560* (2022).

[30] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).

[31] Martin Weyssow, Houari Sahraoui, and Eugene Syriani. 2022. Recommending metamodel concepts during modeling activities with pre-trained language models. *Software and Systems Modeling* (2022), 1–19.

[32] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244* (2023).