Automated variable renaming: are we there yet?

Antonio Mastropaolo[1] ⬤ · Emad Aghajani[1] · Luca Pascarella[1] · Gabriele Bavota[1]

Abstract

Identifiers, such as method and variable names, form a large portion of source code. There-fore, low-quality identifiers can substantially hinder code comprehension. To support devel-opers in using meaningful identifiers, several (semi-)automatic techniques have been proposed, mostly being data-driven (*e.g.,* statistical language models, deep learning mod-els) or relying on static code analysis. Still, limited empirical investigations have been performed on the effectiveness of such techniques for recommending developers with meaningful identifiers, possibly resulting in rename refactoring operations. We present a large-scale study investigating the potential of data-driven approaches to support automated variable renaming. We experiment with three state-of-the-art techniques: a statistical lan-guage model and two DL-based models. The three approaches have been trained and tested on three datasets we built with the goal of evaluating their ability to recommend mean-ingful variable identifiers. Our quantitative and qualitative analyses show the potential of such techniques that, under specific conditions, can provide valuable recommenda-tions and are ready to be integrated in rename refactoring tools. Nonetheless, our results also highlight limitations of the experimented approaches that call for further research in this field.

✉ Antonio Mastropaolo
antonio.mastropaolo@usi.ch

Emad Aghajani
emad.aghajani@usi.ch

Luca Pascarella
luca.pascarella@usi.ch

Gabriele Bavota
gabriele.bavota@usi.ch

[1] SEART @ Software Institute, Università della Svizzera Italiana, Lugano, Switzerland

1 Introduction

Low-quality identifiers, such as meaningless method or variable names, are a recognized source of issues in software systems (Lin et al. 2019). Indeed, choosing an expressive name for a program entity is not always trivial and requires both domain and contextual knowledge (Carter 1982). Even assuming a meaningful identifier is adopted in the first place while coding, software evolution may make the identifier not suitable anymore to represent a given entity. Moreover, the same entity used across different code components may be named differently, leading to inconsistent use of identifiers (Lin et al. 2017). For these reasons, rename refactoring has become part of developers' routine (Murphy-Hill et al. 2011), as well as a standard built-in feature in modern integrated development environments (IDEs). While IDEs aid developers with the mechanical aspect of rename refactoring, developers remain responsible for identifying low-quality identifiers and choosing a proper rename.

To support developers in improving the quality of identifiers, several techniques have been proposed (Thies and Roth 2010b; Allamanis et al. 2014b; Lin et al. 2017; Alon et al. 2019). Among those, data-driven approaches are on the rise (Allamanis et al. 2014b; Lin et al. 2017; Alon et al. 2019). This is also due to the recent successful application of these techniques in the code completion field (Nguyen et al. 2012; Kim et al. 2021; Svyatkovskiy et al. 2020; Liu et al. 2020; Ciniselli et al. 2021), which is a more general formulation of the variable renaming problem. Indeed, if a model is able to predict the next code tokens that a developer is likely to write (*e.g.,* code completion), then it can be used to predict a token representing an identifier. Nevertheless, strong empirical evidence about the performance ensured by such data-driven techniques for supporting developers in identifier renaming is still minimal.

In this paper, we investigate the performance of three data-driven techniques in supporting automated variable renaming. We experiment with: (i) an n-gram cached language model (Hellendoorn and Devanbu 2017b); (ii) the Text-to-Text Transfer Transformer (T5) model (Raffel et al. 2019), and (iii) the Transformer-based model presented by Liu et al. (2020). The n-gram cached language model (Hellendoorn and Devanbu 2017b) has been experimented against what were state-of-the-art deep learning models in 2017, showing its ability to achieve competitive performance in modeling source code. Thus, it represents a light-weight but competitive approach for the prediction of code tokens (including identifiers). Since then, novel deep learning models have been proposed such as, for example, those based on the transformer architecture (Vaswani et al. 2017a). Among those, T5 has been widely studied to support code-related tasks (Mastropaolo et al. 2021, 2021; Tufano et al. 2022; Mastropaolo et al. 2022; Wang et al. 2021; Ciniselli et al. 2021). Thus, it is representative of transformer-based models exploited in the literature. Finally, the approach proposed by Liu et al. (2020) has been specifically tailored to improve code completion performance for identifiers, known to be among the most difficult tokens to predict. These three techniques provide a good representation of the current state-of-the-art of data-driven techniques for identifiers prediction.

All experimented models require a training set to learn how to suggest "meaningful" identifiers. To this aim, we built a large-scale dataset composed of 1,221,193 instances, where an instance represents a *Java* method with its related local variables. This dataset has been used to train, configure (*e.g.,* hyperparameters tuning), and perform a first assessment of the three techniques. In particular, as done in previous works related to the automation of code-related activities Alon et al. 2019; Tufano et al 2019, 2019; Watson et al. 2020; Haque et al. 2020; Tufano et al. 2021), we considered a prediction generated by the models as

correct if it resembles the choice made by the original developers (*e.g.,* if the recommended variable name is the same chosen by the developers). However, this validation assumes that the identifiers selected by the developers are meaningful, which is not always the case.

To mitigate this issue and strengthen our evaluation, we built a second dataset using a novel methodology we propose to increase the confidence in the dataset quality (in our case, in the quality of the identifiers in code). In particular, we mined variable identifiers that have been introduced or modified during a code review process (*e.g.,* as a result of a reviewer's comment). These identifiers result from a shared agreement among multiple developers, thus increasing the confidence in their meaningfulness. This second dataset is composed of 457 *Java* methods with their related local variables.

Finally, we created a third dataset aimed at simulating the usage of the experimented tools for rename variable refactoring: We collected 400 Java projects in which developers performed a rename variable refactoring. By doing so, we were able to mine 442 valid commits. For each commit c in our dataset, we checked-out the system's snapshot before (s_{c-1}) and after (s_c) the rename variable implemented in c. Given v the variable renamed in c, we run the three techniques on the code in s_{c-1} (*e.g.,* before the rename variable refactoring) to predict v's identifier.

Then, we check whether the predicted identifier is the one implemented by the developers in s_c. If this is the case, this means that the approach was able to successfully recommend a rename variable refactoring for v, selecting the same identifier chosen by developers.

Our quantitative analysis shows that the Transformer-based model proposed by Liu et al. ([2020](#)) is by far the best performing model in the literature for the task of predicting variable identifiers. This confirms the effort performed by the authors that aimed at specifically improve the performance of DL-based models in this task. This approach, named CugLM, can correctly predict the variable identifier in $\sim$ 63% of cases when tested on the large scale dataset we built. Concerning the other two datasets, the performance of all models drop, with CugLM still ensuring the best performance with $\sim$ 45% of correct predictions on both datasets.

We also investigate whether the "confidence of the predictions" generated by the three models (*e.g.,* how "confident" the model are about the generated prediction) can be used as a proxy for prediction quality. We found that when the confidence is particularly high ($>$ 90%), the predictions generated by the models, and in particular by CugLM, have a very high chance of being correct ($>$80% on the large-scale dataset). This suggests that the recommendations generated such tools, under specific conditions (*e.g.,* high confidence) are ready to be integrated in rename refactoring tools.

We complement the study with a qualitative analysis aimed at inspecting "wrong" predictions to (i) see whether, despite being different from the original identifier chosen by the developer, they still represent meaningful identifiers for the variable provided as input; and (ii) distill lessons learned from these failure cases.

Concerning the first point, it is indeed important to clarify that even wrong predictions may be valuable for practitioners. This happens, for example, in the case in which the approach is able to recommend a valid alternative for an identifier (*e.g., surname* instead of *lastName*) or maybe even suggesting a better identifier, thus implicitly recommending a rename refactoring operation. Such an analysis helps in better assessing the actual performance of the experimented techniques.

Finally, we analyze the circumstances under which the experimented tools tend to generate correct and wrong predictions. For example, not surprisingly, we found that these approaches are effective in recommending identifiers that they have already seen used, in a

different context, in the training set. Also, the longer the identifier to predict (*e.g.,* in terms of number of terms composing it), the lower the likelihood of a correct prediction.

Significance of Research Contribution To the best of our knowledge, our work is the largest study at date experimenting with the capabilities of state-of-the-art data-driven techniques for variable renaming across several datasets, including two *high-quality datasets* we built with the goal of increasing the confidence in the obtained results. The three datasets we built and the code implementing the three techniques we experiment with are publicly available Replication package (2022). Our findings unveil the potential of these tools as support for rename refactoring and help in identifying gaps that can be addressed through additional research in this field.

2 Related Work

We start by discussing works explicitly aiming at improving the quality of identifiers used in code. Then, we briefly present the literature related to code completion by discussing approaches that could be useful to suggest a variable name while writing code. For the sake of brevity, we do not discuss studies about the quality of code identifiers and their impact on comprehension activities (see *e.g.,* Lawrie et al. 2006, 2007; Butler et al. 2009, 2010).

2.1 Improving the Quality of Code Identifiers

Before diving into techniques aimed at improving the quality of identifiers, it is worth mentioning the line of research featuring approaches to split identifiers (Guerrouj et al. 2013; Corazza et al. 2012) and expand abbreviations contained in them (Corazza et al. 2012; Hill et al. 2008; Lawrie and Binkley 2011). These techniques can indeed be used to improve the expressiveness and comprehensibility of identifiers since, as shown by Lawrie et al. (2006), developers tend to better understand expressive identifiers.

On a similar research thread, (Reiss 2007) and Corbo et al. (2007) proposed tools to learn coding style from existing source code such that it can then be applied to the code under development. The learned style rules can include information about identifiers such as the token separator to use (*e.g.,* Camel or Snake case) and the presence of a prefix to name certain code entities (*e.g., OBJ_varName*). The rename refactoring approaches proposed by Feldthaus and Møller (2013) and by Jablonski and Hou (2007), instead, focus on the relations between variables, inferring whether one variable should be changed together with others.

Although the above-described methods can improve identifiers' quality (*e.g.,* by expanding abbreviated words and increasing consistency), they cannot address the use of meaningless/inappropriate identifiers as program entities' names.

To tackle this problem, (Caprile and Tonella 2000) proposed an approach enhancing the meaningfulness of identifiers with a standard lexicon dictionary and a grammar collected by analyzing a set of programs, replacing non-standard terms in identifiers with a standard one from the dictionaries.

Thies and Roth (2010a) and Allamanis et al. (2014a) also presented techniques to support the renaming of code identifiers. Thies and Roth (2010a) exploit static code analysis: if a variable v_1 is assigned to an invocation of method m (*e.g.,* `name = getFullName`), and the type of v_1 is identical to the type of the variable v_2 returned by m, then rename v_1 to v_2.

Allamanis et al. (2014a) proposed NATURALIZE, a two-step approach to suggest identifier names. In the first step, the tool extracts, from the code AST, a list of candidate names and, in the second step, it leverages an *n*-gram language model to rank the name list generated in the previous step. The authors evaluated the meaningfulness of the recommendations provided by their approach through analyzing 30 methods (for a total of 33 recommended variable renamings). Half of these suggestions were identified as relevant. Building on top of NATURALIZE, (Lin et al. 2017) proposed LEAR, an approach combining code analysis and *n*-gram language models. The differences between LEAR and NATURALIZE are: (i) while NATURALIZE considers all the tokens in the source code, LEAR only focuses on tokens containing lexical information; (ii) LEAR also considers the type information of variables. Note that these techniques are meant to promote a consistent usage of identifiers within a given project (*e.g.,* renaming variables that represent the same *entity* but are named differently within different parts of the same project). Thus, they cannot suggest naming a variable using an original identifier learned, for example, from other projects.

Differently from previous works, we empirically compare the effectiveness of data-driven techniques to support variable renaming, aiming to assess the extent to which developers could adopt them for rename refactoring recommendations.

From this perspective, the most similar work to our study is the one by Lin et al. (2017). However, while their goal is to promote a consistent usage of identifiers within a project, we aim at supporting identifier rename refactoring in a broader sense, not strictly related to "consistency". Moreover, differently from Lin et al. (2017), we include in our study recent DL-based techniques.

Finally, a number of previous works explicitly focused on the method renaming. Daka et al. (2017) designed a technique to generate descriptive method names for automatically generated unit tests by summarizing API-level coverage goals. Høst and Østvold (2009) identify methods whose names do not reflect the responsibilities they implement. Alon et al. (2019) proposed a neural model to represent code snippets and predict their semantic properties. Then, they use their approach to predict a method's name starting from its body. Differently from these works (Høst and Østvold 2009; Daka et al. 2017; Alon et al. 2019), we are interested in experimenting with data-driven techniques able to handle the renaming of variables.

2.2 Automatic Code Completion

While several techniques have been proposed to support code completion (see *e.g.,* (Nguyen et al. 2012; Foster et al. 2012; Zhang et al. 2012; Bruch et al. 2009; Proksch et al. 2015; Niu et al. 2017; Svyatkovskiy et al. 2020; Jin and Servant 2018; Robbes and Lanza 2010; Svyatkovskiy et al. 2019)), we only discuss approaches able to recommend/generate code tokens representing identifiers.

A precursor of modern code completion techniques is the *Prospector* tool by Mandelin et al. (2005). *Prospector* aims at suggesting, within the IDE, variables or method calls from the user's codebase. Following this goal, other tools such as *InSynth* by Gvero et al. (2013) and *Sniff* by Chatterjee et al. (2009) have added support for type completion (*e.g.,* expecting a type at a given point in the source code, the models search for type-compatible expressions).

Tu et al. (2014) introduced a cache component that exploits code locality in *n*-gram models to improve their support to code completion. Results show that since code is locally repetitive, localized information can be used to improve performance. This enhanced model outperforms standard *n*-gram models by up to 45% in terms of accuracy. On the same line,

(Hellendoorn and Devanbu 2017a) further exploit the cached models considering specific characteristics of code (*e.g.,* unlimited, nested, and scoped vocabulary). They also showed the superiority of their model when compared to deep learning for representing source code. This is one of the three techniques we experiment with (details in Section 3.1).

Karampatsis and Sutton (2019), a few years later, came to a different conclusion: Neural networks are the best language-agnostic algorithm for representing code. To overcome the *out of vocabulary problem*, they propose the use of Byte Pair Encoding (BPE) (Gage 1994), achieving better performance as compared to the cached *n*-gram model proposed in Hellendoorn and Devanbu (2017a). Also Kim et al. (2021) showed that novel DL techniques based on the Transformers neural network architecture can effectively support code completion. Similarly, (Alon et al. 2020) addressed the code completion problem with a language-agnostic approach leveraging the syntax to model the code snippet as a tree. Based on Long short-term Memory networks (LSTMs) and Transformers, the model receives an AST representing a partial statement with some missing tokens to complete. The versatility of addressing general problems with deep learning models pushed researchers to explore the potential of automating various code-related tasks.

For example, (Svyatkovskiy et al. 2020) created *IntelliCode Compose*, a general-purpose code completion tool capable of predicting code sequences of arbitrary tokens. Liu et al. (2020) presented a Transformer-based neural architecture pre-trained on two development tasks: code understanding and code generation.

Successively, the model is specialized with a fine-tuning step on the code completion task. With this double training process, the authors try to incorporate a generic knowledge of the project into the model with the explicit goal of improving performance in predicting code identifiers. This is one of the techniques considered in our study (details in Section 3.3).

Ciniselli et al. (2021) demonstrated how deep learning models can support code completion at different granularities, not only predicting inline tokens but even complete statements. To this aim, they adapt and train a BERT model in different scenarios ranging from simple inline predictions to entire blocks of code.

Finally, Mastropaolo et al. (2021)[1] recently showed that T5 (Raffel et al. 2019), properly pre-trained and fine-tuned, achieves state-of-the-art performance in many code-related tasks, including bug-fixing, mutants injection, code comment and assert statement generation. This is the third approach considered in our study (details in Section 3.2).

Companies have also shown active interest in supporting practitioners while developing sources with code completion tools. For example, IntelliCode was introduced by Svyatkovskiy et al. (2020), a group of researchers working at the Microsoft Corporation. It is multilingual code completion tool that aims at predicting sequences of code tokens such as entire lines of syntactically correct code. IntelliCode leverages a transformer model and has been trained on 1.2 billion lines of diverse programming languages. Similarly, GitHub Copilot has been recently introduced as a state-of-the-art code recommender (Github copilot 2021; Howard 2021). It has been experimentally released through their API that aims at translating natural language descriptions into source code (Chen et al. 2021). Copilot is based on a GPT-3 model (Brown et al. 2020) fine-tuned on publicly available code from GitHub. Nonetheless, the exact dataset used for its training is not publicly available. This hinders the possibility to use it in our study, since we cannot check for possible overlaps

---

[1]In the following we refer to Mastropaolo et al. (2021) as Mastropaolo et al. because the set of authors only partially overlaps.

between the Copilot's training set and the test sets used in our study (also collected from GitHub open source projects).

The successful applications of DL for code completion suggest its suitability for supporting rename refactoring. Indeed, if a model is able to predict the identifier to use in a given context, it can be used to boost identifiers' quality as well. As a representative of DL-based techniques to experiment with variable rename refactoring task, we chose (i) the model by Liu et al. (2020), given its focus on the predicting code identifiers; and (ii) the T5 model as used in Mastropaolo et al. (2021), given its state-of-the-art performance across many code-related tasks. We acknowledge that other choices are possible given the vast literature in the field. We focused on two DL-models recently published in top software engineering venues (*e.g.,* ASE'20 (Liu et al. 2020) and ICSE'21 (Mastropaolo et al. 2021)).

3 Data-driven Variable Renaming

In our study, we aim at assessing the effectiveness of data-driven techniques for automated variable renaming. We focus on three techniques representative of the state-of-the-art. The first is a statistical language model that showed its effectiveness in modeling source code (Hellendoorn and Devanbu 2017b). The second, T5 (Raffel et al. 2019), is a recently proposed DL-based technique already applied to address code-related tasks (Mastropaolo et al. 2021). The third is the Transformer-based model presented by Liu et al. (2020) to boost code completion performance on identifiers.

Figure 1 depicts the scenario in which these techniques have been experimented. We work at method-level granularity: For each local variable v declared in a method m, we mask every v's reference in m asking the experimented techniques to recommend a suitable name for v. If the recommended name is different from the original one, a rename variable recommendation can be triggered.

We provide an overview of the experimented techniques, pointing the reader to the papers introducing them (Hellendoorn and Devanbu 2017b; Raffel et al. 2019; Liu et al. 2020) for additional details. Our implementations are based on the ones made available by the original authors of these techniques and are publicly available in our replication package (Replication package 2022). The training of the techniques is detailed in Section 4.

3.1 N-gram Cached Model

Statistical language models can assess a probability of a given sequence of words. The basic idea behind these models is that the higher the probability, the higher the "*familiarity*" of the scored sequence. Such *familiarity* is learned by training the model on a large text corpus. An n-gram language model predicts a single word following the $n - 1$ words preceding it. In other words, n-gram models assume that the probability of a word only depends on the previous $n - 1$ words.
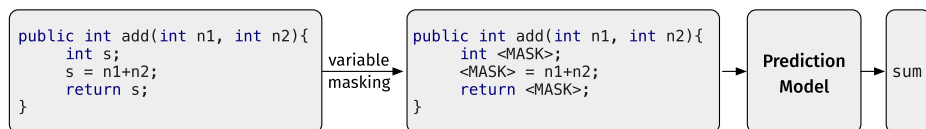


```
public int add(int n1, int n2){
    int s;
    s = n1+n2;
    return s;
}
```
variable masking →
```
public int add(int n1, int n2){
    int <MASK>;
    <MASK> = n1+n2;
    return <MASK>;
}
```
→ Prediction Model → sum

Fig. 1 Variable renaming scenario

Hellendoorn and Devanbu (2017b) discuss the limitations of *n*-gram models that make them suboptimal for modeling code (*e.g.,* the unlimited vocabulary problem due to new words that developers can define in identifiers). To overcome these limitations, the authors present a *dynamic, hierarchically scoped, open vocabulary language model* (Hellendoorn and Devanbu 2017b), showing that it can outperform Recurrent Neural Networks (RNN) and LSTM in modeling code. While Karampatsis and Sutton (2019) showed that DL models can outperform the cached *n*-gram model, the latter ensures good performance at a fraction of the DL models training cost, making it a competitive baseline for code-related tasks.

3.2 Text-To-Text-Transfer-Transformer (T5)

The T5 model has been introduced by Raffel et al. (2019) to support multitask learning in Natural Language Processing. The idea is to reframe NLP tasks in a unified text-to-text format in which the input and output of all tasks to support are always text strings. For example, a single T5 model can be trained to translate across a set of different languages (*e.g.,* English, German) and identify the sentiment expressed in sentences written in any of those languages. This is possible since both these tasks (*e.g.,* translation and sentiment identification) are text-to-text tasks, in which a text is provided as input (*e.g.,* a sentence in a specific language for both tasks) and another text is generated as output (*e.g.,* the translated sentence or a label expressing the sentiment). T5 is trained in two phases: *pre-training*, which allows defining a shared knowledge-base useful for a large class of text-to-text tasks (*e.g.,* guessing masked words in English sentences to learn about the language), and *fine-tuning*, which specializes the model on a specific downstream task (*e.g.,* learning the translation of sentences from English to German). As previously said, T5 already showed its effectiveness in code-related tasks (Mastropaolo et al. 2021). However, its application to variable renaming is a premier. Among the T5 variants proposed by Raffel et al. (2019) that mostly differ in terms of architectural complexity, we adopt the smallest one (T5$_{small}$). The choice of such architecture is driven by our limited computational resources. However, we acknowledge that bigger models have been shown to further increase performance (Raffel et al. 2019).

3.3 Deep-Multi-Task Code Completion Model

Liu et al. (2020) recently proposed the *Code Understanding and Generation pre-trained Language Model (CugLM)*, a BERT-based model for source code modeling. Albeit under-the-hood CugLM still features a Transformer-based network (Vaswani et al. 2017b) as T5, such an approach has been specifically conceived to improve the performance of language models in identifiers, thus making it very suitable for our study on variable renaming. CugLM is pre-trained using three objectives. The first asks the model to predict masked identifiers in code (being thus similar to the one used in the T5 model, but focused on identifiers). The second task asks the model to predict whether two fragments of code can follow each other in a snippet. Finally, the third is a left-to-right language modeling task, in which the classic code completion scenario is simulated, *e.g.,* given some tokens (left part), guess the following token (right part).

Once pre-trained, the model is fine-tuned for code completion in a multi-task learning setting, in which the model has first to predict the type of the following token and, then, the predicted type is used to foster the prediction of the token itself. As reported by the authors, such an approach achieves state-of-the-art performance when it comes to predicting code identifiers.

4 Study Design

The *goal* is to experiment the effectiveness of data-driven techniques in supporting automated variable renaming. The *context* is represented by (i) the three techniques (Hellendoorn and Devanbu 2017b; Raffel et al. 2019; Liu et al. 2020) introduced in Section 3 and (ii) three datasets we built for training and evaluating the approaches. Our study answers the following research question: ***To what extent can data-driven techniques support automated variable renaming?***

4.1 Datasets Creation

To train and evaluate the experimented models, we built three datasets: (i) the *large-scale dataset*, used to train the models, tune their parameters, and perform a first assessment of their performance; (ii) the *reviewed dataset* and (iii) the *developers dataset* used to further assess the performance of the experimented techniques. Our quantitative evaluation is based on the following idea: If, given a variable, a model is able to recommend the same identifier name as chosen by the original developers, then the model has the potential to generate meaningful rename recommendations.

Clearly, there is a strong assumption here, namely that the identifier selected by the developers is meaningful. For this reason, we have three datasets. The first one (*large-scale dataset*) aims at collecting a high number of variable identifiers that are needed to train the data-driven models and test them on a large number of data points. The second one (*reviewed dataset*) focuses instead on creating a test set of high-quality identifiers for which our assumption can be more safely accepted: These are identifiers that have been modified or introduced during a code review process. Thus, more than one developer agreed on the appropriateness of the chosen identifier name for the related variable. Finally, the third dataset (*developers dataset*) focuses on identifiers that have been subject to a rename refactoring operation (*e.g.,* the developer put effort in improving the quality of the identifier through a refactoring). Again, this increases our confidence in the quality of the considered identifiers.

In this section, we describe the datasets we built, while Section 4.2 details how they have been used to train, tune, and evaluate the three models.

4.1.1 Large-scale Dataset

We selected projects on Github website (2008) by using the search tool by Dabic et al. (2021). This tool indexes all GitHub repositories written in 13 different languages and having at least 10 stars, providing a handy querying interface (SEART github search 2021) to identify projects meeting specific selection criteria. We extracted all *Java* projects having at least 500 commits and at least 10 contributors. We do so as an attempt to discard toy/personal projects. We decided to focus on a single programming language to simplify the toolchain building needed for our study. Also, we excluded forks to reduce the risk of duplicated repositories in our dataset.

Such a process resulted in 5,369 cloned Java projects from which we selected the 1,425 using Maven (2004)[2] and having their latest snapshot being compilable. Maven allows to

---

[2]Maven is a software project management tool that, as reported in its official webpage (https://maven.apache.org) "can manage a project's build, reporting and documentation from a central piece of information".

quickly verify the compilability of the projects, which is needed to extract information about types needed by one of the experimented models (*e.g., CugLM*). *CugLM* leverages identifiers' type information to improve its predictions. To be precise and comprehensive in type resolution, we decided to rely on the JavaParser library (Javaparser 2019), running it on compilable projects. This allows to resolve also types that are implemented in imported libraries. We provide the tool we built for such an operation as part of our replication package (Replication package 2022).

We used *srcML* (Scrml website 2019) to extract from each *Java* file contained in the 1,425 projects all methods having *#tokens* ≤ *512*, where *#tokens* represents the number of tokens composing a function (excluding comments). The filter on the maximum length of the method is needed to limit the computational expense of training DL-based models (similar choices have been made in previous works (Tufano et al. 2019; Haque et al. 2020; Tufano et al. 2021), with values ranging between 50 and 100 tokens). All duplicate methods have been removed from the dataset to avoid overlap between training and test sets we built from them.

From these 1,425 repositories, we set apart 400 randomly selected projects for constructing the *developers dataset* (described in Section 4.1.3). Concerning the remaining 1,025, we use ~40% of them (418 randomly picked repositories) to build a dataset needed for the pre-training of the T5 (Raffel et al. 2019) and of the *CugLM* model (Liu et al. 2020) (*pre-training dataset*). Such a dataset is needed to support the pre-training phase that, as shown in the literature, helps deep learning models to achieve better performance when dealing with code-related tasks (Tufano et al. 2020, 2022; Mastropaolo et al. 2022; Ciniselli et al. 2021). Indeed, the pre-training phase conveys two major advantages summarized as follows: (i) once the model has been pre-trained, it can learn general representations and patterns of the language the model is working with, (ii) the pre-trained model yields to a more robust model initialization of the neural network weights that can then support the specialization phase (*e.g.,* fine-tuning).

The remaining 615 projects (*large-scale dataset*) have been further split into training (60%), evaluation (20%), and test (20%). The training set has been used to fine-tune the two DL-based models (*e.g.,* T5 and *CugLM*). This dataset, joined with the *pre-training dataset*, has also been used to train the *n*-gram model. In this way, all models have been trained using the same set of data, with the only difference being that the training is organized in two steps (*e.g.,* pre-training and fine-tuning) for T5 and *CugLM*, while it consists of a single step for the *n*-gram model.

For the T5 model, we used the evaluation set to tune its hyperparameters (Section 4.2), since no previous work applied such a model for the task of variable renaming. Instead, for *CugLM* and *n*-gram we used the best configurations reported in the original works presenting them Hellendoorn and Devanbu (2017b), Liu et al. (2020). Finally, the test set has been used to perform a first assessment of the models' performance.

Table 1 shows the size of the datasets in terms of the number of extracted methods (*reviewed dataset* and *developers dataset* are described in the following).

4.1.2 Reviewed Dataset

Also in this case, we selected GitHub projects using the tool by Dabic et al. (2021). Since the goal for the *reviewed dataset* is to mine code review data, we added on top of the selection criteria used for the *large-scale dataset* a minimum of 100 pull requests per selected project. Also in this case we only selected Maven projects having their latest snapshot successfully compiling. We then mined from the 948 projects we obtained information related to the code

Table 1 Num. of methods in the datasets used in our study

Dataset	train	eval	test
pre-training dataset	500,414	–	–
large-scale dataset	394,574	176,944	149,261
reviewed dataset	–	–	457
developers dataset	–	–	442

review performed in their pull requests. Let us assume that a set of files C_s is submitted for review. A set of reviewer comments R_c can be made on C_s possibly resulting in a revised version of the code C_{r_1}.

Such a process is iterative and can consists of several rounds each one generating a new revised version C_{r_i}. Eventually, if the code contribution is accepted for merging, this concludes the review process with a set of C_f files. This whole process "transforms" C_s → C_f. We use *srcML* to extract from both C_s and C_f the list of methods in them and, by performing a *diff*, we identify all variables that have been introduced or modified in each method as result of the review process (*e.g.,* all variables that were not present in C_s but that are present in C_f). We conjecture that the identifiers used to name these variables, being the output of a code review process, have a higher chance of representing high quality data that can be used to assess the performance of the experimented models.

Also in this case we removed duplicate methods both (i) within the *reviewed dataset*, and (ii) between it and the previous ones (*pre-training dataset* and training set of *large-scale dataset*), obtaining 457 methods usable as a further test set of the three techniques.

4.1.3 Developers' Dataset

We run *Refactoring miner* (Tsantalis et al. 2020) on the history of the 400 Java repositories we previously put aside. Refactoring miner is the state-of-the-art tool for refactoring detection in Java systems. We run it on every commit performed in the 400 projects, looking for commits in which a *Rename Variable refactoring* has been performed on the local variable of a method. This gives us, for a given commit c_i, the variable name at commit c_{i-1} (*e.g.,* before the refactoring) and the renamed variable in commit c_i. We use this set of commits as an additional test set (*developers dataset*) to verify if, by applying the experimented techniques on the c_{i-1} version, they are potentially able to recommend a rename (*e.g.,* they suggest, for the renamed variable, the identifier applied with the rename variable refactoring). After removing duplicated methods from this dataset as well (similarly, we also removed duplicates between *developers dataset*, *pre-training dataset*, and the training set in *large-scale dataset*), we ended up with 442 valid instances.

4.2 Training and Hyperparameters Tuning of the Techniques

4.2.1 N-gram Model

The *n*-gram model has been trained on the instances in *pre-training dataset* ∪ *large-scale dataset*. This means that all Java methods contained in *pre-training dataset* and in the training set of *large-scale dataset* have been used for learning the probability of sequences of tokens. We use $n = 3$ since higher values of n have been proven to result in marginal performance gains (Hellendoorn and Devanbu 2017b).

4.2.2 T5

To pre-train the T5, we use a self-supervised task similar to the one by Raffel et al. (2019), in which we randomly mask 15% of code tokens in each instance (*e.g.,* a Java method) from the *pre-training dataset*, asking the model to guess the masked tokens. Such a training is intended to give the model general knowledge about the language, such that it can perform better on a given *down-stream* task (in our case, guessing the identifier of a variable). The pre-training has been performed for 200k steps (corresponding to ∼13 epochs on our *pre-training dataset*) since we did not observe any improvement going further. We used a 2x2 TPU topology (8 cores) from Google Colab to train the model with a batch size of 128. As a learning rate, we use the *Inverse Square Root* with the canonical configuration (Raffel et al. 2019). We also created a new *SentencePiece* model (*e.g.,* a tokenizer for neural text processing) by training it on the entire pre-training dataset so that the T5 model can properly handle the Java language. We set its size to 32k word pieces.

In order to find the best configuration of hyper-parameters, we rely on the same approach used by Mastropaolo et al. (2021). Specifically, we do not tune the hyperparameters of the T5 model for the pre-training (*e.g.,* we use the default ones), because the pre-training itself is task-agnostic, and tuning may provide limited benefits. Instead, we experiment with four different learning rate schedulers for the fine-tuning phase. Since this is the first time T5 is used for recommending identifiers, we also perform an ablation study aimed at assessing the impact of pre-training on this task. Thus, we perform the hyperparameter tuning for both the pre-trained and the non pre-trained model, experimenting with the four configurations in Table 2: constant (C-LR), slanted triangular (ST-LR), inverse square (ISQ-LR), and polynomial (PD-LR) learning rate. We experiment the same configurations for the pre-trained and the non-pretrained models, with the only difference being the $LR_{starting}$ and LR_{end} of the PD-LR. Indeed, in the non pre-trained model (ablation study), we had to lower those values to make the gradient stable (see Table 2).

We fine-tune the T5 for 100k steps for each configuration. Then, we compute the percentage of correct predictions (*e.g.,* cases in which the model can correctly predict the masked variable identifier) achieved in the evaluation set. The achieved results reported in Table 3 showed a superiority of the ST-LR (second column) for the non pre-trained model, while for the pre-trained model, the PD-LR works slightly better. Thus, we use these two scheduler in our study for fine-tuning the final models for 300k steps.

Table 2 Hyperparameters tuning for the T5 Model

Learning Rate	Parameters	Pre-Trained	No Pre-Trained
C-LR	LR	0.001	0.001
ST-LR	$LR_{starting}$	0.001	0.001
	LR_{max}	0.01	0.01
	$Ratio$	32	32
	Cut	0.1	0.1
ISQ-LR	$LR_{starting}$	0.01	0.01
	$Warmup$	10,000	10,000
PD-LR	$LR_{starting}$	0.01	0.001
	LR_{end}	0.01	0.001
	$Power$	0.5	0.5

Table 3 T5 hyperparameter tuning results

Experiment	C-LR	ST-LR	ISQ-LR	PD-LR
Pre-trained	**30.74%**	29.11%	**30.77%**	**30.80%**
No Pre-trained	21.18%	**27.56%**	26.08%	23.90%

The fine-tuning of the T5 required some further processing to the *large-scale dataset*. Given a Java method m having n distinct local variables, we create n versions of it m_1, $m_2, \ldots, m_n$ each one having all occurrences of a specific variable masked with a special token. Such a representation of the dataset allows to fine-tune the T5 model by providing it pairs (m_j, i_j), where m_j is a version of m having all occurrences of variable v_j replaced with a `<MASK>` token and i_j is the identifier selected by the developers for v_j. This allows the T5 to learn proper identifiers to name variables in specific code contexts. The same approach has been applied on the *large-scale dataset* evaluation and test set, as well as on the *reviewed dataset* and *developers dataset*. In these cases, an instance is a method with a specific variable masked, and the trained model is used to guess the masked identifier. Table 4 reports the number of instances in the datasets used for the T5 model. Note that such a masking processing was not needed for the n-gram model nor for *CugLM*, since they just scan the code tokens during training, and they try to predict each code token sequentially during testing. Still, it is important to highlight that all techniques have been trained and tested on the same code.

4.2.3 CugLM

To pre-train and fine-tune the *CugLM* model we first retrieved the identifiers' type information for all code in the *pre-training dataset* and *large-scale dataset*. Then, we leveraged the script provided by the original authors in the replication package (CugLM Model 2020) to obtain the final instances in the format expected by the model. For both pre-training and fine-tuning (described in Section 3.3), we rely on the same hyper-parameters configuration used by the authors in the paper presenting this technique (Liu et al. 2020).

4.3 Performance Assessment

We assess the performance of the trained models against the *large-scale test set*, the *reviewed dataset*, and the *developers dataset*. For each prediction made by each model, we collect a measure acting as "confidence of the prediction", *e.g.,* a real number between 0.0 and 1.0 indicating how confident the model is about the prediction. For the n-gram model, such a measure is a transformation of the entropy of the predictions. Concerning the T5, we exploited the `score` function to assess the model's confidence on the provided input. The value returned by this function ranges from minus infinity to 0 and it is the log-likelihood (ln) of the prediction. Thus, if it is 0, it means that the likelihood of the prediction is 1 (*e.g.,*

Table 4 Instances in the datasets used for training, evaluating, testing the T5 model

	train	eval	test
large-scale dataset	1,122,864	521,779	437,384
reviewed dataset	–	–	457
developers dataset	–	–	442

the maximum confidence, since $ln(1) = 0$), while when it goes towards minus infinity, the confidence tends to 0. Finally, CugLM outputs the *log-prob* for each predicted tokens. Hence, we normalize this value throught the *exp* function.

We investigate whether the confidence of the predictions represents a good proxy for their quality. If the confidence level is a reliable indicator of the predictions' quality (*e.g.,* 90% of the predictions having $c > 0.9$ are correct), it can be extremely useful in the building of recommender systems aimed at suggesting rename refactorings, since only recommendations with high confidence could be proposed to the developer. We split the predictions by each model into ten intervals, based on their confidence c going from 0.0 to 1.0 at steps of 0.1 (*e.g.,* first interval includes all predictions having $0 \leq c < 0.1$, last interval has $0.9 \leq c$). Then, we report for each interval the percentage of *correct predictions* generated by each model in each interval. To assess the performance of the techniques overall, we also report the percentage of correct predictions generated by the models on the entire test datasets (*e.g.,* by considering predictions at any confidence level).

A prediction is considered "correct" if the predicted identifier corresponds to the one chosen by developers in the *large-scale dataset* and in the *reviewed dataset*, and if it matches the renamed identifier in the *developers dataset*. However, a clarification is needed on the way we compute the correct predictions. We explain this process through Fig. 2, showing the output of the experimented models, given an instance in the test sets.

The grey box in Fig. 2 represents an example of an instance in the test set: a function having all references to a specific variable originally named `sum` masked. The T5 model, given such an instance as input, predicts a **single** identifier (*e.g.,* `sum`) for all three references of the variable. Thus, for the T5, it is easy to say whether the single generated prediction is equivalent to the identifier chosen by the developers or not. The *n*-gram and the *CugLM* model, instead, generate three predictions, one for each of the masked instances, despite they represent the same identifier.

Thus, for these two models, we use two approaches to compute the percentage of correct predictions in the test sets. The first scenario, named *complete-match*, considers the prediction as correct only if all three references to the variable are correctly predicted. Therefore, in the example in Fig. 2, the prediction of the *CugLM* model (2 out of 3) is considered wrong. Similarly, the *n*-gram prediction (1 out of 3 correct) is considered wrong. The second scenario, named *partial-match*, considers a prediction as correct if at least one of the instances is correctly predicted (thus, in Fig. 2 both the *n*-gram and the *CugLM* predictions are considered correct).

We also statistically compare the performance of the models in terms of correct predictions: We use the McNemar's test (McNemar 1947), which is a proportion test suitable to pairwise compare dichotomous results of two different treatments. We statistically compare each pair of techniques in our study (*e.g.,* T5 *vs CugLM*, T5 *vs n*-gram, *CugLM vs n*-gram). To compute the test results for two techniques T_1 and T_2, we create a confusion matrix counting the number of cases in which (i) both T_1 and T_2 provide a correct prediction, (ii)
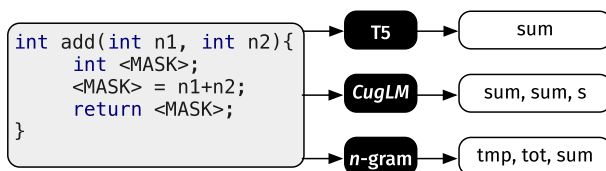

Fig. 2 Example of models' outputs for a test instance

only T_1 provides a correct prediction, (iii) only T_2 provides a correct prediction, and (iv) neither T_1 nor T_2 provide a correct prediction. We complement the McNemar's test with the Odds Ratio (OR) effect size. Also, since we performed multiple comparisons, we adjusted the obtained p-values using the Holm's correction (Holm 1979).

We also manually analyzed a sample of wrong predictions generated by the approaches with the goal of (i) assessing whether, despite being different from the original identifiers used by the developers, they were still meaningful; and (ii) identifying scenarios in which the experimented techniques fail. To perform such an analysis, we selected the top-100 wrong predictions for each approach (300 in total) in terms of confidence level. Three of the authors inspected all of them, trying to understand if the generated variable name could have been a valid alternative for the target one, while the fourth author solved conflicts. Note that, given 100 wrong predictions inspected for a given model, we do not check whether the other models correctly predict these cases. This is not relevant for our analysis, since the only goal is to understand the extent to which the "wrong" predictions generated by each model might still be valuable for developers.

Finally, we compare the correct and wrong predictions in terms of (i) the size of the context (*e.g.,* number of tokens composing the method and number of times a variable is used in the method), (ii) the length of the identifier in terms of number of characters and number of words composing it (as obtained through camelCase splitting); (iii) the number of times the same identifier appears in the training data. We show these distributions using boxplots comparing the two groups of predictions (*e.g.,* compare the length of identifiers in correct and wrong predictions). We also compare these distributions using the Mann-Whitney test (Conover 1998) and the Cliff's Delta (d) to estimate the magnitude of the differences (Grissom and Kim 2005). We follow well-established guidelines to interpret the effect size: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ (Grissom and Kim 2005).

5 Results Discussion

Table 5 reports the results achieved by the three experimented models for each dataset in terms of correct predictions. For T5, both pre-trained and non pre-trained versions are presented. For the n-gram and CugLM, we report the results both when using *perfect match* and the *partial match* heuristic to compute the correct predictions, while this was not required for the T5 for which the results should be interpreted as *perfect matches* (see Section 4.3).

Before commenting on the results is also important to clarify that the cached n-gram model (Hellendoorn and Devanbu 2017b) exploited, as compared to the other two models, additional information due to the caching mechanism. Indeed, the caching allows the model to "look" at code surrounding the one for which tokens must be predicted (in our case, the method in which we want to predict the variable identifier). Given a method m_t in the test set, we provide its cloned repository as "test folder" to the n-gram model, in such a way that it is leveraged by the caching mechanism (we used the implementation from Hellendoorn and Devanbu (2017b)).

Two observations can be easily made by looking at Table 5. First, for T5, the pre-trained model works (as expected) better than its non pre-trained version. From now on, we focus on the pre-trained T5 in the discussion of the results. Second, consistently for all three datasets, CugLM outperforms the other models by a significant margin. In particular, when looking at the correct predictions (*complete match*), the improvement is +26% and +53% over T5 and

Table 5 Correct predictions: C-match indicates the *complete-match* heuristic, P-match the *partial-match*

#Instances	n-gram (c-match)		n-gram (p-match)		CugLM (c-match)		CugLM (p-match)		T5 (non pre-trained)		T5 (pre-trained)	
	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct
large-scale dataset												
437,384	46,126	10.54%	167,868	38.38%	277,595	63.46%	296,590	67.80%	153,708	35.14%	163,368	37.35%
reviewed dataset												
457	20	4.37%	118	25.82%	214	48.35%	232	50.75%	142	31.07%	153	33.48%
developers dataset												
442	8	1.90%	66	14.93%	197	45.02%	209	47.28%	87	19.68%	101	22.85%

n-gram, respectively, in the *large-scale dataset*. The gap is smaller but still substantial for the *reviewed dataset* (+15% and +44% for T5 and *n*-gram) and for the *developers dataset* (+22% and +43%). The difference in performance in favor of CugLM is always statistically significant (see Table 6), with ORs going from 1.98 to 98.0. For example, on the *large-scale dataset* the ORs indicate that CugLM has 3.54 and 23.06 higher odds to generate a correct prediction as compared to the T5 and the *n*-gram model. These results confirm the suitability of the model proposed by Liu et al. (2020) when it comes to predicting code identifiers.

Table 5 also shows that, as expected by construction, the percentage of correct predictions generated by CugLM and by the *n*-gram model increases when considering the *partial match* heuristic. However, for a fair comparison with the T5 model, we mostly focus our discussion on the *perfect match* scenario, that is also the one used in the computation of the statistical tests (Table 6).

The trend in performance is the same across the three datasets. However, the accuracy of all models drops on the *reviewed dataset* and on the *developers dataset*. Still, even in this scenario, CugLM is able to correctly recommend ∼50% of identifiers.

Figure 3 depicts the relationship between the percentage of correct predictions and the confidence of the models. Similarly, to Fig. 4, the orange line represents the *n*-gram model, while the purple and red lines represent CugLM and the T5 pre-trained model, respectively. Within each confidence interval (*e.g.,* 0.9-1.0) the line shows the percentage of correct predictions generated by the model (*e.g.,* ∼80% of predictions having a confidence higher than 0.9 are correct for CugLM in the *large-scale dataset*). The achieved results show a clear trend for all models: Higher confidence corresponds to higher prediction quality. The best performing model (CugLM) is able, in the highest confidence scenario, to obtain 66% of correct predictions on the *developers dataset*, 71% on the *reviewed dataset*, and 82% on the *large-scale dataset*. These results have a strong implication for the building of rename refactoring recommenders on top of these approaches: Giving the possibility to the user (*e.g.,* the developer) to only receive recommendations when the model is highly confident can discard most of the false positive recommendations.

Concerning the manual analysis we performed on 100 wrong recommendations generated by each model on the *large-scale dataset*, a few findings can be distilled. First, the three authors observed that the T5 was the one more frequently generating, in the set of wrong predictions we analyzed, identifiers that were meaningful in the context in which they were proposed (despite being different from the original identifier used by the developers). For example, `value` was recommended instead of `number` or `harvestTasks` instead of

Table 6 McNemar's test (adj. *p*-value and OR) considering complete matches as correct predictions

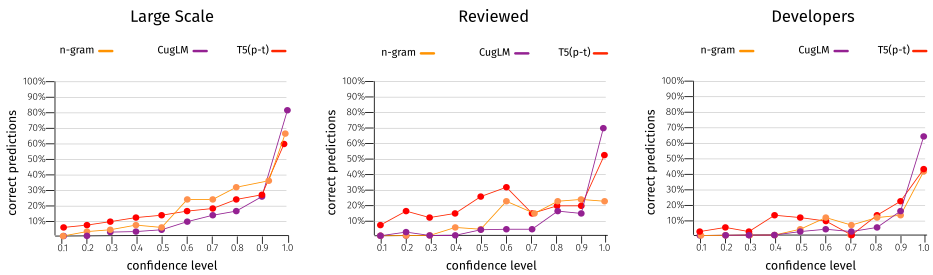| Dataset	Test	*p*-value	OR
	CugLM *vs* T5 (p-t)	< 0.001	3.54
large-scale dataset	CugLM *vs* *n*-gram	< 0.001	23.06
	T5 (p-t) *vs* *n*-gram	< 0.001	6.08
	CugLM *vs* T5 (p-t)	< 0.001	1.98
reviewed dataset	CugLM *vs* *n*-gram	< 0.001	98.0
	T5 (p-t) *vs* *n*-gram	< 0.001	10.50
	CugLM *vs* T5 (p-t)	< 0.001	3.90
developers dataset	CugLM *vs* *n*-gram	< 0.001	32.50
	T5 (p-t) *vs* *n*-gram	< 0.001	24.25

P-t=pre-trained

Fig. 3 Percentage of correct predictions by confidence level

tasks. The three authors agreed on 31 meaningful identifiers proposed by the T5 in the set of 100 wrong predictions they inspected. Surprisingly, this was not the case for the other two models, despite the great performance we observed for CugLM. However, a second observation we made partially explains such a finding: We found that several failure cases of CugLM and of the n-gram model are due the recommendation of identifiers already used somewhere else in the method and, thus, representing wrong recommendations. We believe this is due to the different prediction mechanism adopted by the T5 as compared to the other two models. As previously explained, the T5 generates a single prediction for all instances of the identifier to predict, thus considering the whole method as a context for the prediction and inferring that identifiers already used in the context should not be recommended.

The other two models, instead, scan the method token by token predicting each identifier instance in isolation. This means that if an identifier x is used for the first time in the method after the first instance of the identifier p to predict (*e.g.,* p appears in line 2 while x appears in line 7), the existence of x is not considered when generating the prediction for p. This reduces the information available to CugLM and to the n-gram model. Also, CugLM has limitations inherited from the fixed size of its vocabulary set to the 50k most frequent tokens (Liu et al. 2020), which are the only ones the model can predict. This means that CugLM is likely to fail when dealing with rare identifiers composed by several words. The T5, using the SentencePiece tokenizer, can instead compose complex identifiers.

Finally, Fig. 5 shows the comparison of five characteristics between correct (in green) and wrong (in red) predictions. The comparison has been performed on the three datasets (see labels on the right side of Fig. 5), and for correct/wrong predictions generated by the three models (see labels on the left side of Fig. 5). The characteristics we inspected are summarized at the top of Fig. 5. For each comparison (*e.g.,* each pair of boxplots in Fig. 5) we include a $*$ if the Mann-Whitney test reported a significant difference (p-value < 0.05) and, if this is the case, the magnitude of the Cliff's Delta is reported as well.
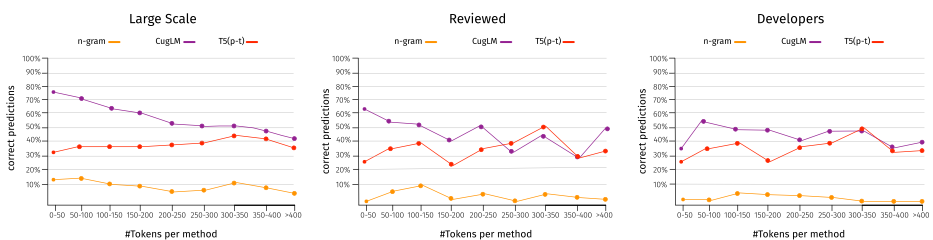

Fig. 4 Percentage of correct predictions by tokens per method

Fig. 5 Characteristics of correct (green) and wrong (red) predictions

Concerning the length of the target identifier (*#Characters per identifier* and *#Tokens per identifier*), the models tend to perform better on shorter identifiers, with this difference being particularly strong for CugLM. Indeed, this is the only model for which we observed a large effect size in the length of identifiers between correct and wrong predictions. Focusing for example on the length in terms of number of tokens (*#Tokens per identifiers*), it is clear that excluding rare exceptions, CugLM mostly succeeds for one-word identifiers. This is again likely to be a limitation dictated by the fixed size of the vocabulary (50k tokens) that cannot contain all possible combinations of words used in identifiers.

The size of the coding context (*e.g.,* method) containing the identifier to predict (*#Tokens per method*) does not seem to influence the correctness of the prediction, with few significant differences accompanied by a negligible or small effect size. This is also visible by looking at Fig. 4, which portrays the relationship between the percentage of correct predictions and the number of tokens composing the input method. The orange line represents the *n*-gram model, while the purple and red lines represent CugLM and the T5 pre-trained model, respectively. Within each interval (*e.g.,* 0-50), the line shows the percentage of

correct predictions generated by the model for methods having a tokens length falling in that bucket. The only visible trend is that of *CugLM* on the *large-scale dataset*, which shows a clear downward trend in correct predictions with the increase in length of the input method. This is indeed the only scenario for which the statistical tests reported a significant differences in the method length of correct and wrong predictions with a small effect size (in all other cases, the difference is not significant or accompanied by a small effect size).

Differently, identifiers appearing in the training set tend to help the prediction (*#Overlapping identifiers within the training set*). This is particularly true for CugLM (large effect size on all datasets), since its vocabulary is built from the training set. The boxplot for the wrong predictions is basically composed only by outliers, with its third quartile equal 0. This indicates that the predictions on which CugLM fails are usually those for identifiers never seen in the training set.

Finally, the number of times that an identifier to predict appears in the context (*e.g.,* #Occurences of identifier within methods), only has an influence for the T5 on the *large-scale dataset*. However, there is no strong trend to discuss for this characteristic.

5.1 Implications of our Findings

Our findings have implications for practitioners and researchers. For the first, our results show that modern DL-based techniques presented in the literature may be already suitable to be embedded in rename refactoring engines. Clearly, they still suffer of limitations that we will discuss later. However, especially when the confidence of their predictions is high, the generated identifiers are often meaningful, matching the ones chosen by developers.

In terms of research, there are a number of improvements these tools can benefit from. First, we noticed that the main weakness of the strongest approach we tested (*e.g.,* CugLM) is the fixed vocabulary size. Such a problem has been addressed in other models using tokenizers such as byte pair encoding (Gage [1994]) or the SentencePiece tokenizer exploited by the T5. Integrating these tokenizers in CugLM (or similar techniques) could help in further improving performance. Second, we noticed that several false-positive recommendations could be avoided by just integrating into the models more contextual information.

For example, if the model is employed to recommend an identifier in a given location l, other identifiers having l in their scope do not represent a viable option, since they are already in use. Similarly, the integration of type information in CugLM demonstrated the boost of performance that can be obtained when the prediction model is provided with richer data. Also, the employed models are predicting identifier names without exploiting information such as (i) the original identifier name that could be improved via a rename refactoring, and (ii) the naming convention adopted in the project. Augmenting the context provided to the models with such information might substantially boost their prediction performance.

Finally, while we performed an extensive study about the capabilities of data-driven techniques for variable renaming, our experiments have been performed in an "artificial" setting. The (mostly positive) achieved results encourage the natural next step represented by case studies with developers to assess their perceived usefulness of these techniques.

6 Threats to Validity

Construct Validity Our study is largely based on one assumption: The identifier name chosen by developers is the correct one the models should predict. We addressed this threat when building two of our datasets: (i) we ensure that the variable identifiers in the *reviewed*

dataset have been checked in the context of a code review process involving multiple developers; (ii) we built *developers dataset* by looking for identifiers explicitly renamed by developers. Thus, it is more likely that those identifiers are actually meaningful.

Internal Validity An important factor that influences DL performance is hyperparameters tuning. Concerning T5, for the pre-training phase we used the default T5 parameters selected in the original paper (Raffel et al. 2019) since we expect little margin of improvement for such a task-agnostic phase. For the fine-tuning, due to feasibility reasons, we did not change the model architecture (*e.g.,* number of layers), but we experimented with different learning rates-scheduler as did before by Mastropaolo et al. (2021). For the other two techniques we relied on the parameters proposed in the papers presenting them.

External Validity While the datasets used in our study represent hundreds of software projects, the main threat in terms of generalizability is represented by the focus on the *Java* language. It is important to notice that the experimented models are language agnostic, but would require the implementation of different tokenizers to support specific languages.

7 Conclusion

We presented a large-scale empirical study aimed at assessing the performance of data-driven techniques for variable renaming. We experimented with three different techniques, namely the *n*-gram cached language model (Hellendoorn and Devanbu 2017b), the T5 model (Raffel et al. 2019), and the Transformer-based model presented by Liu et al. (2020). We show that DL-based models, especially when considering predictions they generate with high confidence, represent a valuable support for variable rename refactoring. Our future research agenda is dictated by the implications discussed in Section 5.1.

Data Availability All the code and data we used in our study is publicly available in our replication package (Replication package 2022). In particular, we provide: (i) the code needed to train the models, (ii) the datasets we built for training, validating, and testing the models, (iii) all predictions generated by the three different approaches, and (iv) our trained models.

References

Allamanis M, Barr ET, Bird C, Sutton C (2014) Learning natural coding conventions. In: Proceedings of the 22nd ACM SIGSOFT International symposium on foundations of software engineering, pp 281–293

Allamanis M, Barr ET, Bird C, Sutton C (2014) Learning natural coding conventions. In: Proceedings of the 22Nd ACM SIGSOFT International symposium on foundations of software engineering, ser. FSE 2014, pp 281–293

Alon U, Sadaka R, Levy O, Yahav E (2020) Structural language models of code. In: Proceedings of the 37th International conference on machine learning, ser. Proceedings of machine learning research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul, pp 245–256

Alon U, Zilberstein M, Levy O, Yahav E (2019) Code2vec: Learning distributed representations of code. Proc ACM Program Lang 13(POPL):1–29

Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, Agarwal S, Herbert-Voss A, Krueger G, Henighan T, Child R, Ramesh A, Ziegler D, Wu J, Winter C, Hesse C, Chen M, Sigler E, Litwin M, Gray S, Chess B, Clark J, Berner C, McCandlish S, Radford A, Sutskever I, Amodei D (2020) Language models are few-shot learners. In: Advances in Neural Information Processing Systems, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., pp 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

Bruch M, Monperrus M, Mezini M (2009) Learning from examples to improve code completion systems. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ser. ESEC/FSE 2009, pp 213–222

Butler S, Wermelinger M, Yu Y, Sharp H (2009) Relating identifier naming flaws and code quality: An empirical study. In: 2009 16th Working Conference on Reverse Engineering. IEEE, pp 31–35

Butler S, Wermelinger M, Yu Y, Sharp H (2010) Exploring the influence of identifier names on code quality: An empirical study. In: 2010 14th European Conference on Software Maintenance and Reengineering. IEEE, pp 156–165

Caprile B, Tonella P (2000) Restructuring program identifier names. In: icsm, pp 97–107

Carter B (1982) On choosing identifiers. ACM Sigplan Notices 17(5):54–59

Chatterjee S, Juvekar S, Sen K (2009) sniff: A search engine for java using free-form queries. In: International conference on fundamental approaches to software engineering. Springer, pp 385–400

Chen M, Tworek J, Jun H, Yuan Q, Pinto HPdO, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G et al (2021) Evaluating large language models trained on code, arXiv:2107.03374

Ciniselli M, Cooper N, Pascarella L, Mastropaolo A, Aghajani E, Poshyvanyk D, Di Penta M, Bavota G (2021) An empirical study on the usage of transformer models for code completion, IEEE Transactions on Software Engineering

Ciniselli M, Cooper N, Pascarella L, Poshyvanyk D, Di Penta M, Bavota G (2021) An empirical study on the usage of bert models for code completion. In: 18th IEEE/ACM International conference on mining software repositories MSR

Conover WJ (1998) Practical Nonparametric Statistics, 3rd. Wiley, New York

Corazza A, Di Martino S, Maggio V (2012) linsen: An efficient approach to split identifiers and expand abbreviations. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM). IEEE, pp 233–242

Corbo F, Del Grosso C, Di Penta M (2007) Smart formatter: Learning coding style from existing source code. In: 2007 IEEE International Conference on Software Maintenance. IEEE, pp 525–526

CugLM Model (2020) https://github.com/LiuFang816/CugLM

Dabic O, Aghajani E, Bavota G (2021) Sampling projects in github for MSR studies. In: 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021. IEEE, pp 560–564

Daka E, Rojas JM, Fraser G (2017) Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In: Proceedings of ISSTA 2017 (26th ACM SIGSOFT International Symposium on Software Testing and Analysis). ACM, pp 57–67

Feldthaus A, Møller A (2013) Semi-automatic rename refactoring for javascript. ACM SIGPLAN Not 48(10):323–338

Foster SR, Griswold WG, Lerner S (2012) witchdoctor: Ide support for real-time auto-completion of refactorings. In: 2012 34th International Conference on Software Engineering (ICSE). IEEE, pp 222–232

Gage P (1994) A new algorithm for data compression. C Users J 12(2):23?38

Github copilot (2021) https://copilot.github.com

Github website (2008) https://www.github.com/

Grissom RJ, Kim JJ (2005) Effect sizes for research: A broad practical approach, 2nd ed Lawrence Earlbaum Associates

Guerrouj L, Di Penta M, Antoniol G, Guéhéneuc Y-G (2013) Tidier: an identifier splitting approach using speech recognition techniques. J Softw:, Evol Process 25(6):575–599

Gvero T, Kuncak V, Kuraj I, Piskac R (2013) Complete completion using types and weights. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'13, Seattle, WA, USA, June, vol 16–19, pp 27–38

Haque S, LeClair A, Wu L, McMillan C (2020) Improved automatic summarization of subroutines via attention to file context

Hellendoorn VJ, Devanbu P (2017) Are deep neural networks the best choice for modeling source code? In: Proceedings of the 2017 11th Joint meeting on foundations of software engineering, ser. ESEC/FSE 2017, p 763?773

Hellendoorn VJ, Devanbu P (2017) Are deep neural networks the best choice for modeling source code? In: Proceedings of the 2017 11th Joint meeting on foundations of software engineering, pp 763–773

Hill E, Fry ZP, Boyd H, Sridhara G, Novikova Y, Pollock L, Vijay-Shanker K (2008) Amap: automatically mining abbreviation expansions in programs to enhance software maintenance tools. In: Proceedings of the 2008 international working conference on Mining software repositories, pp 79–88

Holm S (1979) A simple sequentially rejective multiple test procedure, Scandinavian journal of statistics. pp 65–70

Howard GD (2021) Github copilot: Copyright, fair use, creativity, transformativity and algorithms

Høst EW, Østvold BM (2009) Debugging method names. In: Proceedings of ECOOP 2009 (23rd European Conference on Object-Oriented Programming). Springer, pp 294–317

Jablonski P, Hou D (2007) Cren: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. In: Proceedings of ETX 2007 (2007 OOPSLA workshop on eclipse technology eXchange). ACM, pp 16–20

Javaparser (2019) https://javaparser.org

Jin X, Servant F (2018) The hidden cost of code completion: Understanding the impact of the recommendation-list length on its efficiency. In: Proceedings of the 15th International conference on mining software repositories, pp 70–73

Karampatsis R, Sutton CA (2019) Maybe deep neural networks are the best choice for modeling source code. [Online]. Available arXiv:1903.05734

Kim S, Zhao J, Tian Y, Chandra s (2021) Code prediction by feeding trees to transformers. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering(ICSE). IEEE, pp 150–162

Lawrie D, Binkley D (2011) Expanding identifiers to normalize source code vocabulary. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM). IEEE, pp 113–122

Lawrie D, Feild H, Binkley D (2007) Quantifying identifier quality: an analysis of trends. Empir Softw Eng 12(4):359–388

Lawrie D, Morrell C, Feild H, Binkley D (2006) What's in a name? a study of identifiers. In: 14th IEEE International Conference on Program Comprehension (ICPC'06). IEEE, pp 3–12

Lin B, Nagy C, Bavota G, Marcus A, Lanza M (2019) On the quality of identifiers in test code. In: 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, pp 204–215

Lin B, Scalabrino S, Mocci A, Oliveto R, Bavota G, Lanza M (2017) Investigating the use of code analysis and nlp to promote a consistent usage of identifiers. In: 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, pp 81–90

Liu F, Li G, Zhao Y, Jin Z (2020) Multi-task learning based pre-trained language model for code completion. In: Proceedings of the 35th IEEE/ACM International conference on automated software engineering, ser. ASE 2020 Association for Computing Machinery

Mandelin D, Xu L, Bodík R, Kimelman D (2005) Jungloid mining: helping to navigate the API jungle. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June, vol 12–15, pp 48–61

Mastropaolo A, Aghajani E, Pascarella L, Bavota G (2021) An empirical study on code comment completion. In: 37th International Conference on Software Maintenance and Evolution ICMSE

Mastropaolo A, Pascarella L, Bavota G (2022) Using deep learning to generate complete log statements. In: 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022. ACM, pp 2279–2290

Mastropaolo A, Scalabrino S, Cooper N, Nader-Palacio D, Poshyvanyk D, Oliveto R, Bavota G (2021) Studying the usage of text-to-text transfer transformer to support code-related tasks. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021. IEEE, pp 336–347

Maven (2004) https://maven.apache.org

McNemar Q (1947) Note on the sampling error of the difference between correlated proportions or percentages. Psychometrika 12(2):153–157

Murphy-Hill E, Parnin C, Black AP (2011) How we refactor, and how we know it. IEEE Trans Softw Eng 38(1):5–18

Nguyen AT, Nguyen TT, Nguyen HA, Tamrawi A, Nguyen HV, Al-Kofahi J, Nguyen TN (2012) Graph-based pattern-oriented, context-sensitive source code completion. In: 2012 34th International Conference on Software Engineering (ICSE), pp 69–79

Niu H, Keivanloo I, Zou Y (2017) Api usage pattern recommendation for software development. J Syst Softw 129:127–139

Proksch S, Lerch J, Mezini M (2015) Intelligent code completion with bayesian networks. ACM Trans Softw Eng Methodol 25(1):3:1–3:31

Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou Y, Li W, Liu PJ (2019) Exploring the limits of transfer learning with a unified text-to-text transformer, arXiv:1910.10683

Reiss SP (2007) Automatic code stylizing. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp 74–83

Replication package (2022) https://github.com/antonio-mastropaolo/automatic-variable-renaming

Robbes R, Lanza M (2010) Improving code completion with program history. Autom Softw Eng 17(2):181–212

SEART github search (2021) https://seart-ghs.si.usi.ch

Scrml website (2019) https://www.srcml.org/

Svyatkovskiy A, Deng SK, Fu S, Sundaresan N, Intellicode compose: Code generation using transformer (2020). In: ESEC/FSE '20: 28th ACM Joint European software engineering conference and symposium on the foundations of software engineering, Virtual event, USA, November 8-13, 2020, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, pp 1433–1443

Svyatkovskiy A, Lee S, Hadjitofi A, Riechert M, Franco J, Allamanis M (2020) Fast and memory-efficient neural code completion

Svyatkovskiy A, Zhao Y, Fu S, Sundaresan N (2019) Pythia: Ai-assisted code completion system. In: Proceedings of the 25th ACM SIGKDD International conference on knowledge discovery & data mining, pp 2727–2735

Thies A, Roth C (2010) Recommending rename refactorings. In: Proceedings of the 2nd International workshop on recommendation systems for software engineering, pp 1–5

Thies A, Roth C (2010) Recommending rename refactorings. In: Proceedings of RSSE 2010 (2nd International workshop on recommendation systems for software engineering). ACM, pp 1–5

Tsantalis N, Ketkar A, Dig D (2020) Refactoringminer 2.0, IEEE Transactions on software engineering

Tu Z, Su Z, Devanbu P (2014) On the localness of software. In: Proceedings of the 22nd ACM SIGSOFT International symposium on foundations of software engineering, ser. FSE, vol 2014, pp 269–280

Tufano M, Drain D, Svyatkovskiy A, Deng SK, Sundaresan N (2020) Unit test case generation with transformers. [Online] Available: arXiv:2009.05617

Tufano R, Masiero S, Mastropaolo A, Pascarella L, Poshyvanyk D, Bavota G (2022) Using pre-trained models to boost code review automation. In: 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022. ACM, pp 2291–2302

Tufano M, Pantiuchina J, Watson C, Bavota G, Poshyvanyk D (2019) On learning meaningful code changes via neural machine translation. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May, vol 25–31, pp 25–36

Tufano R, Pascarella L, Tufano M, Poshyvanyk D, Bavota G (2021) Towards automating code review activities. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. IEEE, pp 163–174

Tufano M, Watson C, Bavota G, Penta MD, White M, Poshyvanyk D (2019) An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Trans Softw Eng Methodol 28(4):19:1–19:29

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need, Advances in neural information processing systems, vol 30

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Lu, Polosukhin I (2017) Attention is all you need. In: Advances in Neural Information Processing Systems 30, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., pp 5998–6008. [Online]. Available: http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

Wang Y, Wang W, Joty S, Hoi SC (2021) Codet5:, Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, arXiv:2109.00859

Watson C, Tufano M, Moran K, Bavota G, Poshyvanyk D (2020) On learning meaningful assert statements for unit test cases. In: ICSE'20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020, G. Rothermel and D. Bae, Eds. ACM, pp 1398–1409

Zhang C, Yang J, Zhang Y, Fan J, Zhang X, Zhao J, Ou P (2012) Automatic parameter recommendation for practical API usage. In: 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, pp 826–836

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

Antonio Mastropaolo is a Ph.D. student in the Faculty of Informatics at the Universit? della Svizzera Italiana (USI), Switzerland. He received his MSc. in Software System Security from Università degli studi del Molise, Italy, in July 2020. His research interests mainly include the study and the application of deep-learning techniques to foster code-related tasks. More information available at: https://antoniomastropaolo.com/.

Emad Aghajani is a postdoctoral researcher at the SEART research group, at Software Institute, the Universit? della Svizzera italiana (USI), Switzerland. He finished his Ph.D. studies in Computer Science in 2020 under the supervision of Prof. Michela Lanza and Prof. Gabriele Bavota at USI. He received his M.Sc. in Software Engineering from Sharif University of Technology, Iran, in 2016. His research interests include software evolution, software maintenance, mining software repositories, and empirical software engineering. More information is available at: https://emadpres.github.io/.

Luca Pascarella is a researcher at ETH Zurich, Switzerland. He received his Ph.D. in Computer Science from TU Delft, The Netherlands, in 2020. His broader mission aims at leveraging the heterogeneous data recorded during engineering processes to lighten human-centered tasks through data-driven algorithms. His focus is on empirical software engineering, code review, and computer-vision algorithms. He received an ACM SIGSOFT Distinguished Paper Award at MSR 2017 and a Best Paper Award Honorable Mention at CSCW2018. More information is available at: https://lucapascarella. com/.

Gabriele Bavota is an Associate Professor at the Faculty of Informatics of the Universit? della Svizzera italiana (USI), Lugano, where he is part of the Software Institute and he leads the SEART research group. He received the PhD in Computer Science from the University of Salerno, Italy, in 2013. His research interests include software maintenance and evolution, code quality, mining software repositories, and empirical software engineering. On these topics, he authored over 140 papers appeared in international journals and conferences and has received four ACM Sigsoft Distinguished Paper awards at the three top software engineering conferences: ASE 2013 and 2017, ESEC-FSE 2015, and ICSE 2015. He also received the best/distinguished paper award at SCAM 2012, ICSME 2018, MSR 2019, and ICPC 2020.

He is the recipient of the 2018 ACM Sigsoft Early Career Researcher Award for outstanding contributions in the area of software engineering as an early career investigator and the principal investigator of the DEVINTA ERC project.

More information: https://www.inf.usi.ch/faculty/bavota/.