Building Deduplicated Model Repositories to Assess Domain-Specific Languages Evolution

Alexandre Lachance
lachaa2@mcmaster.ca
McMaster University, McSCert
Hamilton, Ontario, Canada

Sébastien Mosser
mossers@mcmaster.ca
McMaster University, McSCert
Hamilton, Ontario, Canada

ABSTRACT

Software evolution and maintenance is a real challenge in modern software engineering. In the context of model-driven development, which heavily rely on interconnected (meta-)models, tools and generators, evolving both models and their associated meta-models is particularly complex. This issue is also prevalent in language engineering, where evolving a language’s grammar or semantics must remain consistent with the pre-existing models. In this paper, we explore how techniques inspired by repository mining can help a model designer/language engineer to build a deduplicated dataset of existing models available in open source repositories. Deduplication is essential to ensure the evolution made on the meta-model/language can be efficiently assessed. We apply the method to the P4 language, an industrial domain-specific language (Intel, Linux foundation) used to model software defined networks.

CCS CONCEPTS

• **Software and its engineering** → **Software development techniques**; **Open source model**; *Empirical software validation*; *Compilers.*

KEYWORDS

Compiler, DSL, Model, Mining, Evolution

ACM Reference Format:
Alexandre Lachance and Sébastien Mosser. 2024. Building Deduplicated Model Repositories to Assess Domain-Specific Languages Evolution. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion ’24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3652620.3688338

1 INTRODUCTION

In the software engineering world, Git is the industrial standard for version control. It enables development teams across the world to collaborate on software projects. A lot of these projects are open-sourced and available to everyone through *Git forges* like *GitHub*, *GitLab*, and *Bitbucket* to name a few. Many of these *forges* provide Application Programming Interfaces (APIs) to query projects, authors, issues, etc. This provides researchers with easy access to a vast amount of data. For example, on *GitHub* alone, 52 million new public repositories were created in 2022 [20]. These projects can provide incredibly deep insights into the software field, as advocated by the Mining Software Repositories research community.

Extracting information from projects can be complex since many pitfalls must be avoided. The reason for that is the fact that software projects are inherently very complex structures. For example, inconsistent coding practices, the evolution of projects over time, the large size of some projects, the lack of documentation, and duplicate projects can all be challenges when analyzing *Git* projects.

Interestingly, Model-Driven Engineering and model evolution trigger new challenges when mining repositories. This is mainly due to the domain-specific aspect of MDE tools and models, which makes them less mainstream and, as such, different from mainstream applications using fashionable languages or frameworks. In this paper, we will focus on the challenge of deduplication, a prototypical example of a situation that exists in classical mining but is amplified by the specificities of working with the evolution of MDE models or Domain-Specific Languages (DSLs).

This paper is organized as follows. In Section 2, we motivate the need for specific deduplication mechanisms when mining artifacts in an MDE/DSL context. Section 3 focuses on the related work regarding mining and deduplication, describing the different inspirations used in this work. Section 4 describes the solution we defined, as well as its implementation. In section 5, we apply the proposition to the P4 language, an industrial DSL and de facto standard for implementing Software-Defined Networks (SDN), to validate it at scale. Section 6 sketches some perspectives of this work, and finally, Section 7 concludes this paper.

2 MOTIVATING SCENARIOS

In this section, we describe two scenarios exemplifying how the evolution of both (i) models and (ii) the associated tooling is impacted by the existence of duplicates. Based on these scenarios, we express a “problem statement” to emphasize the importance of dataset deduplication when working with model-driven tools.

2.1 Model(s) Evolution

A trending research field in the MDE community is to leverage Machine Learning (ML) for several purposes, to ultimately help model designers with, *e.g.*, model completion and analysis. In this context, research can rely on pre-trained ML models (*e.g.*, GPT [10]), fine-tune such models with domain-specific elements, or train their own from scratch [18]. Building a dataset of MDE models to support training is crucial for the last two options. Moreover, for all these

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3688338

options, building a dataset to support the validation of the ML-based tool is also essential.

When encountered for the first time, this might seem like an easy problem to solve; however, it is far from trivial, especially when model evolution is taken into account. Some research has been done on this specific subject and found that some results could be inflated by up to 100%. Although the complete effects of duplicates in this type of dataset are still not entirely understood [4][21], it allows researchers and industry practitioners to provide less biased data to train ML models. So, when building your dataset, if you aggregate all the MDE models you can find from a forge, you might end up with duplicates and, as a consequence, train or validate your approach incorrectly.

This situation is emphasized by how designers interact with their MDE models and implement their evolution. In an ideal world, software modellers would rely on a version control system (VCS, e.g. Git, SVN) to version their system and, as such, record their evolution. However, as classical tools are not really supported by VCS (which only provides a syntactical way of handling conflicts), it is common for software modellers to rely on copy-paste and clones of their models, recording only the final results in the VCS without version history. As such, classical methods to identify duplicates cannot be used, as they rely on VCS’ mechanisms to support internal traceability to decide which file is a duplicate of another one.

The naive solution to this problem would be to compare every single model of the projects in the dataset to every single other one. Due to the complex nature of code repositories, this is computationally unfeasible: for a set of only 200 MDE models, one would have to compute $19,900$ comparisons ($O(n^2)$). Even if syntactical comparisons can be fast (*e.g.*, calling the `diff` command), this method cannot be used on models, as two models can be duplicates while not holding the exact same syntactical structure (*e.g.*, a set of model elements not recorded in the same order when serialized as XML). Semantic diffs are usually computationally intensive, as they are, in the end, a specialization of the graph isomorphism problem, which is NP-intermediate. Even if it is possible to solve it in quasi-polynomial time using heuristics and problem restriction (e.g., GumTree provides semantic diffs on some programming languages in $O(n^2)$ [8]), identifying two models as a duplicate from each other is a computational-intensive task.

Summary. *In a context where VCS cannot be leveraged to their full potential and duplication detection is cost-intensive, it is crucial to reduce the search space of duplicates when building a dataset of MDE models to avoid bias.*

2.2 Tooling Evolution

This scenario is the dual of the previous one. In this case, we are not necessarily interested in the evolution of the MDE models but in the tooling used to support them. For example, consider the Docker ecosystem [7]. This tool creates and deploys containers, providing a de facto standard for modern application deployment. At the heart of the ecosystem, Docker relies on a DSL used by developers to model `Dockerfile`(s). Such files model how a given Docker image is built in a prescriptive way. For example, start from an Ubuntu image, install Java 22, copy a JAR file from the local computer to the image, mount a file system directory from the host to the image, and run a given command to start the application. This DSL is widely used, as it was first launched in 2013. The tooling relying on the DSL has substantially evolved over the last 11 years[1]. Consequently, what was considered a practice at a given time can have evolved into an anti-pattern later on [5].

An interesting aspect of these successful DSLs is that they often rely on tutorials to support their success. In the case of Docker, “Docker Birthday” tutorials are released yearly. These tutorials have been instrumental in Docker’s success, accelerating the growth of people who wanted to learn how the platform works. But, sometimes, for logistics reasons, participants cannot always leverage the VCS when attending these tutorials (e.g., flaky internet connection at a conference venue). As a consequence, like in the previous scenario, they might obtain the source code used initially by the tutorial on a USB stick and finally publish their code to the forge as soon as they can access a better internet connection.

As a consequence, if one wants to evaluate the retrocompatibility of a tool to detect anti-patterns in Dockerfiles, one will have to sort the wheat from the chaff and identify among the thousands of Dockerfiles available in the different forges which ones come as an immediate outcome of a “birthday” event and which ones are genuinely independent artifacts. These artifacts might rely on something other than the classical *fork* mechanism offered by VCS to trace repository origins.

Summary. *In a context where the success of a DSL is enforced by a large number of easily accessible tutorials that developers are encouraged to publish, it is crucial to be able to sort out all the codes coming as derivative from such tutorial setup not to introduce immediate imbalance in the collected MDE models.*

2.3 Problem Statement: Deduplication

The two previous sections illustrate scenarios where building a dataset of MDE models is necessary to assess “something,” such as how tooling or models react to an evolution-driven situation. In these scenarios, we focused on the evolution of models or the evolution of their associated tooling, but overall, the problem encountered is amplified by the very nature of model-driven artifacts: models available in a scattered way and often originating from the same set of sources.

If leveraging the artifacts available in the open-source community has become a common way of approaching quantitative evaluation, it might not be immediately suitable when evaluating MDE artifacts. Not only do duplicates lead to unnecessary redundancy, consuming and wasting valuable storage and computational resources, but they also compromise the result of subsequent research that is based on them.

Based on our motivating scenarios, we identified the following characteristics of the problem:

- Models to be collected might originate from a set of common sources (*e.g.*, tutorials)
- Version Control Systems (*e.g.*, Git) tooling is not always used adequately due to the nature of the artifacts
- There is a need to reduce the size of the dataset collected as tools working on MDE models (*e.g.*, diff, model checking, compilers) can be resource-intensive

[1]https://docs.docker.com/build/dockerfile/release-notes/

- Assessing the retrocompatibility of tools or models is a classical task in MDE and DSL engineering.

As such, we have to refine classical mining techniques to consider these characteristics. We should assume that a "regular" contribution to a forge exists: user can create *new* repositories to share their models, *pushing* their code to it. A repository can also be obtained as a *fork* of an existing repository, *i.e.*, the classical way of expressing a divergence from a common origin in a VCS. Finally, the specificity of MDE artifacts triggers the last situation, where people classically rely on sharing artifacts outside of the VCS before sharing them, breaking these existing links.

The approach proposed in this paper adapts state of the art (which focuses on the two points) to support the last situation, which is a bad practice in regular development but quite common when working with MDE models and DSLs. We summarize in Figure 1 these different kind of interactions.

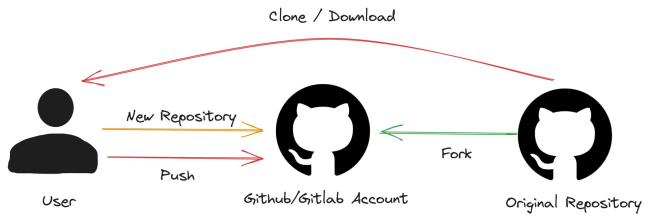


Figure 1: Types of Duplication in Model Repositories

3 RELATED WORK

In exploring the subject of deduplication of software repositories, it became evident that little work had been done on the subject yet. There is, however, a lot of previous work on code duplication detection. This section will discuss the few truly related works and how the tangentially related work can help us with this problem.

3.1 Code duplication

Some work has been done to identify the issue of duplicate code. For example, a study found that 70% of files on *GitHub* were duplicates of other files [11]. More relevant to this paper, the same study also found that between 9% and 31% of projects were made up of at least 80% of duplicate files [11]. While this study [11] doesn't address the problem of complete project duplication, it may hint at it. More focus studies are required to understand the scope of the issue better.

Furthermore, the impact of code duplication cannot be ignored. There has also been work on analyzing the problem of duplicate code when training ML models. A study found that some results could be inflated by up to 100% when models are trained on a dataset containing duplicates as opposed to a deduplicated one [4]. These results show the importance of addressing the duplication issue when working with code data-trained ML models. It is essential to know that duplicates only affect training in some fields and contexts. A study on malware detection models found that duplicates had little effect on the efficiency of the final system [21]. Further study on whether duplicate impact ML models trained on projects (or repositories) might be necessary. However, the impact of empirical tool testing on duplicates cannot be denied due to the bias introduced in the results.

3.2 Repository similarity

Another essential part of this paper and the proposed approach is how the similarity between different projects is calculated. Some projects might not be 100% duplicated and still be considered duplicates depending on the application of the dataset. It is then essential to find a set of metrics to evaluate similarity and set a threshold for these metrics on what is considered a duplicate. The state-of-the-art tool for this is *CrossSim* [14]; it is a tool that enables computing the similarity of different *Open Source Software* (OSS) projects. However, the focus is not on finding duplicates. It was made to make it easier to find similar projects, and it is not suitable for this use case. A major limitation of this tool, and similar ones, is its impracticality for general deduplication. Most, if not all, approaches focus on projects with a singular programming language: *Java* [14]. Modern-day projects use, on average, five programming languages per project and *Java* is only a part of it [12]. We then need to find a quick and language-agnostic way to compare projects for our approach.

There is, however, another approach that is interesting for our use case. This approach vectorizes repositories and all associated data into comparable embeddings. In *Natural Language Processing* (NLP), embeddings represent words or phrases as vectors that capture semantics and enable more accessible computational analysis. This has been used successfully for the deduplication of complex texts [9]. The same principle can be used for repositories. It has already been implemented by Rokon *et al.* [17] and has proven very effective. Their approach captures the semantics of the associated metadata, the structure of the repository, and the entire source code [17]. If the generated vectors genuinely represent the full semantics of a repository, as the authors propose, the distance between vectors would be an ideal metric for repository similarity. However, their code has yet to be available and is, therefore, unusable in our case. The reproduction of such an approach falls far from the scope of this paper in terms of complexity and the size of the work.

3.3 Repository deduplication

There has been some work on the specific subject of *Git* repository deduplication. It mainly consists of a single paper by *Spinellis et al. (2020)* [19], in which the authors found 30 thousand project duplicates out of 1.8 million projects. The approach consisted of geometric mean-based grouping and denoising of project clumps [19]. They used various metadata points like stars, *Git* history, fork information, etc. However, their approach has a lot of limitations and problems. First of all, the authors only focused on *GitHub* when there are a lot of different forges available that could provide a variety of projects. This might be fine on a large scale, but the extra data provided by other forges could help make the final dataset more versatile and helpful when working on a smaller scale. Secondly, their approach could have been clearer and depended on much manual work to denoise and clean the data. While this led to better results, the approach took time to reproduce, even on a smaller scale. Also, the liberal approach to denoising is not applicable in smaller-scale scenarios since it would remove many potential candidates. Thirdly, their method only examined surface-level metadata about each project and never examined the projects' actual content. For this reason, the authors only found duplicates with common *Git* histories and fork relationships. Because of this, many duplicates were likely missed and are still present in the final dataset. Metadata is also unreliable; for example, they used stars as a metric to determine attractor projects. One study looked at the correlation between *GitHub* stars and code quality, and the authors were unsuccessful in linking the two [13]. Lastly, their choice of technology could have been more optimal. They used a relational database to represent graph data, leading to convoluted SQL queries. However, We can draw some inspiration from this paper to build our own approach, notably using *Git* history and fork data.

4 PROPOSED SOLUTION

This section describes the complete proposed approach in detail, including its strengths and weaknesses. The approach is structured as a pipeline, composed of six sequential steps, spanning from the creation of the initial dataset to the final deduplicated dataset (you can refer to Fig. 2 for an overview of the entire solution pipeline).

The approach relies on creating a relationship graph to reduce duplicate search space at each step. That is the sole focus of the three steps that follow the initialization of the dataset. The final two steps are used to delete (or flag) duplicates. The *Simple Duplicate Deletion* step uses metadata to delete projects that are obvious duplicates, further reducing the number of project pairs that need to be compared in the final step. The final step (*Full Similarity Metrics Deletion*) is a very computationally intensive step because of the intrinsic complexity of software repositories. This is why the whole approach focuses on reducing the search space before reaching it. It involves a complete comparison of project pairs.

The rest of the section will focus on describing the process behind each step and the thought process behind all those decisions.

4.1 Initial Dataset

To start the construction of the dataset, we first need to acquire the initial repositories. To do so, we use the different REST APIs the different forges provide. The two main *Git* forges that provide APIs are *GitHub* and *GitLab*. Using their query features, we can build a dataset that focuses on whatever characteristic of the repositories we want. This could be the programming language used, the number of stars, the license used, the description of the project, etc. However, since this approach relies on the existing relationships between the different projects, it is better to target a known community. For example, targeting the community that uses a specific programming language would be better since there will be clearer links between the different projects.

There are some limitations with the different APIs that make collecting repositories difficult. For example, the *GitHub* API only allows up to 1000 repositories per query. This can be circumvented by splitting the query into multiple queries. The easiest way to do so is to split the whole span of your query into multiple date ranges. Splitting can be implemented to be done automatically. The *GitLab* API is also very unstable, often returning errors. This can make collecting data more strenuous. To prevent missing relationships with the following steps, if the metadata of a repository dictates that it is a fork, but the parent is absent in our collected data (that can be for various reasons), we add these missing projects to our data. These APIs only return the metadata of the repositories that match our query to us.

The schema of this metadata is very similar across forges, but some terminology varies. Therefore, we need to do some schema matching before proceeding. Some fields do not serve any purpose for deduplication, we delete those. We also want to keep track of which forge each repository comes from, so we add a field for that (see Fig. 3).

For our approach, we also need the actual code and full *Git* history of each project. To do so, we need to clone all of these repositories. This can be a problem since, depending on the number of repositories, it can take up a lot of storage space. Luckily, because of the nature of code (text), most repositories are only a few megabytes. In case of storage constraint, this could be optimized by deleting irrelevant files for each project, such as *PDF* documents or *ZIP* archives.

After all these steps, we have a unified multi-forge dataset ready to be deduplicated.

4.2 Forks

This is the first step of the pipeline that starts building the relationship graph and, therefore, starts reducing the final search space. It addresses the first type of duplicate, forks (see Fig. 1).

We start by loading all the metadata that we get from the first step as nodes in a graph. The different types of relationships will be represented as different edge types between nodes (repositories/projects). In this case, it is better to use a graph database. It will make the data more accessible to work with since the database will be adapted to the structure.

This step involves connecting the projects that have been forked to their parent. It is simple since the different repositories already link to their parent in their fields. After linking those, we can start to see a graph forming. We can also see which projects are "source" or central projects (projects that are parent to others and that are not the children of any other). These projects will be crucial to the next steps.

4.3 Git History

This step aims to add to the relationship graph by using the information given by *Git*, more specifically, the *Git* history. It addresses the second type of duplicates, projects that were cloned and then pushed as new projects (see Fig. 1).

Projects that are cloned from online forges keep their *Git* history. We can then use this information to link them to the original repository. To do so, we need to compare a project's first commit ID to the first commit ID of central projects found in the last step. We only need to compare them to central projects since projects that were forked from them will have the same starting commit history. Commit IDs are designed to be as globally unique as possible [2], there is then a very low chance of false positive. However, some projects could have diverted significantly after the first commit and would no longer be considered duplicates. This is why we still

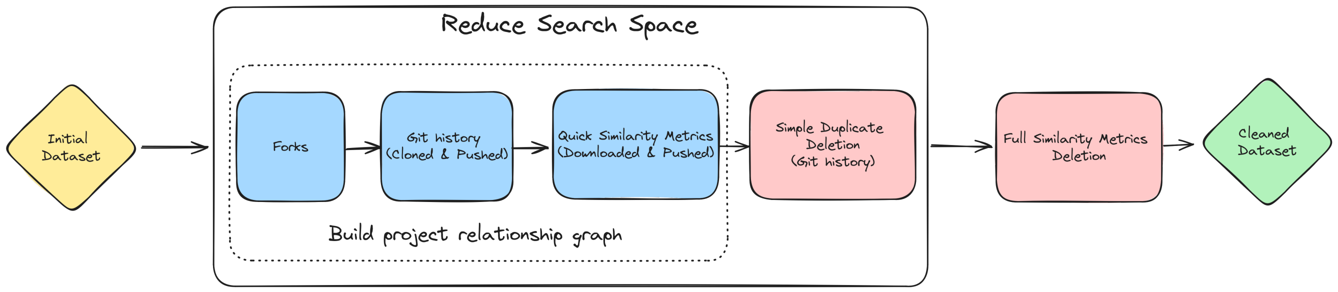

Reduce Search Space
Initial Dataset
Forks
Git history (Cloned & Pushed)
Quick Similarity Metrics (Downloaded & Pushed)
Build project relationship graph
Simple Duplicate Deletion (Git history)
Full Similarity Metrics Deletion
Cleaned Dataset

Figure 2: Diagram of the full solution pipeline

- **forge**: The project's forge.
- **id**: The project id.
- **name**: The project name.
- **full_name**: The project name with name space.
- **description**: The project description.
- **created_at**: The project creation date.
- **updated_at**: The last time the project was updated.
- **allow_forking**: If the project allows forking.
- **forks_count**: The number of times this project was forked.
- **stars**: The project's amount of stars.
- **owner_id**: The id of the project's owner.
- **owner_username**: The username of the project's owner.
- **fork**: If the project is a fork.
- **parent_id**: If fork, the project's parent id.
- **parent_name**: If fork, the project's parent name.
- **parent_full_name**: If fork, the project's parent full name.
- **parent_creator_id**: If fork, the project's parent creator id.

Figure 3: Repository Node Metadata Fields

do full project comparison at the last step within the relationship graph.

4.4 Quick Similarity Metric

This step aims to finalize the relationship graph using additional surface-level data about the project. In this case, we use the name of the project and the project structure. It addresses the second type of duplicates, projects that were downloaded/copy-pasted and then pushed as new projects (see Fig. 1). In this case, we can no longer rely on information provided by the *Git* forge or *Git* itself. This is because the projects are downloaded directly from a forge, they lose all *Git* related information and are downloaded as an archive (typically a *ZIP* file). Therefore, we can only base this step on the information contained in the project itself. To identify the project pairs we need to compare, we compare central projects to disconnected projects. In doing so, we risk missing duplicates of other projects that are already in the relationship graph. However, it is less likely that a project was duplicated from a child project than a more popular central graph. We can partly circumvent this limitation by setting a lower similarity threshold in this step and relying more on the last step to weed out the true duplicates.

It would be too computationally expensive to compare all central and disconnected projects using their file content. This is the reason we only use surface-level information for this step. We identified two metrics to quickly calculate the similarity between project pairs: the similarity between the names and the similarity between the file trees. If better ones are identified (in a framework approach), these metrics could easily be swapped out for different ones.

The name of a project represents the purpose of it. It then makes sense that the name would be similar if the code is the same. There are multiple ways to calculate the similarity between two project names. The most accurate is to use natural language processing (NLP) techniques, like word embeddings, that capture the semantics of the words. That would allow completely different names with similar meanings to have a high similarity score. However, incorporating this into our solution involves adding a lot more complexity and computations. Instead, we chose to go with the edit distance between the names, precisely the Levenshtein distance. We also normalized the distance between 0.0 and 1.0 to make it comparable across project pairs with varying project name lengths. This technique is much more straightforward to implement and requires no additional computational resources. It, however, does not capture semantics and is less accurate.

The file tree structure of a project usually reflects the composition of it. Projects with a similar structure, while the content of the files may vary, are more likely to be duplicated. Therefore, by comparing the file trees, we can capture the surface-level project similarity without diving into the actual file content and code, making this operation very fast and efficient. The best way to compare trees is to calculate the distance between trees. Tree edit distance is the minimum number of operations (add, delete, replace) needed to transform one tree structure into another. The state-of-the-art way to calculate this is using the *APTED* algorithm [15][16]. To get a similarity score out of this, we normalize the distance between 0.0 and 1.0.

To combine these two scores into a single similarity score, we weigh both of the values and add them up ($\omega_0 \cdot D_{tree} + \omega_1 \cdot D_{name}$). If this score is over a configurable threshold, we add them to our relationship graph.

We also add a heuristic to reduce the amount of computations and time needed by a very significant amount. When project trees contain a lot of files and are structurally very different from each other, calculating the *tree edit distance* becomes very expensive. The similarity is calculated only if the file count ratio between the two projects ($\frac{max_(n_a,n_b)}{min_(n_a,n_b)}$, n being the number of files in a project) is below a specified threshold (θ). The threshold ensures that the similarity computation is only performed for project pairs with a reasonable file count ratio, preventing unnecessary and computationally expensive comparisons for projects significantly different in size. The entire quick similarity algorithm can be seen in equation 1.

4.5 Simple Duplicate Deletion

This step is the first of two that deletes (or flags) duplicates, but it also reduces the search space of the next and last steps. This deletion is done using the *Git* history of a project. It compares a repository with its direct parent in the relationship graph. The only thing that it looks at is the commit history of both. If the latest commit ID of the child is present in the parent's history, then no changes have been made since the project was forked or cloned and pushed as a new project. It is then a duplicate and is deleted (or flagged), reducing the search space one last time.

4.6 Full Similarity Metrics Deletion

Now that the search space has been reduced as much as possible, we can now proceed to a complete project comparison. We look within the relationship graph to identify pairs that need to be evaluated for duplicate identification. This graph contains all projects that come from others and, therefore, contains all potential duplicates. The pairs are then picked by looking at every child-parent pair in the graph.

We want our approach to be language-agnostic and fast. This rules out state-of-the-art techniques for software project comparison. Most of them are either language-specific (mainly *Java*) and/or not made to be used at this scale. The next best technique that could be considered is an NLP approach. However, because the number of file comparisons that need to be done per project can be in the hundreds or thousands, depending on the project pair, this is not computationally feasible and adds a lot of complexity. We then need to look at simpler methods to compare projects.

To simplify the problem, instead of comparing the project as a monolith, we split it up by comparing the similarity of each pair of files that have the same path in the two different projects. To prevent unnecessary computations, we give the option to only compare specific types of files. We can then do the average of all the file pairs similarity scores (see Eq. 2). If a file exists in one project and not the other, the similarity score for that file pair is 0.

$$Similarity = \frac{S_1 + S_2 + ... + S_n}{n} \tag{2}$$

where:

$$S_1 + S_2 + ... + S_n = \text{Sum of scores of } n \text{ files}$$
$$n = \text{Number of files}$$

We use a sequence matching algorithm to evaluate the similarity between the content of two files. More specifically, we use Python standard library's "difflib.SequenceMatcher", an improved version of the Gestalt Approach [3] by Ratcliff and Obershelp. This algorithm compares two sequences by recursively finding the longest common subsequence and assigning similarity scores based on the lengths of the common subsequences between the sequences. It has a linear best-case complexity and a quadratic complexity for the worst-case. There are better algorithms for the use case, but given its speed and simplicity, it is a good compromise. Given better resources, an NLP approach could be considered and swapped, given the framework approach.

The source code implementing the approach described in this paper is publicly available on GitHub[2].

5 EXPERIMENTS

To validate the proposed solution, we set up an experiment using the *P4* programming language ecosystem. We chose this system since it is small (around 2000 public projects), which allows us to experiment and iterate on our solution more quickly. This section is divided into the experimental setup (software and hardware), results, and analysis. Finally, at the end of this section, we illustrate how the deduplication approach was used to support the evolution of the DSL compiler frontend.

5.1 Experimental Setup

In terms of hardware, experiments were done on a computer running Linux, equipped with a 9th-gen Intel i5-9600K (6 cores, 6 threads) CPU and 32 GB of RAM. This is consumer hardware, and the results could be enhanced by improving the hardware to a more professional level. However, the impact would only be in terms of computation time.

For software, multiple choices were made. As the general implementation programming language, *Python* was used for its simplicity and useful standard library. *Javascript* was also used to query the *GitHub* since the language has libraries developed by *GitHub* themselves. As we recommended in our approach, we chose a graph database for the database. We went for *Neo4J* for multiple reasons. Firstly, it has a free-to-use community version that implements every feature we need. Secondly, it also implements many algorithms by default, like the Levenshtein distance, that we would have needed to implement otherwise. Thirdly, the query language, *Cypher*, is straightforward to use and adapted to graph queries, making working with the graph simple. Lastly, it implements some visualization technologies by default, which are very useful for better understanding the data and showing how the approach is working.

5.2 Results and Analysis

5.2.1 Building the dataset. The dataset comprises 2610 repositories sourced from both *GitHub* and *GitLab*. For *GitHub*, 2529 repositories were found with a query for *P4* programming language projects. We then found that these projects also referenced 33 repositories that were absent as their parent, so we added them. For *GitLab*, the API identified 72 repositories, but we were only able to retrieve

[2]https://github.com/AlexandreLachanceGit/git-deduplication

$$\text{Similarity}(n_a, n_b, D_{tree}, D_{name}, \theta) = \begin{cases} \omega_0 \cdot D_{tree} + \omega_1 \cdot D_{name} & \text{if } \frac{\max(n_a,n_b)}{\min(n_a,n_b)} < \theta \\ 0.0 & \text{otherwise} \end{cases} \tag{1}$$

where:

$$\begin{aligned} \omega_0 &= \text{weight of the tree edit distance} & n_a &= \text{number of files in project a} \\ D_{tree} &= \text{normalized tree edit distance} & n_b &= \text{number of files in project b} \\ \omega_1 &= \text{weight of the name edit distance} & \theta &= \text{maximum project file ratio} \\ D_{name} &= \text{normalized name edit distance} & & \end{aligned}$$

a maximum of 48 before encountering an internal error that was difficult to debug. While *GitHub* projects vastly outnumber *GitLab* projects (see Fig. 4), this isn't a problem since they are both represented with the same schema in the dataset and won't be processed differently.

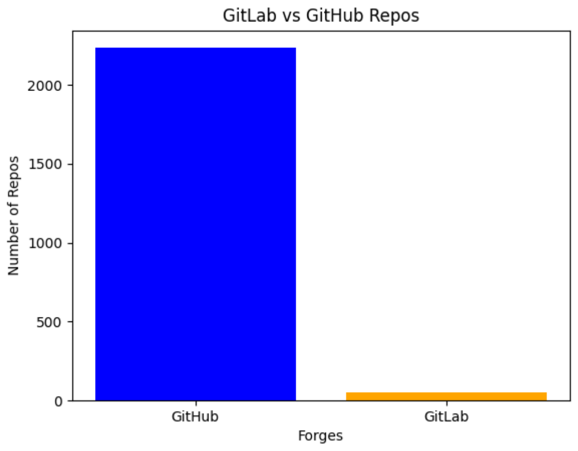


Figure 4: Graph of repository distribution across forges

5.2.2 Fork Relationships. The *Forks* step in the pipeline created a total of 1600 relationships in the relationship graph, resulting in it containing 1864 connected nodes. This means that a total of 71.41% of the repository nodes were connected. It highlights how interconnected the *P4* programming community projects are.

5.2.3 Git History Relationships. The *Git History* step in the pipeline added 71 new relationships in the graph, resulting in a total of 1904 connected nodes (72.95%). This step's execution time was only 7.5 seconds, which highlights its effectiveness and efficiency.

5.2.4 Quick Similarity Relationships. The *Quick Similarity* step in the pipeline created 9 new relationships, resulting in a total of 1913 connected nodes (73.30%). This step involved a substantial amount of computations. As shown in Fig. 5, the results follow something resembling an inverse-square relationship. This is in line with expectations. Given that most project pairs are likely to differ unless the dataset is saturated with duplicates, it is logical for the similarity scores to show a trend toward 0. The threshold at which a project was considered "similar" or a potential duplicate was set at 0.7, which resulted in 9 new relationships.

The problem with this step is the return on investment in computation time, as opposed to new relationships. Most relationships were already found efficiently using *Git* metadata in the previous steps. This step of the pipeline looks for outliers that do not have metadata linking them. This step, depending on the use case, could be skipped. It could also be improved by parallelizing the algorithm since this method of doing independent pair comparisons lends very well to parallelization.

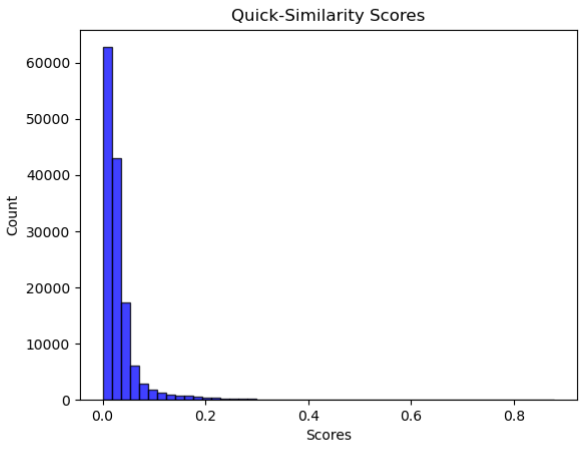


Figure 5: Graph of quick-similarity score distribution

5.2.5 Simple Duplicate Deletion. The *Simple Duplicate Deletion* step in the pipeline resulted in a total of 1412 projects flagged as duplicates. It involved comparing 1680 pairs *Git* history and the whole operation, a single *Cypher* query in the *Neo4j* database, was completed in only 87 milliseconds. This took down the final search space for the next step to 256 pairs.

5.2.6 Full Similarity Metrics Deletion. The final step in the pipeline found 115 new duplicate repositories in the dataset. This step's computation time is notably higher: 22 minutes to compare the 256 pairs left. It initially took a lot longer, but parallelizing the algorithm was straightforward, reducing the time to what it is now. The similarity score threshold was set at 0.75. This could be adapted based on the use case for the dataset (see Fig. 6). The system only compared files of type *Python*, *Bash*, and *P4* since those were the relevant plain text files. This saved computation by preventing the comparison of other file types like *PDF*s and *ZIP* archives.

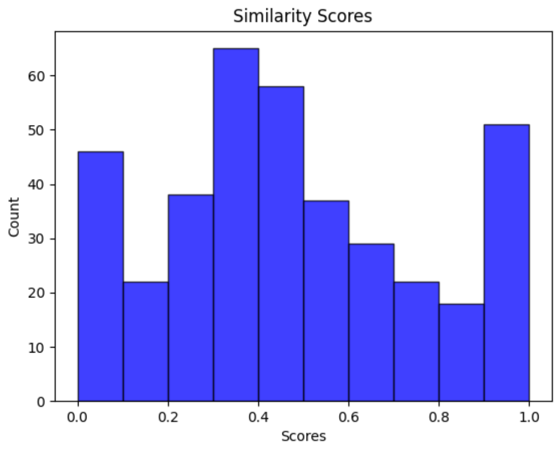


Figure 6: Graph of full-similarity score distribution

5.2.7 Total. After all the steps of the pipeline, 1527 duplicates were found. This represents 58.51% of the original dataset, showing the prevalence of duplicate projects on *Git* and the need for this solution. Overall, the deduplication took around 25 minutes on the previously specified hardware for this relatively small dataset, ignoring the initial dataset collection. Depending on parameters and the use case, this approach could prove difficult to justify for larger datasets. The number of project pairs to compare will increase exponentially, even if the pipeline aims to reduce the search space as much as possible.

5.3 Threaths to validity

The validation of our results presents several challenges that may affect the validity of our findings.

5.3.1 Internal Validity. One significant issue lies in the scale and thoroughness of our approach compared to previous studies. Earlier research worked with datasets containing millions of repositories, while our analysis was limited to thousands. They were also not as thorough. For one reason or another, the datasets used in these earlier works are no longer available online, as all copies seem to have been deleted. This prevents us from directly comparing our findings against a well-established ground truth.

5.3.2 Construct Validity. Manually verifying false negatives or identifying duplicate projects that were missed is an overly time-consuming task to complete within a reasonable timeframe. This is because there would be a total of 6,812,100 pairs to verify, and even with random sampling, it would be tough to have a representative sample that a person can evaluate. However, while looking at the data, we found that projects that were nested inside a repository (i.e. `tutorial/` vs `mytutorial/tutorial/`), even if the content of both directories are the same, were not found as duplicates. Some false negatives are then to be expected. We then focus on attempting to find false positives or repositories that were flagged as duplicates but were not. To do so, we randomly sample duplicate and original project pairs and then qualitatively assess whether the projects are duplicates. We evaluated 50 project pairs out of 286 qualitatively by looking at project structure and file content. No false positives were found within that sample. We found that the similarity scores reflected the similarity of the projects well. Projects that trended toward our threshold (0.75) had more differences, and those that trended toward 1.0 were nearly identical. However, files that were only present in one project were too heavily weighted in the result using our equation. This could be improved by weighing every similarity score by the number of lines of the file before calculating the average.

5.3.3 External Validity. The external validity of our findings may be limited by the specific focus on the P4 programming language ecosystem. The high duplication rate observed in this context may not generalize to other domains or programming languages. Future studies involving a broader set of programming languages are necessary to evaluate the generalizability of our approach.

5.3.4 Conclusion Validity. Although our qualitative assessment found no false positives, the potential for undetected false negatives and the bias introduced by unverified project pairs could influence the overall reliability of our findings. Future work should explore more robust sampling methods and develop automated tools to improve the efficiency and accuracy of duplicate detection across larger datasets.

5.4 Evolution assessment scenario: P4LSP

Despite being an industrial standard for software-defined networks, surprisingly few tools support the editing of P4 source code. As part of an industrial project involving Telus (the second-largest telecommunications operator in Canada), McSCert worked on defining a language server to make the development of P4 applications more accessible by providing an IDE plugin[3] supporting code completion, syntax highlighting, and classical refactoring operations.

As the legacy compiler (P4C) is written in C++, it is not compatible with mainstream language engineering tooling such as Xtext [6] or Langium [1]. Consequently, we had to port the implemented grammar into a new technology. More specifically, we ported the grammar initially defined using Flex/Bison[4] into Treesitter[5].

Porting grammar is a tricky task and is a prototypical example of our second motivating scenario: tooling evolution. In P4, the *frontend* part of the compiler represents 21, 000 lines of code, almost 10% of the entire source. When porting the grammar, we had to empirically validate that the new tool was retrocompatible with the existing P4 sources. Typically, one would reject an IDE plugin that does not identify errors that the compiler will identify, ending up with code that is valid in the IDE but rejected by the compiler. To validate our IDE plugin, we collected a dataset of P4 code and measured the ratio between the code successfully compiled by P4C

[3]https://marketplace.visualstudio.com/items?itemName=mcscert.p4lsp
[4]https://github.com/p4lang/p4c/tree/main/frontends
[5]https://github.com/ace-design/tree-sitter-p4

and the one the new frontend was accepting. We also determined that the new tool considered the invalid code for P4C invalid.

Like Docker, P4 followed the same trend of intensively relying on tutorials[6] to support developers' learning curve. As an immediate consequence, simply mining all P4 sources without deduplicating them resulted in a non-representative ratio. If a file were predominant in the dataset, failing at parsing, it would trigger a catastrophic coverage ratio (*e.g.*, "65% of legacy code cannot be compiled by the new tool!"), while it was, in the end, one syntactic construction used in a file duplicated a thousand times. For example, the `core.p4` file defines basic features (such as reading a packet from the network). Even if the compiler provides it as a standard include, we observed a practice of copy-pasting the file to the local repository, supposedly to ensure the build's portability for beginners. This results in this file being largely duplicated when mining the P4 source code without taking care of deduplication.

To assess the development of P4LSP, our IDE plugin to support P4 development, the team created a visualization benchmark as part of the project, represented in Figure 7. The benchmark identified the number of files compatible with P4C and the new tool (here named Treesitter). For each file part of the dataset, the benchmark indicated how many instances of this file was found in the original dataset, indicating its prevalence.

6 PERSPECTIVES

State-of-the-art Project Similarity Metrics. The solution proposed in this paper uses a very bare-bones technique to calculate the similarity of different projects as its last step. This is different from the state-of-the-art on the subject, as mentioned in the related works section. In future work and experiments, it would be interesting to attempt to deduplicate a dataset of *Java* repositories instead of a *P4* one. It would allow us to look at the approach's functions when better language tooling is available. Because of the framework approach and the fact that the rest of the deduplication pipeline is language-agnostic, it should be painless to swap out the current final similarity calculation for *CrossSim* [14] for example.

Repository Embeddings. There is interesting new research being done on the subject of vectorizing repositories [17]. These recent advancements could be very helpful in developing a new language-agnostic method for evaluating the similarity between repositories. This approach takes the full semantics of a repository and represents it using a vector; it considers the associated metadata, the repository's structure, and the entire source code [17]. Vector distance calculations are faster than the distance between raw file structures and file contents. It would, however, introduce more complexity to the solution since these embeddings need to be trained. Depending on the computational overhead of vectorizing different repositories, significantly reducing the search space, as presented in this paper, could still be relevant. Our approach allows us to reduce the number of projects that need to be compared; therefore, we would only need to vectorize these projects.

Broader Evaluation of the Amount of Duplicates. Previous research only found less than 2% of duplicates (30 thousand out of 1.8 million projects) [19]. At the same time, our approach allowed us to discover that 58.51% of our initial dataset was composed of duplicates (1527 out of 2610 projects). This leads to questions about whether *P4* programming projects (our chosen dataset) were inherently more prone to duplicates or if previous research had an overwhelmingly high amount of false negatives. Logically, it should be a mix of both since our approach, while being orders of magnitudes more computationally intensive, was significantly more thorough. A more representative sampling of different *Git* forges should allow a better evaluation and quantification of the breadth of the problem.

Impact of Repository Duplicates on Machine Learning Models. Lastly, another important research avenue that should be explored is the impact of repository duplicates on ML models. This is because there is an apparent lack of available studies on the subject. It is crucial to understand how the presence of duplicates could influence the performance, generalization, and overall accuracy of ML models applied to repositories. A better understanding of this would allow more focused research to negate the adverse effects of duplicates.

7 CONCLUSION

In this paper, we presented a multi-forge git repository dataset deduplication framework designed to support the assessment of MDE artifacts as they evolve. Our key finding is that by leveraging different types of relationships between projects, such as forks, cloned and pushed projects, and downloaded and pushed projects, we can significantly reduce the computational requirements for identifying duplicates. Specifically, our approach decreased the number of required full project comparisons from over 6 million to just 256, ultimately identifying 1527 duplicates, which represents 58.51% of the original dataset.

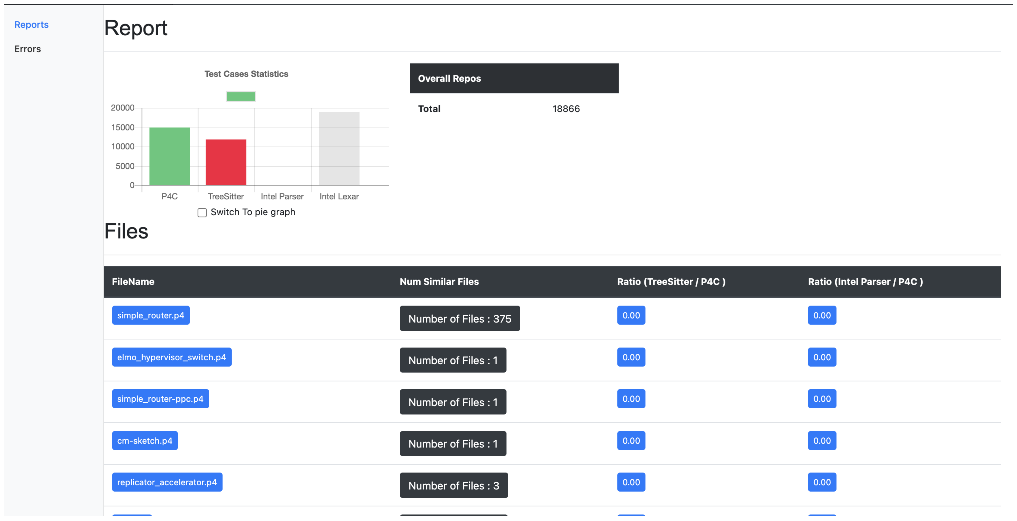
This finding underscores the prevalence of duplicate projects in software repositories and highlights the importance of addressing this issue to ensure the accuracy of empirical research. Moreover, our approach demonstrates that it is possible to create a scalable and efficient deduplication process that can be adapted to various programming languages and contexts.

For future work, researchers could explore applying this framework to other domains and programming languages, as well as investigate the impact of repository duplicates on ML models. Additionally, further refinement of similarity metrics could enhance the precision and applicability of the deduplication process, paving the way for more robust and reliable datasets in the field of software engineering.

ACKNOWLEDGMENTS

This work is supported by the *Natural Sciences and Engineering Research Council of Canada* (NSERC) thanks to a Discovery Grant (`RGPIN-2020-05791`), and an Alliance industrial project with Telus (`ALLRP-576653-22`). This project has also received support from Mitacs thanks to its Accelerate internship program (`IT32248`). The authors want to thank Noel Chungath Gregory and Maël Charpentier who participated in the implementation of the P4 benchmark. Thanks to Professor Fei Chiang for her help and guidance.

[6]https://github.com/p4lang/tutorials



Report

Overall Repos	
Total	18866

Files

FileName	Num Similar Files	Ratio (TreeSitter / P4C)	Ratio (Intel Parser / P4C)
simple_router.p4	Number of Files : 375	0.00	0.00
elmo_hypervisor_switch.p4	Number of Files : 1	0.00	0.00
simple_router-ppc.p4	Number of Files : 1	0.00	0.00
cm-sketch.p4	Number of Files : 1	0.00	0.00
replicator_accelerator.p4	Number of Files : 3	0.00	0.00

Figure 7: Benchmarking the P4C legacy compiler versus the new TreeSitter implementation

REFERENCES

[1] [n. d.]. Langium. https://langium.org/. Accessed: 2024-07-15.

[2] 2023. Git - Git Objects. https://git-scm.com/book/en/v2/Git-Internals-Git-Objects [Online; accessed 14. Dec. 2023].

[3] 2023. Pattern Matching: the Gestalt Approach. https://www.drdobbs.com/database/pattern-matching-the-gestalt-approach/184407970?pgno=5 [Online; accessed 14. Dec. 2023].

[4] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Onward! 2019: Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Association for Computing Machinery, New York, NY, USA, 143–153. https://doi.org/10.1145/3359591.3359735

[5] Benjamin Benni, Sébastien Mosser, Naouel Moha, and Michel Riveill. 2019. A delta-oriented approach to support the safe reuse of black-box code rewriters. *J. Softw. Evol. Process.* 31, 8 (2019). https://doi.org/10.1002/smr.2208

[6] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd.

[7] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. 2017. An Empirical Analysis of the Docker Container Ecosystem on GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 323–333. https://doi.org/10.1109/MSR.2017.67

[8] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE ’14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[9] Bikash Gyawali, Lucas Anastasiou, and Petr Knoth. 2020. Deduplication of Scholarly Documents using Locality Sensitive Hashing and Word Embeddings. *European Language Resources Association* (May 2020). https://oro.open.ac.uk/70519

[10] Michael Townsen Hicks, James Humphries, and Joe Slater. 2024. ChatGPT is bullshit. *Ethics and Information Technology* 26, 2 (08 June 2024), 38. https://doi.org/10.1007/s10676-024-09775-5

[11] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017), 1–28. https://doi.org/10.1145/3133908

[12] Philip Mayer and Alexander Bauer. 2015. An empirical analysis of the utilization of multiple programming languages in open source projects. In *EASE ’15: Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2745802.2745805

[13] Muhammad Shumail Naveed. 2022. Correlation Between GitHub Stars and Code Vulnerabilities. *JCBI* 4, 01 (Dec. 2022), 141–151. https://doi.org/10.56979/401/2022/111

[14] Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, and Davide Di Ruscio. 2020. An automated approach to assess the similarity of GitHub repositories. *Software Qual. J.* 28, 2 (June 2020), 595–631. https://doi.org/10.1007/s11219-019-09483-0

[15] Mateusz Pawlik and Nikolaus Augsten. 2015. Efficient Computation of the Tree Edit Distance. *ACM Transactions on Database Systems (TODS)* (2015), 3:1–3:40. https://doi.org/10.1145/2699485

[16] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Information Systems* 56 (2016), 157–173. https://doi.org/10.1016/j.is.2015.08.004

[17] Md Omar Faruk Rokon, Pei Yan, Risul Islam, and Michalis Faloutsos. 2021. Repo2Vec: A Comprehensive Embedding Approach for Determining Repository Similarity. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 355–365. https://doi.org/10.1109/ICSME52107.2021.00038

[18] Sébastien Mosser Sathurshan Arulmohan, Marie-Jean Meurs. 2023. Extracting Domain Models from Textual Requirements in the Era of Large Language Models.. In *5th Workshop on Artificial Intelligence and Model-driven Engineering (co-located with 26th International Conference on Model-Driven Engineering, Languages and Systems (MODELS))*.

[19] Diomidis Spinellis, Zoe Kotti, and Audris Mockus. 2020. A Dataset for GitHub Repository Deduplication. In *MSR ’20: Proceedings of the 17th International Conference on Mining Software Repositories*. Association for Computing Machinery, New York, NY, USA, 523–527. https://doi.org/10.1145/3379597.3387496

[20] Martin Woodward. 2022. Octoverse 2022: 10 years of tracking open source. *GitHub Blog* (Nov. 2022). https://github.blog/2022-11-17-octoverse-2022-10-years-of-tracking-open-source

[21] Yanjie Zhao, Li Li, Haoyu Wang, Haipeng Cai, Tegawendé F. Bissyandé, Jacques Klein, and John Grundy. 2021. On the Impact of Sample Duplication in Machine-Learning-Based Android Malware Detection. *ACM Trans. Software Eng. Method.* 30, 3 (May 2021), 1–38. https://doi.org/10.1145/3446905