Noname manuscript No.
(will be inserted by the editor)

Java Extensible dataset for Many ML4Code Applications

Anjan Karmakar · Miltiadis Allamanis · Romain Robbes ·

Received: date / Accepted: date

Abstract Machine Learning for Source Code (ML4Code) is an active research field in which extensive experimentation is needed to discover how to best use source code's richly structured information. With this in mind, we introduce JEMMA—Java Extensible dataset for Many ML4Code Applications, which is a large scale, diverse, and high-quality dataset targeted at ML4Code.

Our goal with JEMMA is to lower the barrier to entry in ML4Code by providing the building blocks to experiment with source code models and tasks. JEMMA comes with a considerable amount of pre-processed information such as metadata, representations (e.g., code tokens, ASTs), and several properties (e.g., metrics, static analysis results) for 50,000 Java projects of the 50K-C dataset, with over 1.2 million classes and over 8 million methods.

JEMMA is also extensible through an API that allows users to add new properties and representations to the dataset, and evaluate tasks on them. Thus, JEMMA becomes a workbench that researchers can use to experiment with novel representations and tasks operating on source code.

To demonstrate the utility of the dataset, we also report results from two empirical studies of our data, showing that significant work lies ahead in the design of context-aware source code models that can reason over a broader network of source code entities in a software project—the very task that JEMMA is designed to help with.

Keywords Software Engineering · Machine Learning · Empirical Datasets

A. Karmakar
Free University of Bozen-Bolzano, Italy
E-mail: akarmakar@unibz.it

M. Allamanis
Microsoft Research, UK
E-mail: miltiadis.allamanis@microsoft.com

R. Robbes
Free University of Bozen-Bolzano, Italy
E-mail: rrobbes@unibz.it

1 Introduction

Software systems are complex networks of interacting entities. This makes them extremely challenging to develop, understand, and modify—despite the constant need to do so. In this context, appropriate tool support for source code can make developers faster and more productive at doing their job. A variety of such tools have been proposed over the years, ranging from Integrated Development Environments (IDEs), testing tools, static analyzers, version control systems and issue tracking systems, to name a few.

Machine learning for source code. In recent years, a significant research effort has also been undertaken towards developing effective machine learning models of source code (Allamanis et al 2018). This work started from the observation that simple statistical models of source code, such as n-gram models, were surprisingly effective at source code prediction tasks such as code completion (Hindle et al 2016).

Since then such probabilistic models of source code have come a long way. Large-scale machine learning models of source code, based on the Transformer architecture, e.g. `CuBERT` (Kanade et al 2020), `PLBART` (Ahmad et al 2021), `CodeBERT` (Feng et al 2020), and `GraphCodeBERT` (Guo et al 2020) have achieved state-of-the-art performance on a number of software engineering (SE) tasks such as code generation, code search, code summarization, clone detection, code translation, and code refinement. Largely by increasing the capacity of models and training datasets, deep learning based code completion has transitioned from the token level (Karampatsis et al 2020) to completing entire snippets of code (Chen et al 2021), the latter being now available on IDEs in the form of an extension called *GitHub Copilot*[1].

In parallel, other works on modelling of source code have observed that source code has a well-known structure compared to natural language. Source code can be unambiguously parsed into structured representations, such as Abstract Syntax Trees (ASTs); functions and methods can have control flows and data flows; functions and methods can interact with each other via calls, parameters and return values. Therefore, even though modelling source code as a series of tokens—analogous to words in a sentence or a paragraph—has proven to be effective, another view shows that accounting for the structure of source code to be more effective.

A fair amount of research has addressed this issue in source code modelling, by proposing the incorporation of the inherent structural information of source code. Several works model source code as Abstract Syntax Trees (Mou et al 2016; Alon et al 2018; LeClair et al 2019). Allamanis et al. were among the first to model source code snippets as graphs, including a wide variety of structural information, from data flow information, control flow information, lexical usage information, to call information (Allamanis et al 2017).

[1] `https://copilot.github.com`

The space of possibilities to model source code is vast, from text to tokens to advanced graphs—although each comes with its own issues and challenges. Thus, while being mindful of how we represent source code with as much information as possible, we also need to make sure that the models trained on such representations are scalable and reliable for a number of source code tasks and corresponding applications.

From snippets to projects. An important limitation of the current breed of deep learning models for source code is that the vast majority of the work has so far focused much more on single code snippets, methods, or functions, rather than the complex relationships between source code elements, particularly when these relationships cross file boundaries. From this current stage, we now need to gradually move towards building context-aware models that can reason over larger neighborhoods of interacting entities.

A major reason for the lack of such work is that the necessary data is not yet collected or organized, and is missing. The following section highlights the absence of such datasets for code that have the right mix of source code granularity, size, scale, and detail of information to allow researchers to research on models that go beyond single code snippets.

Datasets that are large focus either on individual code snippets at the method-level, or at best, source files; while other datasets are either too small, or lack significant preprocessing. Choosing good quality data in sufficient quantity, downloading and storing the data, extracting valuable information from the data or simply running tools to preprocess the data and gather additional information, and then building an experimental infrastructure in place, requires a large amount of time and effort—even before a single experiment is run. This is all the more true when this has to be done for source code models, where some of the pre-processing and analysis tools can be extremely time-consuming and resource-intensive at scale. Therefore, in this paper, we contribute such a dataset: `JEMMA`, which stands for Java Extensible dataset for Many ML4Code Applications.

`JEMMA` as a dataset. `JEMMA` has multiple levels of granularity: from methods, to classes, to packages, and entire projects. It consists of over 8 million Java method snippets along with substantial metadata; pre-processed source code representations—including graph representations that comes with control- and data-flow information; call-graph information for all methods at the project-level; and a variety of additional properties and metrics. `JEMMA` *stands on the shoulders of giants*: it is built upon the `50K-C` dataset of compilable projects[2] (Martins et al 2018), but complements it with very significant pre-processing, measured in years of compute time. Section 3 presents all the components of the `JEMMA` datasets, and shows how it differs from the `50K-C` dataset.

[2] Researchers citing our work should also cite Martins et al (2018).

`JEMMA` as a workbench. A discernible hurdle in building comprehensive large-scale models for source code is that it is far from obvious how to scale models to handle larger entities of input source code. For instance, in recent work, we have observed that for datasets at the class level, significant truncation occurred in Transformer models (Prenner and Robbes 2022). The models are simply unable to process the entire input. We anticipate that thorough experimentation will be needed to find how to make models able to process these larger inputs. To this end, `JEMMA` is not a static dataset: we purposefully designed it to be extensible in a variety of ways. Concretely, `JEMMA` comes with a toolchain and has a set of APIs to: add metrics or labels to source code snippets (e.g., by utilizing a source code analysis tool); define predictive tasks based on metrics, labels, or the representation themselves; process the code snippets and existing representations to generate new representations of source code; and run supported models on a task. We describe how to extend the dataset, along with several examples in Section 4. This extensibility is critical, because it transforms `JEMMA` to a workbench with which users can experiment with the design of ML models of code and tasks, while saving a lot of time in pre-processing the data.

Empirical studies with `JEMMA`. In Sections 5 and 6, we show how `JEMMA` can be used to gain insights via empirical studies. The first is a study on the *non-localness* of software, and how it impacts the performance of models on a variant of the code completion task. This study shows how `JEMMA` can be used to gain insights on how the models perform on code samples, highlighting what performance issues exist and what we can do to address such issues (Section 5).

The second is the study of the size of entities that constitute software projects, and how it relates to the context size of popular machine-learning (ML) models. The second study confirms that significant work lies ahead in designing models that efficiently encode large contexts (Section 6).

While these examples are related to empirical analyses in the field of Machine Learning for Software Engineering, we can envision other uses for `JEMMA` in empirical studies. Finally, we document the limitations of `JEMMA` in Section 7, and then conclude with a summary of our work in Section 8.

2 Related Work

Traditional source code datasets have contributed immensely to the progress made in the field of software engineering. Studies based on such datasets have helped to uncover the truth behind myriad empirical hypotheses. Subsequently, with the gradual evolution of machine learning techniques suitable for processing code—where data plays a central role—a multitude of efforts have been made for collecting and organizing quality data. Such datasets have not only contributed to the development of competent models of source code, but also opened the avenues for empirical analysis of these models. In this section, we outline some of the datasets from both genres.

2.1 Datasets for machine learning on code

Since machine learning requires considerable amounts of data, multiple datasets have been produced, usually as a means towards validating a specific machine learning method, rather than as a principled standalone effort. This has resulted in datasets that contain input data either not far from raw text, or that contain a lossy view of the underlying analyzed software systems.

Code Datasets. Allamanis and Sutton (2013) collected a set of about 1 billion Java code tokens and provide the code text per file. Later, Karampatsis et al (2020) extended this with additional datasets for C and Python; and a different extension of the dataset was provided by Alon et al (2018). Raychev et al (2016) released Py150 and JS150, two datasets of 150,000 Python and Javascript functions parsed into ASTs.

Several datasets focus on specific tasks, such as the BigCloneBench (Svajlenko and Roy 2015) dataset for large-scale clone detection in Java. ManyTypes4Py (Mir et al 2021) is a Python dataset aimed for evaluating type inference in Python, and Devign (Zhou et al 2019), provide labeled code with coarse-grained source code vulnerability detection in mind.

Datasets with specific representations of code have been common. CoCoGum (Wang et al 2020) use class context represented as abstracted UML diagrams, for code summarization, at the file-level. Allamanis et al (2017, 2020) extract control, data flow graphs, along with syntax within a single file.

Datasets of code from student assignments, programming competitions, and other smaller programs, have also been created. Among them, Google Code Jam[3] and POJ-104 (Mou et al 2016) are clone detection tasks (clone detection in this case is formulated as a program classification task). COSET (Wang and Christodorescu 2019), and CodeNet (Puri et al 2021) also feature smaller programs, but complement them with additional metrics and labels. Although these datasets have many desirable properties, they do not represent source code used in real-life software systems and thus it is unclear if learning on these datasets can generalize to general-purpose software. Semi-synthetic datasets, such as NAPS (Zavershynskyi et al 2018), also fall into the same category.

Code Datasets with Natural Language. Natural language presents an interesting, yet separate, modality from source code and is central to the NLP task of semantic parsing (i.e., text-to-code). A few datasets have focused on this: CodeNN (Iyer et al 2016), CoNaLa (Yin et al 2018), and StaQC (Yao et al 2018). Datasets such as NL2Bash (Lin et al 2018) provide data for semantic parsing from natural language to Bash commands, while Spider (Yu et al 2018) is a dataset for the text-to-SQL semantic parsing task. Finally, Barone and Sennrich (2017), CodeSearchNet (Husain et al 2019), and LeClair and McMillan (2019) pair natural language documentation comments with code, targeting code search and code summarization applications.

[3] `https://code.google.com/codejam/contests.html`

All these datasets provide dumps of source code text per-file, and while it is possible to parse the code text and perform some intra-procedural analyses for the few file-level datasets, information about external dependencies is commonly lost rendering it impossible to extract relatively accurate semantic data.

Code Datasets with Higher-level Representations While the above datasets focus on code snippets or files, some work has extracted datasets aiming for representations that capture information beyond a single file. However, commonly these datasets opt for an application-specific representation that loses information that could be useful for other research. For example, DeFreez et al (2018) extract path embeddings over functions in Linux. LambdaNet (Wei et al 2020) extracts type dependency graphs in TypeScript code but removes most code text information; their dataset is also limited to 300 projects, which range from 500 to 10,000 lines of code. The dataset by LeClair et al. was also refined and used in a source code summarization approach that defined a *project-level* encoder, that considers functions in up to 10 source code files in a project (Bansal et al 2021).

Code Datasets with Changes Given the importance of software maintenance in the development lifecycle, a few datasets have focused on edit operations in code. The goal of these datasets is to foster research in Neural Program Repair. ManySStubs4J (Karampatsis and Sutton 2020) and Bugs2Fix (Tufano et al 2019) both fall in this category: they are corpora of *small* bug fixes extracted from GitHub commits. These datasets often focus on local changes (e.g., diffs) and ignore the broader context.

2.2 Datasets for empirical studies

Several corpora of complete software systems have been built with the primary goal to conduct traditional empirical studies, without direct considerations necessary for machine learning research.

The Qualitas Corpus and its descendants. The Qualitas corpus (Tempero et al 2010), is an influential corpus of 111 large scale Java systems that was used for a large number of empirical studies of the Java and the characteristics of the systems implemented in it. While this dataset was source code only, it was post-processed in various ways, producing several derived datasets. The Qualitas.class corpus, (Terra et al 2013) is a version of the Qualitas corpus that is also compiled in `.class` files. The QUAATLAS corpus (De Roover et al 2013), is a post-processed version of the Qualitas corpus that allows better support for API usage analysis. XCorpus (Dietrich et al 2017), is a subset of the Qualitas corpus (70 programs) complemented by 6 additional programs, that can all be automatically executed via test cases (natural, or generated).

Java Datasets. Lämmel et al (2011) gathered a dataset of Java software from Sourceforge, that had 1,000 projects that were parsed into ASTs. The BOA dataset and infrastructure, by Dyer et al (2013), provides an API for pre-processing software repositories, such as providing and analyzing ASTs, for 32,000 Java projects. The 50K-C dataset of Martins et al (2018) contains 50,000 Java projects that were selected because they could be automatically compiled. A follow-up effort is the Normalized Java Resource (Palsberg and Lopes 2018) (NJR). A first release, NJR-1, provides 293 programs on which 12 different static analyzers can run (Utture et al 2020), but has a stated goal of gathering 100,000 *runnable* Java projects, but is still a work in progress.

Other datasets. Spinellis (2017) released a dataset that contains the entire history of Unix as a single Git repository. The entire Maven software ecosystem was released as a dataset with higher-level metrics, such as changes and dependencies (Raemaekers et al 2013). Fine-GRAPE is a dataset of fine-grained API usage across the Maven software ecosystem (Sawant and Bacchelli 2017). Finally, both Software Heritage (Pietri et al 2019) and World of Code (Ma et al 2021) are very large-scale efforts that aim to gather the entirety of open-source software as complete and up-to-date datasets. The main goal of World of Code is to enable analytics, while the main goal of Software Heritage is preservation (although it also supports analytics).

2.3 The 50K-C Dataset

Having surveyed the landscape of existing datasets, we conclude that most machine learning datasets focus on small-scale entities such as functions, methods, or sing;e classes. The ones that offer higher-level representations are specific and too small in scale. The corpora of systems used for empirical studies provide a better starting point, as they can be pre-processed to extract additional information. Of the existing datasets, the most suitable option that is large enough and that allows the most pre-processing is the `50K-C` dataset of 50,000 compilable projects.

Since `JEMMA` builds upon `50K-C`, we provide detailed background information on it in this section. The `50K-C` dataset is a collection of 50,000 compilable Java projects, with a total of almost 1.2m Java class files, its compiled bytecode, dependent jar files, and build scripts. It is divided into three subsets:

- `projects`: It contains the 50,000 java projects, as zipped files. The projects are organized into 115 subfolders each with about 435 projects.
- `jars`: It contains the 5,362 external jar dependencies, which are required for successful project builds. This is important as missing dependencies is the common cause for failing to compile code at scale.
- `build_results`: It contains the build outputs for the 50,000 projects, including compiled bytecode, build metadata, and original build scripts. In addition to the above data, a mapping between each project and its GitHub `URL` is also provided. The bytecode is readily available for a variety of tasks,

such as running static analysis tools, or, if the projects can also be executed, as input for testing, and dynamic analysis tools.

Beyond the size of the dataset, the fact that the projects are compilable is the main reason we chose to build upon `50K-C`. The extensive pre-processing that we perform on top of `50K-C` requires the use of static analysis tools, to do things such as call graph extraction, and to extract valuable metrics about the systems. Since the vast majority of static analysis tools operate on bytecode, `50K-C` was the most suitable option that combines both scale and the ability to automate the analysis at such scale.

Selection Criteria. The dataset authors downloaded close to 500,000 Java projects, attempted to compile all of them, and selected 50,000 projects among the ones that could be compiled. Two filters were applied: projects that were Android application were excluded, and projects that were clones were also excluded—using the DéjàVu clone repository (Lopes et al 2017), and the Sourcerer CC tool (Sajnani et al 2016). Based on our own assessment, we find that the projects have a diverse set of domains (e.g., games, websites, standalone applications, etc), and of development settings (ranging from student projects to industry-grade open-source projects).

The 50K-C dataset consists of both large-scale projects with as many as 5000 classes, and smaller projects with as low as 5 classes. While the larger projects are good representatives of real-world projects, the smaller projects are valuable too. Machine learning models of code still need to make significant headway before scaling up interprocedural machine learning models (see Section 6): these smaller projects are good intermediate targets towards that goal, allowing the field to make progress.

3 The `JEMMA` Datasets

Our goal with the `JEMMA` project is to provide the research community with a large, comprehensive, and extensible dataset for Java that can be used to advance the field of source code modelling. The `JEMMA` datasets consists of a large collection of code samples in varying granularities, with wide-ranging and diverse metadata, a range of supported source code representations, and several properties. In addition, it also includes source code information related to code structure, data-flow, control-flow, and caller-callee relationships for code entities.

For every project in the `JEMMA` dataset, we gather data at the project-level, and provide information on all the packages and classes of the project along with its corresponding metadata, representations, and selected properties. Furthermore, for every class, we provide data on all the methods - including respective metadata, several representations and properties. The detail of data provided for every method entity is comprehensive, with data at the level of AST with data-flow, control-flow, lexical-usage, and call-graph edges among others. In addition to other necessary information such as line numbers and position numbers of every source code token, supplementary information such as token types, node types, etc, are also provided. More details are presented in the following sections.

JEMMA also comes equipped with a toolchain and a corresponding API that allows a variety of tasks, such as: transforming code samples into intermediate source code representations, making tailored selections of entities to define tasks and forming custom datasets, or to run supported models (Section 4).

Statistics. The original 50K-C dataset contains a total of 50,000 projects. It has 85 projects with over 500 classes (with a maximum of 5549 classes in a project), 1264 projects with 101–500 classes, 2751 projects with over 51–100 classes, 10693 projects with over 21–50 classes, 14322 projects with more than 11–20 classes, and 20885 projects with 10 or fewer classes (with a minimum of 5 classes in a project). We have collected metadata for all of these projects. Overall, the data consists of 1.2 million Java classes, which together define over 8 million unique methods.

Granularity. The primary entity-of-interest in our datasets is a method, as it is the most basic unit of behaviour in an object-oriented program. Coarser granularity entities are available, since the data has been collected at the level of projects. For instance, method snippets can be extracted from the datasets and used independently to run code-analysis tools, or along with their parent Java class file, depending on the tool used. In addition, information at finer-grained levels (e.g., AST nodes) is also available since the representations that we post-processed contain this level of detail.

Compilability. Successful compilation ensures that the source code snippets from the projects have been type-checked and parsed successfully, and are valid Java code. Having full-scale compilable projects gives us the assurance that the source code is complete and self-contained; and thus, all the inter-relationships among code entities can be captured and studied. Additionally, static and dynamic analysis tools can be run to generate information for new code tasks.

Some of the tools that we use to post-process the data require the ability to compile the code, rather than just analyzing compiled code. These tools insert themselves in the compilation process (for instance, Infer (Calcagno et al 2015)). Therefore, we also have to be able to compile the code on demand. Practically, we found that recompilation was not 100% replicable. Of the 50,000 projects, we were able to compile about 90% of the projects; a failed compilation is usually linked to a missing or inaccessible dependency.

Runtime considerations. The post-processing that we apply to the projects is very computationally intensive for some of the tools. For instance, the analyses run just by a single tool—the Infer static analyzer (Calcagno et al 2015)—can take on the order of half an hour for a single medium-sized project. Analyzing 50,000 projects with a number of tools and then post-processing the outputs is thus both time-consuming and resource-intensive.

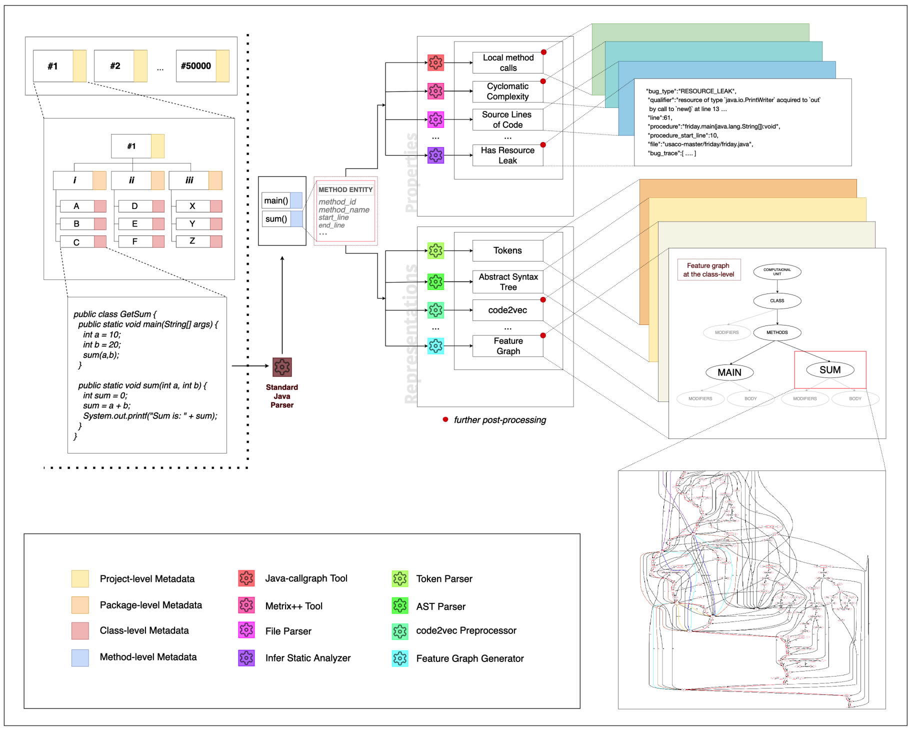
Table 1 JEMMA dataset artifacts, locations, and sizes

Artifact	DOI	Size
Metadata: Projects	https://doi.org/10.5281/zenodo.5807578	4.7 MB
Metadata: Packages	https://doi.org/10.5281/zenodo.5807586	42.2 MB
Metadata: Classes	https://doi.org/10.5281/zenodo.5808902	269.7 MB
Metadata: Methods	https://doi.org/10.5281/zenodo.5813089	2.8 GB
Properties: [TLOC]	https://doi.org/10.5281/zenodo.5813102	335.5 MB
Properties: [SLOC]	https://doi.org/10.5281/zenodo.5813094	335.0 MB
Properties: [NUID]	https://doi.org/10.5281/zenodo.5813028	335.6 MB
Properties: [NTID]	https://doi.org/10.5281/zenodo.5813029	336.7 MB
Properties: [NMTK]	https://doi.org/10.5281/zenodo.5813032	342.5 MB
Properties: [NMRT]	https://doi.org/10.5281/zenodo.5813034	333.3 MB
Properties: [NMPR]	https://doi.org/10.5281/zenodo.5813053	333.3 MB
Properties: [NMOP]	https://doi.org/10.5281/zenodo.5813055	334.5 MB
Properties: [NMLT]	https://doi.org/10.5281/zenodo.5813054	333.4 MB
Properties: [NAME]	https://doi.org/10.5281/zenodo.5813308	432.0 MB
Properties: [MXIN]	https://doi.org/10.5281/zenodo.5813081	267.0 MB
Properties: [CMPX]	https://doi.org/10.5281/zenodo.5813084	267.1 MB
Represent.: (TEXT)	https://doi.org/10.5281/zenodo.5813705	3.8 GB
Represent.: (TKNA)	https://doi.org/10.5281/zenodo.5813717	3.3 GB
Represent.: (TKNB)	https://doi.org/10.5281/zenodo.5813730	4.6 GB
Represent.: (ASTS)*	https://doi.org/10.5281/zenodo.5813880	4.1 GB
Represent.: (FTGR)*	https://doi.org/10.5281/zenodo.5813933	5.2 GB
Represent.: (C2VC)*	https://doi.org/10.5281/zenodo.5813993	6.1 GB
Represent.: (C2SQ)*	https://doi.org/10.5281/zenodo.5814059	10.9 GB

Storage and sharing considerations. The amount of data produced is considerable. To maximize accessibility, we provide it as a set of Comma Separated Values (CSV) files, so that users can choose and download the data that they need. Some of the representations that are highly structured are stored in JSON files. Note that only the metadata and the original source data are absolutely necessary; other data can be downloaded on a per-need basis. The JEMMA toolchain and API allows the recomputation of the other properties, if, for some properties, it is more efficient to recompute them than to download them. The data is uploaded on Zenodo; due to its size, it is provided as multiple artifacts. Table 1 presents the components of the dataset, along with their DOIs (links to the download page), and sizes.

Interacting with the data. Most of the data from the JEMMA are organized in Comma-Separated-Values (CSV) files, consequently basic analyses can be run with tools such as *csvstat*. Furthermore, our API can be used to gather extensive statistics of the projects, classes, methods, bytecode, and data and control-flow information.

The JEMMA datasets are grouped into three major parts: data at the metadata level (Section 3.1), data at the property level (Section 3.2), and at the representation level (Section 3.3). In addition, we also provide project-wide callgraph information for the 50,000 projects, uniquely identifying and associating source



```
public class GetSum {
  public static void main(String[] args) {
    int a = 10;
    int b = 20;
    sum(a,b);
  }

  public static void sum(int a, int b) {
    int sum = 0;
    sum = a + b;
    System.out.printf("Sum is: " + sum);
  }
}
```

```
"bug_type":"RESOURCE_LEAK",
 "qualifier":"resource of type `java.io.PrintWriter` acquired to `out`
  by call to `new()` at line 13 ...
 "line":61,
 "procedure":"friday.main(java.lang.String[]):void",
 "procedure_start_line":10,
 "file":"usaco-master/friday/friday.java",
 "bug_trace":[ .... ]
```

Fig. 1 Overview of data-level contributions

and destination nodes in the callgraph with the help of the metadata defined by JEMMA (Section 3.4). This allows for accessing project-wide data on the whole, for different granularities of code entities.

Fig. 1 gives a glimpse of the extent and detail of data contribution made by JEMMA. The top-left corner represents the raw data from 50K-C, which we extend by adding UUIDs (symbolized by colored squares). The rest of the figures depicts the additional pre-processing we performed: the gears on colored background represent external tools that we run to collect additional data (properties and representations), while the red dots represent further post-processing that we perform on the tool outputs to integrate it in our dataset.

3.1 JEMMA: Metadata

In this section we present the metadata for the JEMMA datasets. The metadata is made available in comma-separated value (CSV) files. This allows for easy processing, even with simple command-line tools. The metadata are organized in four parts, following the largest units of interest: projects, packages, classes, and methods. The units of interest can then be inter-related systematically. The metadata serves two major purposes:

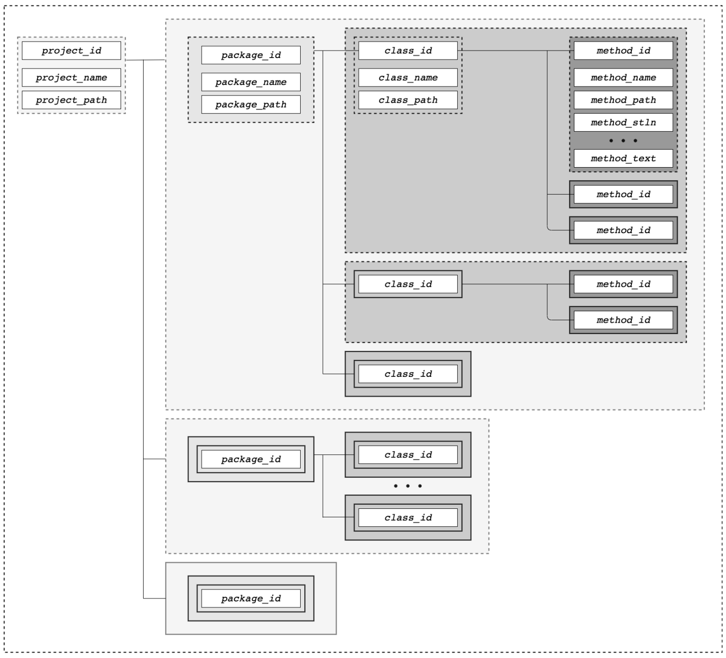


Fig. 2 Meta-structure of project entities and their inter-relationships

1. Uniquely identify a source code entity with an `UUID`.
2. Gather basic and often-used information on each source code entity.

Taken together, these two allow users to extend the dataset with additional properties. The `UUID` allows to uniquely identify an entity in the dataset, and the supplementary metadata helps disambiguate entities (file paths, parent relationships, location information in the file, etc). In Section 4 we show how this metadata can be used to add an additional property to source code entities.

Since the metadata formalizes the organization of the data, and establishes the relationship between projects, packages, classes, and methods, `JEMMA` users can leverage it to construct custom data queries and make selections from the large collection of data at different granularities. The overall organization of the metadata and their inter-relationships is shown in Figure 2.

3.1.1 Projects

For the project-level metadata, we provide a single CSV file that lists all the projects in the `50K-C` dataset along with their corresponding metadata—*project_id*, *project_path*, *project_name*. The `UUID` is referenced by the entities contained in the project. The project path is relative to the root directory of the `50K-C` dataset[4], and can be used to access the raw source code of the project. Table 2 details the metadata for projects.

[4] `http://mondego.ics.uci.edu/projects/jbf/downloads/50K-C_projects.tgz`

Table 2 JEMMA projects metadata csv

column_name	data_type	description
project_id	str	UUID assigned to project
project_path	str	Relative path of the project
project_name	str	Name of the project

Table 3 JEMMA packages metadata csv

column_name	data_type	description
project_id	str	*project_id* of parent project
package_id	str	UUID assigned to package
package_path	str	Relative path of the package
package_name	str	Name of the package

Table 4 JEMMA classes metadata csv

column_name	data_type	description
project_id	str	*project_id* of parent project
package_id	str	*package_id* of the parent package
class_id	str	UUID assigned to class
class_path	str	Relative path of the class
class_name	str	Name of the class

3.1.2 Packages

For the package-level metadata, a single CSV file lists all the packages present in the projects. The metadata comprises of the UUID of the parent project as *project_id*, the UUID assigned to the package as *package_id*, the relative path of the package as *package_path*, and the name of the package directory as *package_name*. Table 3 details the metadata we provide for packages.

3.1.3 Classes

For the class-level metadata, we provide a single CSV file that lists all the classes in the 50K-C dataset along with their corresponding metadata: *project_id*, *package_id*, *class_id*, *class_path*, *class_name*. Table 4 details the metadata we provide for classes. Similarly to the projects, the class path is a relative path starting from the 50K-C dataset's root directory, that allows to access the raw source code of the class.

3.1.4 Methods

At the method-level, the metadata is more extensive. Just having the name of a method might not be enough to disambiguate methods, due to the fact that several methods in the same class can have the same name but different arguments. Thus, the metadata is a CSV file lists all the methods in

Table 5 JEMMA methods metadata csv

column_name	data_type	description
project_id	str	*project_id* of parent project
class_id	str	*package_id* of parent package
class_id	str	*class_id* of parent class
method_id	str	UUID assigned to the method
method_path	str	Relative path of the parent class
method_name	str	Name of the method
start_line	int	Method start line in the parent class
end_line	int	Method end line in the parent class
method_signature	str	Method signature of the method

the 50K-C dataset along with their corresponding metadata: *project_id*, *package_id*, *class_id*, *method_id*, *method_path*, *method_name*, *start_line*, *end_line*, *method_signature*. Table 5 details the method metadata.

3.2 JEMMA: Properties

JEMMA leverages the UUIDs assigned to projects, classes, and methods as a way to attach additional properties to these entities. Thus, a property can be an arbitrary value that is associated to an entity, such as a metric. Even though we have gathered several properties associated with code entities comprehensively, it should be noted that a particular property may not be available or may not apply for a given code entity. Users can add new properties associated with code entities as contributions to the dataset, where the property should be given a unique name and be stored in the correct location for it to be visible to JEMMA API (Section 4 provides more details).

We have run several static analysis tools to define properties associated with source code entities:

- The *Infer* static analyser (Calcagno et al 2015) is a tool that provides advanced abstract interpretation-based analyses for several languages, including Java. Examples of the analyses that Infer can run include an interprocedural analysis to detect possible null pointer dereferences. Infer can also perform additional analyses such as taint analysis, resource leak detection, and estimate the run-time cost of methods.
- *Metrix++* is a tool that can compute a variety of basic metrics on source code entities, such as lines of code, code complexity, and several others[5].
- *PMD* is a static code analysis tool[6] that can compute a variety of static analysis warnings and metrics, such as the *npath complexity* metric, among many, many others.

[5] https://metrixplusplus.github.io/metrixplusplus/
[6] https://pmd.github.io

The following is the list of properties that are currently defined at the method level in `JEMMA`, and made available as CSV datasets. The properties marked with * are not yet available for all the samples, but computations are in progress for the remaining, since some of the analyses are compute-heavy. For instance, to determine that a null dereference is possible at a given point of the program, Infer performs an analysis based on abstract interpretation of the source code. Infer's output includes an execution path that the program can take that could lead to a null dereference. Such analyses can take significant amounts of time (on the order of tens of minutes even for small projects).

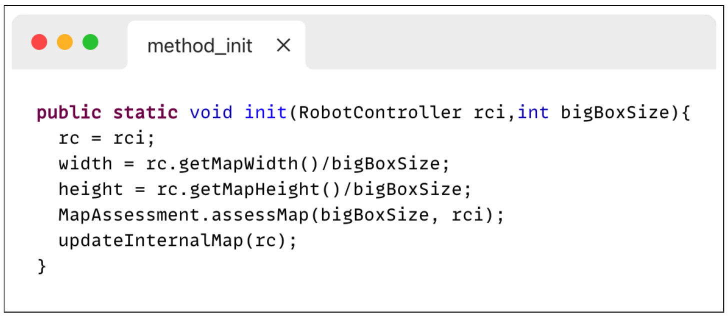
1. `[TLOC]` Total Lines of Code
2. `[SLOC]` Source Lines of Code
3. `[NMTK]` Number of Code Tokens
4. `[CMPX]` McCabe or Cyclomatic Complexity
5. `[NPTH]` Npath Complexity
6. `[MXIN]` Maximum indent depth of nesting[7]
7. `[NMPR]` Number of parameters
8. `[NUID]` Number of unique identifiers
9. `[NMOP]` Number of operators
10. `[NMLT]` Number of literals
11. `[NMRT]` Number of return statements
12. `[NAME]` Name of source code entity
13. `[NMOC]` Number of calls outside class*
14. `[NUPC]` Number of unique parent callers*
15. `[NUCC]` Number of unique child callees*
16. `[NLDF]` Presence of Null Dereference*
17. `[RSLK]` Presence of Resource Leak*
18. `[MCST]` Method Runtime Cost* [runtime complexity estimation]

Other tools that could be run to extend the dataset include static analysis tools, such as FindBugs (Hovemeyer and Pugh 2004), PMD, or similar tools such as Error Prone and NullAway. The warnings and outputs from these tools can serve as metrics for code entities. Bespoke static analyses from Soot or another static analysis research frameworks, or clone detection (Cordy and Roy 2011) tools could be run as well. These properties could be useful to conduct studies similar to the ones from Habib and Pradel (2018).

3.3 `JEMMA`: `Representations`

Machine learning models are trained on a collection of feature vectors derived from the input data. For source code machine learning models the input data can be the raw text of a source code entity. For example, for the Java method shown in Figure 3a, a corresponding source code representation could be its raw tokens as shown in Figure 3b.

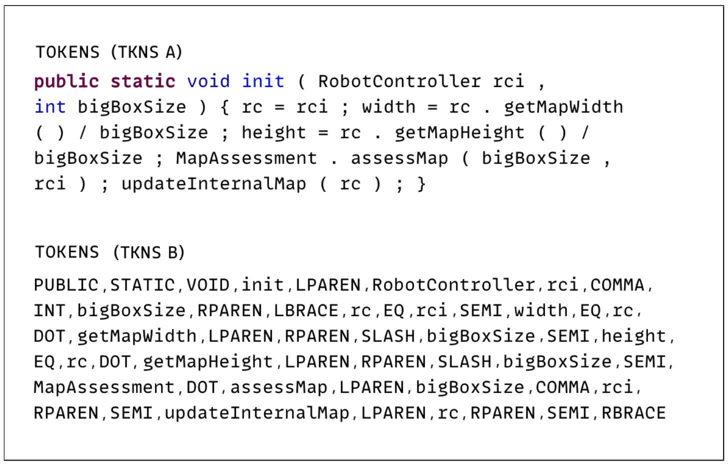
[7] A measure of indentation of the source code, a proxy for complexity (Hindle et al 2008)



method_init

```java
public static void init(RobotController rci,int bigBoxSize){
  rc = rci;
  width = rc.getMapWidth()/bigBoxSize;
  height = rc.getMapHeight()/bigBoxSize;
  MapAssessment.assessMap(bigBoxSize, rci);
  updateInternalMap(rc);
}
```

(a) Java method as original text representation



```
TOKENS (TKNS A)
public static void init ( RobotController rci ,
int bigBoxSize ) { rc = rci ; width = rc . getMapWidth
( ) / bigBoxSize ; height = rc . getMapHeight ( ) /
bigBoxSize ; MapAssessment . assessMap ( bigBoxSize ,
rci ) ; updateInternalMap ( rc ) ; }


TOKENS (TKNS B)
PUBLIC,STATIC,VOID,init,LPAREN,RobotController,rci,COMMA,
INT,bigBoxSize,RPAREN,LBRACE,rc,EQ,rci,SEMI,width,EQ,rc,
DOT,getMapWidth,LPAREN,RPAREN,SLASH,bigBoxSize,SEMI,height,
EQ,rc,DOT,getMapHeight,LPAREN,RPAREN,SLASH,bigBoxSize,SEMI,
MapAssessment,DOT,assessMap,LPAREN,bigBoxSize,COMMA,rci,
RPAREN,SEMI,updateInternalMap,LPAREN,rc,RPAREN,SEMI,RBRACE
```

(b) Java method represented as tokens

Fig. 3 A Java method and its possible token representation

Since source code is highly structured, the design space for representations is vast and diverse. This has been been explored to some extent, with approaches that model source code as sequences of tokens or subtokens via RNNs (Pradel and Sen 2018), LSTMs (Karampatsis et al 2020), or Transformers (Feng et al 2020). Other approaches have leveraged the structure of code either via ASTs (Mou et al 2016) or linearized ASTs (Alon et al 2019). Yet other approaches use more expressive structures incorporating, for instance, data flow, and use Graph Neural Networks (GNNs) to represent code (Allamanis et al 2017).

Our goal with JEMMA is to provide the building blocks to experiment with the design space of representations. Since extracting the relevant information is costly in terms of computational resources, a significant effort went into adding several basic representations at the method level, ranging from the raw source code to the information behind a very complete graph representation. At the representation level, we provide several ready-to-use source code representations (compatible for different models) for over 8 million method snippets. The method level representations that we provide are:

1. (`TEXT`) Raw source code.
2. (`TKNA`, `TKNB`) Source code tokens parsed with an `ANTLR4` grammar.
3. (`ASTS`) Abstract Syntax Tree representation
4. (`C2VC`, `C2SQ`) Linearized `AST` representations as Bags of `AST` paths
5. (`FTGR`) Feature graph representation, which are basic `ASTs` enriched with a significant number of additional edges, depicting various inter-relationships between the `AST` nodes (e.g., data-flow, lexical-usage etc.).

3.3.1 Raw text (`TEXT`)

First and foremost, the original code for each method is provided by default, with no preprocessing. This allows approaches that need to operate on the raw text to do so directly (e.g., a model that implements its own tokenization). The default method text includes its annotations and also comments within the code snippet, if any. The whitespace for each method text is also preserved intentionally (it can be easily stripped off at any point). The raw text can also be used to re-generate the other representations as needed.

3.3.2 Tokens (`TKNA`, `TKNB`)

Each Java method snippet is tokenized with an `ANTLR4` grammar (Parr 2013) and made available to the user preprocessed. The tokenized code includes method annotations, if any, but does not include natural language comments. However, with the entire raw text of method snippets made available by default, users are free to include comments in their custom tokenized representations.

For every method snippet in our dataset, we provide the corresponding string of tokens. In fact, we provide two types of tokenized strings. First, a simple space-separated string of tokens. This representation is meant to be directly passed to an existing ML model that has its own tokenizer, without any further processing. The downside is that some literals that include spaces may be treated as more than one token, or symbols and special characters may be ignored while using certain tokenizers. Should this be an issue, users may use the second representation.

In the second type of tokenized representation the tokens are made available as a comma-separated string with the symbols and operators replaced with suitable string tokens (commas in literals are replaced suitably with `<LITCOMMA>` tokens). This representation is recommended for users who would tokenize the code themselves, or would want to avoid literals being split into several tokens, or avoid ambiguities with symbols and special characters when using NLP tokenizers.

3.3.3 Abstract Syntax Tree (`ASTS`)

The AST representations are a subset of the information from the feature graphs (detailed below). Specifically, for the AST we keep all the nodes from the parse tree, and all edges that represent parent/child relationships.

Nodes can either be terminal nodes (leaf nodes), which are the tokens of the grammar, or inner non-terminal nodes, representing the higher-level units such as method calls, code blocks, etc. This information is represented for each method as a set of nodes, followed by a set of node pairs representing child edges.

3.3.4 code2vec (`C2VC`) and code2seq (`C2SQ`)

The code2vec (Alon et al 2019) and code2seq (Alon et al 2018) representations are derivatives of the AST representation. The goal of these approaches is to linearize ASTs by sampling a fixed number of AST paths (i.e., selecting 2 AST nodes at random and finding the path between them). The difference between the approaches are that code2vec represents each identifier and each path as unique symbols leading to large vocabularies, and consequently Out-Of-Vocabulary (`OOV`) issues, while code2seq models identifiers and paths as sequences of symbols from smaller vocabularies, which alleviates the issues. The downside is that the code2seq representation is significantly larger. Both kinds of inputs are fed to models that use the attention mechanism to select a set of AST paths that are relevant to the model's training objective (by default, method naming).

We have generated the code2vec and code2seq representations of every method in the dataset, which can serve as a ready-to-use input to the code2vec and code2seq path-attention models. Furthermore, if alterations to the code snippets are needed, our toolchain and `API` enables the users to easily transform a valid code snippet into its corresponding representation(s) using the original code2vec and code2seq preprocessors.

3.3.5 Feature Graph (`FTGR`)

The feature graph representation is obtained from Andrew Rice's feature graph extraction tool[8]. This tool is a plugin for the Java compiler—hence we need to be able to compile the source code data ourselves—which extracts information at par with the one used to build the feature graphs in the work of Allamanis et al (2017) for C# code. The Feature Graph representations are represented as a set of nodes, and then node pairs representing different edge types; the nodes are also presented in a sequence to capture the order of code tokens. Specific edge types can be filtered out as needed (such as to produce the AST representations, or to reduce the size of the graph (Hellendoorn et al 2019b)). The full list of included edges are:

[8] `https://github.com/acr31/features-javac`

- `Child` edges encoding the `AST`.
- `NextToken` edges, encoding the sequential information of code tokens.
- `LastRead`, `LastWrite`, and `ComputedFrom` edges that link variables together, and provide data flow information.
- `LastLexicalUse` edges link lexical usage of variables (independent of data flow).
- `GuardedBy` and `GuardedByNegation` edges connecting a variable used in a block to conditions of a control flow.
- `ReturnTo` edges link return tokens to the method declaration.
- `FormalArgName` edges connect arguments in method calls to the formal parameters.

Note that this representation is computed at the granularity of files (entire classes). We keep this file-level representation for completeness, but also extract the representations of individual methods from it. This representation is extremely complete, as it includes, for instance, all the source code tokens and their precise locations in the original source code, the signatures of all the methods called in the class, the source code comments, if any, including a variety of data-flow, control-flow, lexical-usage, and hierarchical edges. Derivative representations such as AST with dataflow information, a subset of the feature graph representation, which corresponds to the data used by models such as `GraphCodeBERT` can also be produced from the feature graph representations.

3.3.6 Extended utility

While the above representations can be used as-is as input for various machine learning models, they can be further pre-processed to tailor them to specific use cases. For example:

(a) The definition of variants of the existing representations, such as the AST with dataflow representation above, which is a simplification of the feature graph. Experimenting with these variants is important, to obtain a better understanding of the trade-offs between the kinds of information available, what they can bring to a model, and the difficulty of obtaining the information. See Section 4.1.2 for further examples.

(b) The definition of tasks that operate on altered versions of the input, such as tasks where part of the input is masked (e.g., MethodNaming task), or modified (e.g., VarMisuse task). Section 4.2.2 provides an example of this. This could also serve as the basis for data augmentation approaches, where semantically equivalent variations of the source code could be programmatically generated.

(c) Finally, the representations can also be combined to form the basis for representations of higher-level entities, such as classes, packages, or projects. This is key to provide machine learning models of code that can reason at these higher level. This is also a further incentive to explore variations of representations, since, as shown in Section 6, scaling machine learning models to larger code snippets comes with significant challenges.

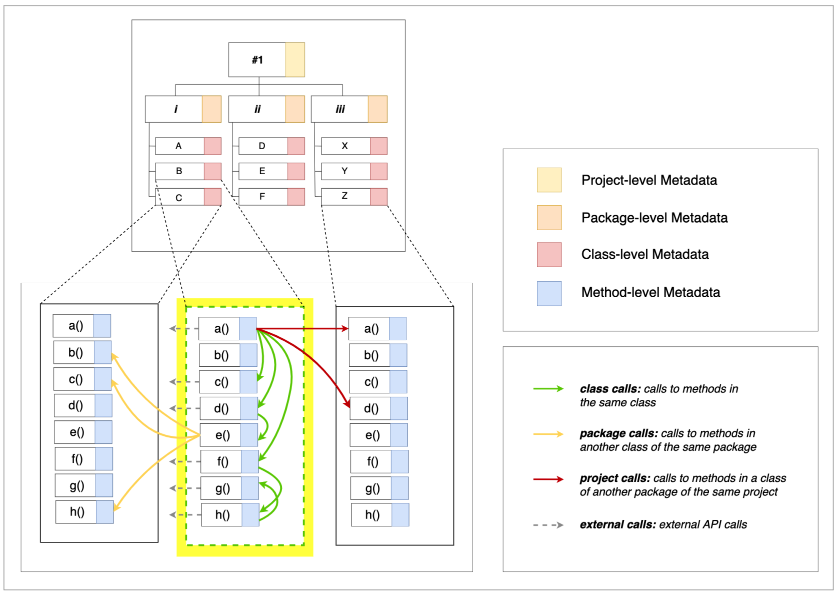

#1
i
ii
iii
A
B
C
D
E
F
X
Y
Z
a()
b()
c()
d()
e()
f()
g()
h()
Project-level Metadata
Package-level Metadata
Class-level Metadata
Method-level Metadata
class calls: calls to methods in the same class
package calls: calls to methods in another class of the same package
project calls: calls to methods in a class of another package of the same project
external calls: external API calls

Fig. 4 Project-level callgraph edges connecting method-entities

3.4 Project-wide call graph (`CG`)

Since many relationships among source code entities are not simply hierarchical containment relationships, we also provide a very useful additional data: the project's call graph (`CG`), in which methods calling each other are explicitly linked. Thanks to our metadata, these method call information (Figure 4) can then be used to combine representations to create interesting global contexts for large-scale source code models.

Previous techniques are useful to design representations at the level of methods. However, designing models that reason about larger entities require more data. Hierarchical relationships can be already inferred from the metadata. In addition, since software systems are composed of modules that interact with each other, caller-callee relationships are crucial to model systems accurately. For this, we use the Java callgraph extractor tool by Georgios Gousios[9], to extract project-wide call graphs, from which callers and callees are identified and linked to their respective `UUIDs` through post-processing (links to external calls are still recorded but we do not assign `UUIDs` to them).

Method signatures are used to disambiguate method with similar names. Note that for polymorphic calls, the call graph provides links to the statically identified type, not to all possible types. Additional post-processing would be possible to add these links to the call graph. In previous work, we have seen that the use of polymorphism in Java is significant (Milojkovic et al 2015), so this would be a useful addition.

[9] `https://github.com/gousiosg/java-callgraph`

4 Extending and Using `JEMMA`

When building `JEMMA`, we intended it to be both extensible and flexible. In this way, researchers could use `JEMMA` as a workbench to experiment with variants of datasets, models, and tasks while minimizing the preprocessing that is involved. In this section, we show how `JEMMA` can be extended and then describe various scenarios in which it can be used.

4.1 Extending `JEMMA`

In Section 4.1.1 we principally describe how `JEMMA` can be extended with a new property (e.g., a metric or a tool warning associated with a code entity), and in Section 4.1.2 we describe how it can be extended with a new representation.

4.1.1 Adding a new property

The simplest way to extend `JEMMA` is to add a new property. This could be any property of interest that can be computed for a source code entity. Examples include defining a new source code metric, such as source code complexity, or the result of a static analysis tool indicating the presence (or absence) of a specific source code characteristic.

To extend `JEMMA` with a new property, the workflow has three main steps: a) accessing a set of source code entities, b) generating associated properties, and c) merging the associated property values to the dataset. `JEMMA` facilitates accessing the correct code input by providing the location metadata for code entities, and several initial representations (raw text, `ASTs`, etc.), and further allowing the source code to be compiled, if necessary. An associated property could then be obtained either directly (e.g. method name) or by means of a code analysis tool (e.g. cyclomatic complexity).

These tools may be run on different granularities of source code as input (e.g. classes, entire projects), and may output results based on the whole projects rather than finer code entities. Therefore, once a property is computed, the user must post-process the output of the tool to find to which entity it should be associated.

Fortunately, `JEMMA` provides the most common ways to reference an entity, and from there, to find the entity's `UUID`. For instance, for method-level code entities, `JEMMA` not only provides the method name but also additional information such as start-end line numbers and positions in the original source file (useful for tools that pinpoint a specific line, or to disambiguate identical method names in the same class).

Snippet A.1 in the appendix gives us an example of how output metrics from the *Metrix++* tool can be associated with the methods defined in `JEMMA`, and how easily it can be added to the `JEMMA` datasets as properties.

4.1.2 Adding a new representation

Different machine learning models of code require different source code representations as input. Some models reason over tokenized source code, while other models reason over more complex structures such as ASTs. Each representation comes with its own set of advantages and drawbacks, while one is extremely feature-rich the other is simple, scalable, and practical. Therefore, the work on representations is still an active area of research—as researchers are continuing to develop new source code representations, or improving the present ones, e.g., by augmenting them with further information.

`JEMMA` makes is quite simple to do both: create new representations, and modify existing ones. There are three main steps to extend `JEMMA` with a representation: a) accessing a set source code entities, b) generating associated representations, and c) merging the representations to `JEMMA`.

The raw source text, or even representations such as the AST, could be accessed directly to produce new representations for associated code entities. And with an array of source code representations readily made available for over 8 million code enities, simplifying or augmenting such representations to create others would save a lot of pre-processing time for the users. In addition, newer representations could also be derived from existing representations based on specific model architectures and needs.

The feature graph representation which we include in our dataset (see Section 3.3.5) is built upon the code `AST`, and is extended with a number of additional edges, depicting various inter-relationships between the `AST` nodes (e.g., data-flow, control-flow, lexical-usage, call-edges among others). In addition to other necessary information such as line numbers and position numbers of every source code token, supplementary information such as token types, node types, are also provided. Thus, with this representation, the detail of data provided for every code entity is comprehensive.

Several derivative representations can be created directly from this one representation by choosing the necessary edge types from the feature graph. For example, for models that require the AST representation as input, choosing just the *Child* edges of the feature graph representation would result in the AST representation. Yet for models that reason over the dataflow information, choosing the *LastRead* and *LastWrite* edges of the feature graph would result in a new dataflow-based representation.

Beyond deriving descendant representations, adding further edge types to the feature graph is always possible making it even more feature-rich, and `JEMMA` facilitates such extensions by providing the base representations for several million code entities. In a similar manner, the other representations included with `JEMMA` could also be simplified, modified, augmented to create new representations.

Once new representations are created, they are associated with corresponding source code entities by means of `UUIDs`. The representations can then be added to `JEMMA` using the underlying toolchain and API—quite similar to that of adding new properties as demonstrated in Snippet A.1.

4.2 Using JEMMA

In this section, we describe various scenarios in which JEMMA can be put to use. In Section 4.2.1 we describe how a property can be used to define a prediction task, while discussing ways in which JEMMA can help avoid common pitfalls and biases. In Section 4.2.2 we explain how source code representations can be used to define a prediction task. Then in Section 4.2.3 we describe how models can be trained and evaluated on prediction tasks using the JEMMA API, and finally, in Section 4.2.4 we describe how new and extended representations can be formulated with a greater context.

4.2.1 Defining tasks based on properties

Once a property is defined, it can be used in a variety of ways. One such way is to define a property prediction task. While this can appear trivial—taking a random sample of entities that have that property defined, and splitting it into training, validation, and test sets—in practice this is often more complex.

The reason why this can be more complex, is that care must be taken that the data does not contain biases that provide an inaccurate estimate of model performance. In this context, there are several groups of issues that JEMMA helps contend with while defining the task datasets.

Rare data. The first is that some properties may be very rare, making them hard to learn at all. Examples of this would be uncommon bugs and errors such as, e.g., resource leaks. Since JEMMA is large to start with (over 8 million method entities), it makes it much more likely that there is enough data to learn in the first place, compared to other alternatives.

Defining task prediction labels. Once a property is defined, JEMMA allows flexibility in the definition of the task prediction labels. For instance, for classification tasks, JEMMA allows for the definition of a balanced set of prediction labels, including, for numerical values, binning close values together to define a single label. A subset of the data can also be selected, if one wants to define a task for which data is more scarce, in order to incentivize sample-efficient models.

Investigating and mitigating biases. When defining a task, care must be taken that the models learn from the right signal, and not from correlated signal that may be easier to learn from, but is not actually predictive.

A well-known example in computer vision is a model that learns to classify cows and camels, but does it by focusing on the background of an image (grass vs sand), instead of the animal's shape; some models have been found to underperform when animals are in unusual settings, such as a cow on the beach (Beery et al 2018). Similar issues have been observed in NLP as well (McCoy et al 2019), (Gururangan et al 2018).

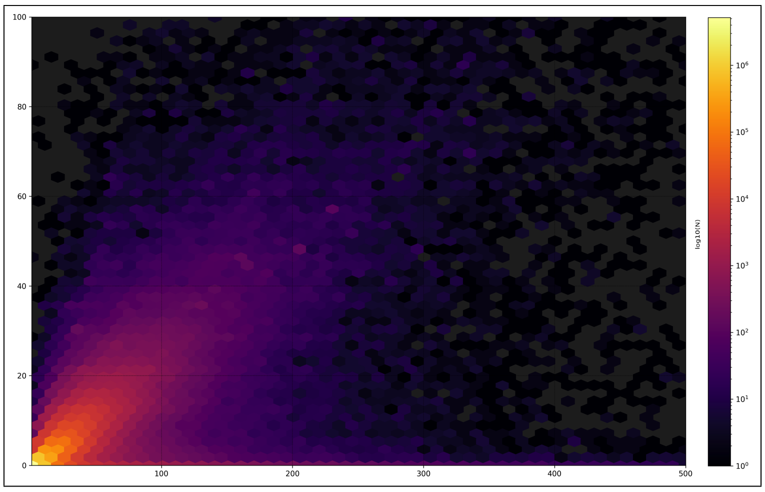


Fig. 5 Hexbin plot of cyclomatic complexity (y-axis) vs. source lines of code (x-axis)

In source code, other issues might be present, e.g., a random sample of methods may contain a lot of small methods (including many easy to learn getters and setters), which may inflate performance. For instance, the performance of method naming models is much higher on very short methods (3 lines), than it is for longer methods (Alon et al 2018). To mitigate this, `JEMMA` can be used to leverage the already existing properties to empirically investigate the performance of models on the tasks and get insights. In the case of the code complexity example, Figure 5 shows the relationship between the size of methods and their complexity as a hexbin plot. From it, we observe that there is an overall tendency for shorter methods to be less complex, and longer methods to be more complex. On the other hand, there also methods that are very long, but have a very low complexity (along the bottom axis). This information can be used to properly balance the data, for instance, by making sure that examples that are short and complex, and examples that are long and simple, are also included in the training and evaluation datasets.

Avoiding data leakage. A specific source of bias that has been found in source code data is data leakage. Multiple studies have shown that code duplication is prevalent in large source code corpora (Schwarz et al 2012), (Lopes et al 2017), and that it impacts machine learning models (Allamanis 2019). Since `JEMMA` is built on top of `50K-C`, we benefit from its selection of projects, which intentionally limited duplication.

Furthermore, since source code is more repetitive within project than across projects it could also be a potential source of data leakage. Models that are trained and tested with files from the same project can see their performance affected (LeClair and McMillan 2019). Since `JEMMA` keeps the metadata of which project a method belongs to, it is easy to define training, validation and test splits that all contain code from different projects, if necessary.

4.2.2 Defining tasks based on representations

`JEMMA` can also be used to define tasks that operate on the source code representations themselves, rather than predicting a source code property. These tasks are usually of two forms: a) masked code prediction tasks, and b) mutation detection tasks.

(a) **Masked code prediction tasks.** In a masking task, one or more parts of the representation are masked with a special token (e.g., "`<MASK>`"), and the model is tasked with predicting the masked parts of the representations. Examples of this would include the method naming task, where the name of the method is masked, or a method call completion task, where a method call is masked in the method's body. A simpler variant of this would be to use a multiple-choice format, where the model has to recognize which of several possibilities is the token that was masked (e.g., an operator is masked in the representation, and the model should choose whether the `<=` or the `>=` operator was masked).

(b) **Mutation detection tasks.** In a mutation detection task, the representation is altered with a fixed probability, presumably in a way that would cause a bug (for instance, two arguments in a method call can be swapped (Pradel and Sen 2018)). The task is to detect that the representation has been altered. This can either be formulated as a binary classification task (altered vs not altered), or, as a "pointing" task, where the model should learn to highlight which specific portion of the given input was altered (Vasic et al 2019).

For both of these tasks, the input representation needs to be modified in some way. `JEMMA` can help in this. For simple modifications (e.g., masking the first occurrence of an operator), it is enough to directly change the default textual representation, and then use the `JEMMA` toolchain and `API` to re-generate the other representations.

For more complex changes, the feature graph representation and the call graph representation can be used to perform queries on the program structure and determine which parts of the input should be masked. It is then possible to navigate back to the raw textual representation (since the feature graph representation contains the information of where the token is located), perform the masking operation, and then re-generate the derived representations. Snippet A.2, in the appendix, shows an example of how to generate new representations for a simple masking task—method call completion.

When doing these kinds of changes, particular care has to be given to data leakage issues. For instance, for a method naming task, the name of the method should be masked in the method's body if it occurs there. Other bias issues can affect these tasks as well, such as a method naming task that overemphasizes performance on getters and setters. `JEMMA` can be used to analyse the performance of the models on the task and extract insights that may affect the design of the task (Section 5.2 provides such an example).

4.2.3 Running models

Once a task is defined, the `JEMMA API` eases the running of models on the task. Several basic baselines are implemented and can be easily run on a task. These include: Multi Layer Perceptrons (`MLPs`); Convolutional Neural Networks (`CNNs`); uni-directional and bi-directional Long Short-Term Memory (`LSTM`) Recurrent Neural Networks, among others.

The `JEMMA API` also facilitates the interaction with other libraries, in particular to run models using the code2vec and code2seq architectures, as well as Graph Neural Network (`GNN`) models implemented with the ptgnn[10] library.

Finally, since models based on the transformer architecture, currently, have been the state of the art for a variety of tasks, `JEMMA` allows to easily interface with HuggingFace's Transformer library (Wolf et al 2019b). This allows a variety of pre-trained models to be fine-tuned on the tasks defined with `JEMMA` (such as `CodeBERT` (Feng et al 2020), `GraphCodeBERT` (Guo et al 2020) etc.). Snippet A.3 in the appendix shows how to run a Transformer model on the method complexity task using the `JEMMA API`.

4.2.4 Defining representations with larger contexts

One of our goals with `JEMMA` is to allow experimentation with novel source code representations. In particular, we want users to be able to define representations that can take into account a larger context than a single method, or a single file, as is done with the vast majority of work today.

The key to building such extended representations is to have access to the necessary contextual information. The extensive pre-processing we did to create `JEMMA` gives us all the relevant tools to gather that information. The metadata of `JEMMA` documents the containment hierarchies (e.g., which files belong to which project, and which classes belong to which package etc.) and provides the ability to uniquely and unambiguously identify source code entities at different granularities. In addition, the call graph data documents which are the immediate callers and callees of each individual method. Since the call graphs link to each method identified by their `UUID`, all the properties of the methods, including their representations, can be accessed easily and systematically. Thus, from navigating the call graph and the containment hierarchy, various types of global contexts can be defined at the class-, package-, or even project-level. We present two simple examples in the appendix.

Snippet A.4 shows how to combine representations of a given method with the representations of its direct callees to include greater context. We encourage users to experiment with more complex representations adding context information that go beyond a single method. The extensive pre-processing of data, at the scale of tens of thousands of projects, combined with the `JEMMA` toolchain and `API` makes it possible to do so easily.

[10] https://github.com/microsoft/ptgnn

5 On the extent of non-localness of software

Having presented the structure of the JEMMA dataset and its capabilities in the previous two sections, we now turn our focus on the need for this dataset (and workbench) through empirical studies.

We first study the extent to which software is made up of interacting functions and methods in a sample of projects contained in JEMMA by analysing their call graphs. We observe how often method calls are local to a file, cross file boundaries, or are calls to external APIs. Then, we analyze the performance of the method call code completion task through the lens of call types.

Each time a method call is executed, control flow jumps from the original method to another method. Relative to the source code location of the calling entity, the called entity can be located: a) in the same file; b) in the same module or package; c) in a different module or package of the same project; or d) in an external library. We study the prevalence of these categories of calls on a subset of our data, showing that, indeed, software is strongly non-local according to this definition.

Based on this, we use the JEMMA API to analyze the performance of some source code models on a code task. Specifically, the method call completion task—in which the a random method call in the source code is masked and the model must predict the correct method call token to succeed. We observe that all of our baselines perform significantly better on calls to external APIs than local calls (Section 5.2).

5.1 Characteristics of method calls in Java

To study the non-localness of method calls, we take a sample of projects from JEMMA, and measure the frequency of the various types of calls in these projects. To account for possible differences due to project sizes, we select our sample projects equitably: 100 small projects with less than 20 classes, 100 medium-sized projects with 21-50 classes, 100 projects with 51-100 classes, and 100 projects with 100+ classes.

For each method defined in a project, we first count the number of callers and number of callees in the call graph. Figure 6 shows the percentage of methods and their direct caller/callee counts. We observe that significantly high percentage of methods are not explicitly called, meaning, they have no callers defined the project. There can be several reasons for this. Some methods are called by frameworks, such as unit tests or event handlers. Other methods rely on the same base mechanism (polymorphism) but the base class is defined in the project. Since our call graphs only have links to the method that can be statically determined, there are no links to the overridden methods in the subclasses. We are currently investigating how to enrich our model to add additional links to these methods. Finally, some methods may be called by reflection; others are really never called, constituting *dead code.*

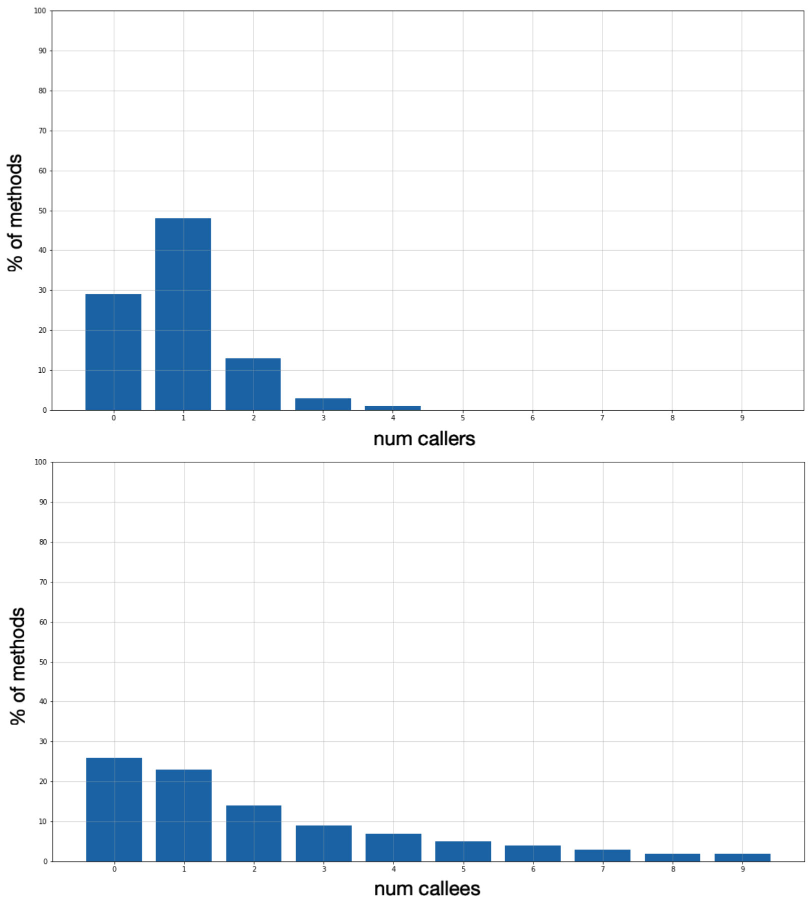


Fig. 6 Number of callers and callees of a sample of the dataset

Regardless, from the count of the number of callers in Figure 6, we notice that the greater majority of the methods have at least one or two unique caller entities: indicating that a significant number of caller methods rely on these callee methods for their functional tasks. Adding context from callee methods could help models reason on a broader context of information.

From the count of the number of callees in Figure 6, we observe that a little more than a quarter of methods have no callees in the project, thus being either purely local or relying only on external calls beyond the project. We intentionally exclude external api or library calls in the caller-callee counts, since we want to study the calls defined in the context of the project.

The figure shows that the greater majority of methods indeed do have calls to other methods defined in the project. Thus, models that wish to reason on software at a more detailed level must take this into account.

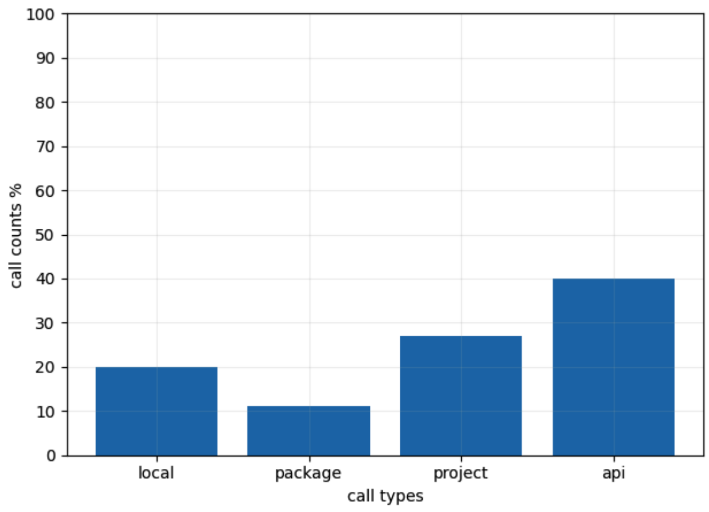


Fig. 7 Distribution of calls by type

To study the non-localness of method calls in the context of the project, we classify all calls into four categories:

- *Local calls.* The entity is defined in the same file; thus, a machine learning model that has a file context would be likely to see it.
- *Package calls.* The entity is defined in the same Java package (i.e., the classes as in the same file directory).
- *Project calls.* The called entity is defined in the project, but in a different package than the caller.
- *API calls.* The called entity is not defined in the project, but is a call to an imported library.

Figure 7 shows the distribution of the calls. We can see that 20% of calls are *local calls*; these are the calls whose callees are visible to the models that learn from the entire file context, such as CoCoGum (Wang et al 2020); the remaining 80% of calls are *non-local* and are not visible to models that learn from the file context only. Of these, 12% are *package calls*; thus a model that builds a context of the classes in the same directory to absorb a larger context than the file would have visibility into these callees. On the other hand, 28% of calls are *project calls*, thus models would need either a larger context, or the ability to select from this larger context in order to have visibility in the callees. Finally, *API calls* constitute 40% of all calls (inflated by the vast majority of standard library calls). While these are out of reach for most models, a silver lining is that, in practice, it is often possible to learn from their usage in other projects, as modern large-scale source code models do.

Table 6 Performance of the method call completion task according to method call type

Models	*Local*	*Package*	*Project*	*API*
`MLP`	0.259	0.111	0.143	0.330
`CNN`	0.236	0.140	0.203	0.462
`LSTM`	0.299	0.154	0.228	0.524
`codeBERTa`	0.530	0.265	0.388	0.857

5.2 Impact of non-localness on code completion

Having some insight on the characteristics of method calls in Java, we turn to the issue of whether this can have an impact on the performance of models. The study of Hellendoorn et al (2019a) investigated the differences between code completions from synthetic benchmarks and real world completions. It found that synthetic benchmarks that evaluate models on all the tokens in a corpus underweighted the frequencies of method completions relative to real-world completion, and that those were the most difficult. They also observed that among method completions, the hardest ones were the ones from project internal identifiers.

Hellendoorn's study offers valuable insights but has limitations. It motivates our choice to design a code completion task with much more data-points that focuses exclusively on methods. Since the data for the real world completions was obtained by monitoring developers, it is extremely rich and of very high quality, but relatively small (15,000 completions from 66 developers). Second, the study evaluated RNNs with a closed vocabulary, which were thus unable to learn new identifiers. Since then, open-vocabulary RNNs (Karampatsis et al 2020) and Transformers (Ciniselli et al 2021) have considerably improved the state of the art.

We thus use `JEMMA` to analyse the performance of recent models on a variant of the code completion task, specifically method call completion task. Informed by the results of the previous section, we analyse separately the performance of models on different strata of the test set, according to the categories defined above: *local calls*, *package calls*, *project calls* and *API calls*.

Task definition: We define the method call completion task as follows. We extract 1,000,000 methods from `JEMMA`, keeping 800,000 as training set, 100,000 as validation set, and 100,000 as test set. We define a masking task: for each method, we mask one single method call at random. These methods can be present in the same class (18% of the dataset), in another class in the same package (10%), in another package in the system (26%), or imported from a dependency (46%). The goal of the task is for a source code model to predict the *exact* method name that was masked.

We then analyze the performance of three (very simple) baselines: an `MLP`, a `CNN`, and a Bidirectional `LSTM`. Each of these model is of a small size, and has a closed vocabulary of the 10,000 most common tokens in the training set. We also include a model close to the state of the art: `CodeBERTa` , fine-tuned on our dataset. The accuracy on the test set is shown in Table 6. We can clearly see that the performance of the various models are much higher on the *API calls* than the other categories, with the second highest being the *local calls*; the *project calls* and the *package calls* having the lowest performance. While we can expect that different models would perform differently, the margin between *API calls* and the other types of calls is clear enough to show that the models perform much better at predicting *API calls* that can be learned from other projects, than calls defined in the project.

5.3 Implications

Thus, drawing from the study on the characteristics of calls in Section 5.1, specifically from the caller/callee numbers, and concluding from the analysis results in Section 5.2, we see that: a) a large number of method contexts are non-local, and b) source code models struggle to predict call completions of methods defined in the same project. This nudges us to explore the notion of designing and training source code models in a way that it can reason over a larger context of information, at least at the project-level. It becomes necessary to determine ways in which models could be made aware of the inter-relationships that exist among code entities by providing a feature-rich representation with as much context information that we can possibly fit.

6 OOW is the next OOV

The studies in the previous section show that software entities have complex relationships that can affect the performance of models. This section provides data to inform the design of possible architectures that can absorb a larger context beyond source code entities at the method-level granularity. In particular, we focus on the *size* of this context, as deep learning models can be strongly affected by the input size.

Machine learning models of code once struggled with *Out-Of-Vocabulary* (`OOV`) issues (Hellendoorn and Devanbu 2017), until more recent models introduced and adopted an open vocabulary (Karampatsis et al 2020). We argue that the next problem to address is the *Out-Of-Window* (`OOW`) issue: all modern state-of-the-art models tend to have a fixed input size, which may not be enough to fit the context needed. How to best use this limited resource is thus, an open question.

6.1 Transformers, window sizes, and tokenizations

For many machine learning tasks, Transformers (Vaswani et al 2017) are now the state of the art. Some transformer models that have achieved state-of-the-art performance on source code tasks include include `CodeBERT` (Feng et al 2020), `CodeBERTa` (Wolf et al 2019a), `PLBART` (Ahmad et al 2021) and

GraphCodeBERT (Guo et al 2020). Codex (Chen et al 2021) is yet another of these large pre-trained Transformer models, that has demonstrated compelling competence on a variety of tasks without necessarily needing fine-tuning, ranging from program synthesis and program summarization (Chen et al 2021) to program repair (Prenner and Robbes 2021).

However, all Transformers that follow the classic architecture have a fixed window sizes: for CodeBERT, it is 512 tokens, while for the largest Codex model (codex-davinci), it is 4,096 tokens (a smaller version, codex-cushman, has a window of 2,048 tokens). If an input is longer than the window, it is truncated. Furthermore, the size of the window has a major impact on the performance of the model. Transformers rely on self-attention, where the attention heads *attend* to each pair of tokens: the complexity is hence quadratic, which renders very large windows prohibitive in terms of training time and inference time. This raises the question: for a given window size, how much code can we expect to fit?

Since Transformers are open-vocabulary models, the tokens that they take as input are actually subtokens, common subsequences of characters learned from a corpus, rather than entire tokens. A word that would be unrecognized by a closed-vocabulary model will, instead, be split up in several more common subtokens. This means that the number of lexical tokens in a method does not match the length of the method in terms of subtokens, and depends on the corpus that was used to train the subword tokenizer. It is important to note that both CodeBERT and Codex are not models trained from scratch on source code: given the amount of time needed to train such a model from scratch, previous models trained on English (RoBERTa for CodeBERT, a version of GPT-3 for Codex) were fine-tuned on source code instead. This means that both CodeBERT and Codex use a subword tokenizer that was not learned for source code, but for English, which might lead to a sub-optimal tokenization.

To estimate the number of tokens that a method will take in the model's input window, we first selected a sample of 200,000 Java methods, and used several subword tokenizers to estimate the ratio of subtokens that each subword tokenizer will produce. We first noticed that the choice of subword tokenizer has a significant impact on the produced tokenization, and consequently the amount of code that can fit in a model's input window. We used the following tokenizers for our analyses.

- RoBERTa tokenizer. A byte-level BPE tokenizer, trained on a large English corpus, with a vocabulary of slightly more than 50,000 tokens. A similar tokenizer is used by CodeBERT and Codex.
- CodeBERTa tokenizer. The tokenizer used by CodeBERTa. This tokenizer was trained on source code from the CodeSearchNet corpus, which comprises of 2 million methods in 6 programming languages, including Java.
- Java BPE tokenizer. A tokenizer similar to CodeBERTa tokenizer, trained on 200,000 Java methods from Maven, instead of several languages.
- Java Parser. A standard tokenizer from a Java Parser, that does not perform sub-tokenizations. We use this as a baseline for our analyses.

By comparing the average size of the tokenizations to the actual number of tokens, we find that the source-code specific tokenizations are considerably more efficient than the English one. The `CodeBERTa tokenizer` learned on multiple programming languages is, on average, slightly better, using 98% of the tokens than the `Java Parser` tokenization. This is possible since some common token sequences can be merged in a single token (e.g, *();* can be counted as one token instead of three tokens). The learned `Java BPE tokenizer` is even more efficient, using on average 85% of the tokens. This is possible since, for instance, specific class names will be common enough that they can be represented by a single token (e.g., `ArrayIndexOutOfBoundsException`). On the other hand, the `RoBERTa tokenizer` is considerably less efficient, needing 126% of the lexical Java tokens.

Thus, the tokenization used can have a *significant impact* on the effective window size of the models. With an equal vocabulary size, the most efficient, language-specific encoding can fit close to 50% more effective tokens in the same window size. For a window size of 512, a Java-specific tokenizer will, on average, be able to effectively fit 602 actual tokens, while the English-specific tokenizer—used by both `CodeBERT` and `Codex`—will be able to fit only 409 actual tokens.

6.2 Fitting code entities

Taking the same 400 projects as in the code completion study, we tokenize the methods and the classes in these projects with the four tokenizers above. We then estimate the size of higher-level entities (packages and projects) by summing the token sizes of the classes in them. We compare these sizes with various Transformer window size thresholds.

- ◇ *Small*. A window size of 256 tokens, representing a small transformer model
- ◇ *Base*. A window size of 512 tokens, representing a model with the same size as `CodeBERT` (Feng et al 2020).
- ◇ *Large*. A window size of 1,024 tokens, which is the context size used by the largest `GPT-2` model (Radford et al 2019).
- ◇ *XL*. A window size of 2,048 tokens, which is the context size used by the largest `GPT-3` model (Brown et al 2020).
- ◇ *XXL*. A window size of 4,096, which is the context size used by the largest `Codex` model (Chen et al 2021).

It is important to note that these models are very expensive to train. In practice, training a model with a *Base* window size of 512 tokens, from scratch, is a significant endeavour inaccessible for most academic groups, leaving fine-tuning as the only practical option. Only industry research groups or large consortium of academics may have the resources necessary to train such large-scale models. The largest models reach sizes for which even doing inference becomes impractical.

6.2.1 Methods

Figure 8 (a) shows the percentage of methods that fit within different window size thresholds. We can see that even the *Small* model is able to comfortably fit the vast majority of methods (over 94%). The choice of tokenization still matters, as a more efficient tokenization can make up to 97% of methods fit in the *Small* model. Overall, a *Base* model with a window size of 512 tokens can fit 99% of the methods in our sample, while only extreme outliers do not fit even in the *XXL* models with a limit of 4096 tokens.

6.2.2 Classes

We tokenize the entire source file to compute the context size needed for classes. Figure 8 (b) shows the percentage of classes that fit within different window size thresholds. We can see that models with the smaller window sizes are beginning to struggle. A *Small* model with a token limit of only 256 tokens will be able to process between 47-59% of the classes. A *Base* model would, instead be able to process between 68 and 78% of the classes, while a *Large* model would fit up to 90% of the classes. *XL* models can fit almost more than 95% of the classes on average, but some outliers (2-3%) will remain even for a Codex-sized model.

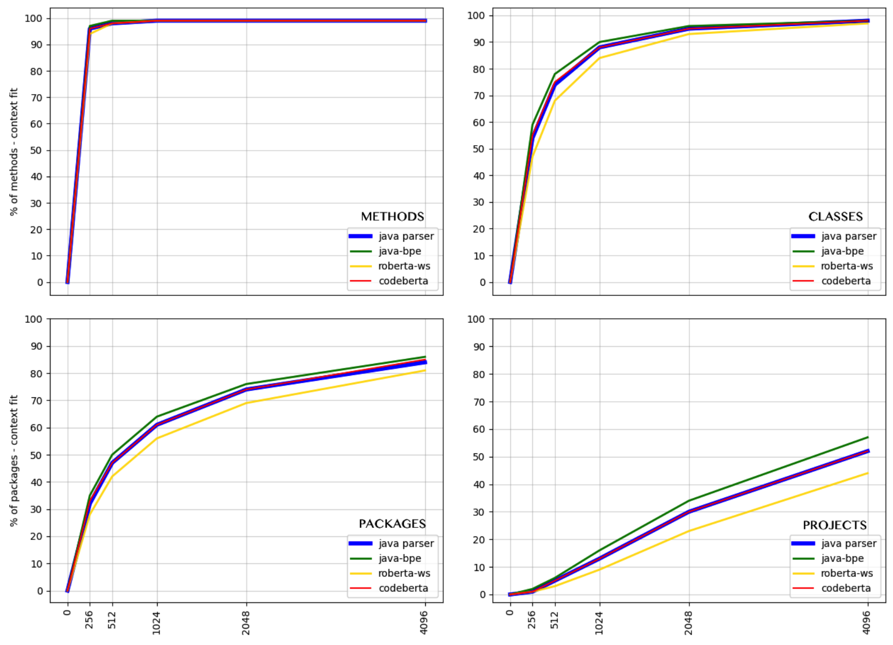


Fig. 8 Percentage of context-fit for: (a) methods; (b) classes; (c) packages; (d) projects.

6.2.3 Packages

Figure 8 (c) shows the percentage of packages that fit within different window size thresholds. Models with smaller window sizes struggle significantly, with a *Small* model able to fit only a 30 to 35% of the packages, and a *Base* model 42 to 50%, depending on the tokenization. A *Large* model succeeds in 55 to 65% of the cases. We can clearly see that even the models with largest token limits start to struggle while fitting packages into context: 69-76% fit in a window size of 2048 tokens, and 81-86% fit in a window-size of 4096 tokens.

6.2.4 Projects

On average only half the projects can fit in the window sizes (Figure 8 (d)). But since we expect that larger projects would behave differently, we present a context-fit graph for projects based on size (Figure 9). We observe that while models with large window sizes are able to fit 66-81% of small-sized projects that have 20 or less classes, the rate drops drastically as project size increases—falling to 14-28% for medium-sized projects. Beyond this, very few (less than 6%) of the larger projects can fit for any window-size. Of note, the largest projects that do not fit the model window sizes, being the most complex, are likely the ones for which the source code models might be the most useful.

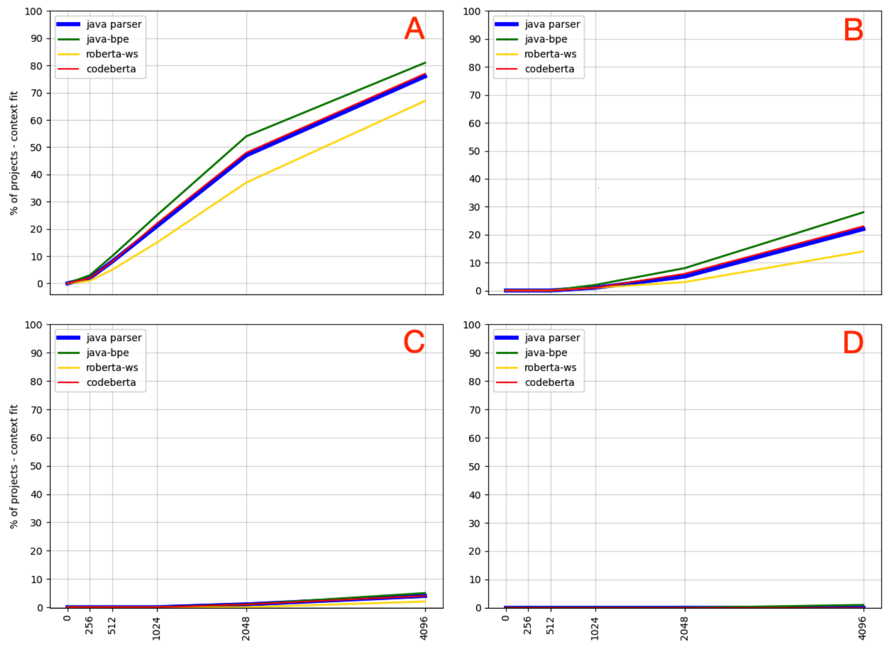


Fig. 9 Percentage of context-fit for full projects by project sizes. A: up to 20 classes; B: 21-50 classes; C: 51-100 classes, D: more than 100 classes.

6.3 Implications

Clearly, significant work is needed to find architectures that can fit contexts at the project-level, especially if the model size is to be kept small enough to be manageable. We also observe that while large-scale models can comfortably fit most methods and classes, they still struggle with fitting larger contexts.

Furthermore, we find that a model that efficiently encodes its code input using a code-specific tokenizer, is able to encode the same data in somewhat less space. This leads to a greater amount of context-fitting. Therefore, we need to encourage researchers and model architects to adopt such changes, instead of relying on sub-tokenizations from tokenizers trained on English text.

It's worth noting that classical Transformer models exhibit a quadratic complexity in terms of the input size due to the attention layers. This contributes to their issues in scaling beyond a threshold limit. Thus, reasoning at the scale of packages or projects would require a rethink of the architecture, such as using a Transformer variant which better handles longer sequences such as a Reformer (Kitaev et al 2020), or another *efficient* Transformer (Tay et al 2020b) which exhibits lower complexities as input size increases. Whether this is sufficient is uncertain: *efficient* transformers can struggle with very long sequences, as exhibited in specialized benchmarks (Tay et al 2020a).

While we focused specifically on Transformers as they have a fixed context size window, other models will also be challenged by large input sizes. The `ASTs` and graph representations of classes, packages, and projects will also have scaling issues as the number of nodes to consider will grow very quickly. Furthermore, Graph Neural Networks can also struggle with long-distance relationships in the graph (Alon and Yahav 2020).

On the other hand, we see promise in an approach that is able to select the input relevant to the task. Of note, recent work has started to go in this direction for code summarization, both at the file level (Clement et al 2021) and multiple files (Bansal et al 2021). Significant work lies ahead in devising techniques that truly take into account a larger global context (Figure 10), thus addressing the *"Out-Of-Window"* (`OOW`) problem; at a minimum, `JEMMA` provides the tools to investigate this.

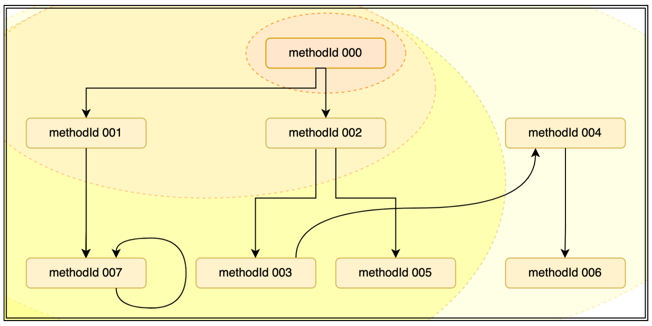


Fig. 10 Step-wise inclusion of greater context information

7 Limitations

JEMMA is the only effort we are aware of in gathering enough data that is pre-processed sufficiently to enable empirical research of machine learning models that can reason on a more global context than the file or method level. Nevertheless, it has several limitations. Some of these issues are inherited from our use of 50K-C, while others are due to limitations in our pre-processing; while the former will be hard to overcome (barring extensive additional data collection), the latter could be mitigated by further processing from our side.

7.1 Limitations stemming from the use of 50K-C

Monolingual. JEMMA is comprised of projects in the Java programming language only. This poses issues as to whether models that work well for Java would also work well for other languages. The reason for this limitation is twofold: 1) adding other languages at a similar scale would drastically increase the already extremely significant time we invested in pre-processing data, and 2) restricting to one language frees us from tooling issues: we don't need to settle on a "common denominator" in tool support (e.g., Infer supports few programming languages, and many of its analyses are limited to a single programming language).

Monoversion. JEMMA is comprised of snapshots of projects, rather than multiple project versions. This prevents us from using it for tasks that would rely on multiple versions, or commit data, such as some program repair tasks. On the other hand, this frees us from issues related to the evolution of software systems, such as performing origin analysis (Godfrey and Zou 2005), which is essential as refactorings are very common in software evolution, and can lead to discontinuities in the history of entities, particularly for the most changed ones (Hora et al 2018). Omitting versions also considerably reduces the size of the dataset, which is already rather large as it is.

Static data only. While the projects included in 50K-C were selected because they could be compiled, 50K-C provide no guarantees that they can be run. Indeed, it is hard to know if a project can run, even if it can be compiled. In case it can run, the project likely expects some input of some sort. This leaves running test cases as the only option to reliably gather runtime data. In our previous work in Smalltalk, where we performed an empirical study of 1,000 Smalltalk projects, we could run tests for only 16% of them (Callaú et al 2014). Thus, JEMMA makes no attempt at gathering properties that comes from dynamic analysis tools at this time. In the future, JEMMA's property mechanism could be used to document whether a project has runnable test cases, as a first step towards gathering runtime information. We could also expand the dataset with the 76 projects coming from XCorpus, which were selected because they are runnable (Dietrich et al 2017).

7.2 Limitations stemming from our pre-processing

Incomplete compilation. While the projects in `50K-C` were selected because they were successfully compiled, we were not able to successfully recompile all of them. Roughly 18% of the largest projects could not be compiled; this number trends down for smaller projects. We are not always sure of the reasons for this, although we suspect that issues related to dependencies might come into play. This may add a bias to our data, in case the projects that we are unable to compile are markedly different from the ones that we could compile.

Imprecisions in call graphs. The call graph extraction tool that we use has some limitations that we inherit. In particular, handling methods called via reflection is a known problem for static analysis (Bodden et al 2011); the call graph extraction tool does not handle these cases. A second issue is related to polymorphism, where it is impossible to know, in the absence of runtime information, which of the implementations can be called. In this case, our call graph has an edge to the most generic method declaration.

Inner classes. Our handling of inner classes is limited. Since inner classes are contained in methods, the models can have access to their definitions. However, we do not assign `UUIDs` to them or to the methods defined in them, as this would significantly increase the complexity of our model (in terms of levels of nesting in the hierarchy), while these cases are overall rare. Additional pre-processing could handle these cases, but we do not expect this to become necessary.

Class-level data. Since most machine learning models of code take methods as inputs, we gave priority to this representation in our dataset. As a consequence, our modeling of classes and packages is limited. While information about, for instance, the class attributes is not explicitly modelled at this time, it is easily accessible in the file-level feature graph representation, so that models that wish to use this information can access it.

Incomplete preprocessing. At the time of writing, not all the representation data is present for all the projects, due to the very computationally expensive pre-processing that is needed. We started with the largest projects, and worked our way down to the smaller ones. All of the basic metadata is present for all of the projects. However, some of the smaller projects (the ones with less than 20 classes) will have their feature graph representation computed and added to `JEMMA` in the coming weeks. A second category of incomplete pre-processing is that some tools will very occasionally fail on some very specific input (e.g., the parser used by an analysis tool may handle some edge cases differently than the official parser).

8 Conclusion

In this article, we presented JEMMA, a dataset and workbench to support research in the design and evaluation of source code machine learning models. Seen as a dataset, JEMMA is built upon the 50K-C dataset of 50,000 compilable Java projects, which we extend in several ways. We add multiple source code representations at the method level, to allow researchers to experiment on the effectiveness of these, and their variations. We add a project-level call graph, so the researchers can experiments with models that consider multiple methods, rather than a single method or a single file. Finally, we add multiple source code properties, obtained by running source code static analyzers—ranging from basic metrics to advanced analyses characteristics based on abstract interpretation.

Seen as a workbench, JEMMA APIs help achieve a variety of objectives. JEMMA can extend itself with new properties and representations. It can be used to define machine learning tasks, using the properties and the representations themselves as basis for prediction tasks. The properties defined in JEMMA can be used to get insight on the performance of tasks and pinpoint possible sources of bias. Finally, JEMMA provides all the tools to experiment with new representations that combine the existing ones, allowing the definition of models that can learn from larger contexts than a single method snippet.

Alongside, we have provided examples of usage of JEMMA. We have shown how JEMMA can be used to define a metric prediction and a method call completion task. We have also shown how JEMMA can be used for empirical studies. In particular, we investigated how the performance of our code completion task was impacted by the type of identifier to predict, showing that models performed much better on API methods than on methods defined in the project, indicating the need for models that take into account the project's context. Finally, we have shown that taking into account this global context will be challenging, by studying its size. While state of the art transformer models such as CodeBERT can fit most methods in the dataset, fitting package-level or higher context is much more challenging, even for the largest models such as OpenAI's Codex model. This indicates that significant effort lies ahead in defining models able to process this amount of data, a task that we hope JEMMA will support the community in achieving.

References

Ahmad W, Chakraborty S, Ray B, Chang KW (2021) Unified pre-training for program understanding and generation. In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Association for Computational Linguistics, DOI 10.18653/v1/2021.naacl-main.211

Allamanis M (2019) The adverse effects of code duplication in machine learning models of code. In: Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, pp 143–153

Allamanis M, Sutton C (2013) Mining source code repositories at massive scale using language modeling. In: 2013 10th Working Conference on Mining Software Repositories (MSR), IEEE, pp 207–216

Allamanis M, Brockschmidt M, Khademi M (2017) Learning to represent programs with graphs. arXiv preprint arXiv:171100740

Allamanis M, Barr ET, Devanbu P, Sutton C (2018) A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR) 51(4):1–37

Allamanis M, Barr ET, Ducousso S, Gao Z (2020) Typilus: Neural type hints. In: Proceedings of the 41st acm sigplan conference on programming language design and implementation, pp 91–105

Alon U, Yahav E (2020) On the bottleneck of graph neural networks and its practical implications. arXiv preprint arXiv:200605205

Alon U, Brody S, Levy O, Yahav E (2018) code2seq: Generating sequences from structured representations of code. arXiv preprint arXiv:180801400

Alon U, Zilberstein M, Levy O, Yahav E (2019) code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages 3(POPL):1–29

Bansal A, Haque S, McMillan C (2021) Project-level encoding for neural source code summarization of subroutines. arXiv preprint arXiv:210311599

Barone AVM, Sennrich R (2017) A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. arXiv preprint arXiv:170702275

Beery S, Van Horn G, Perona P (2018) Recognition in terra incognita. In: Proceedings of the European conference on computer vision (ECCV), pp 456–473

Bodden E, Sewe A, Sinschek J, Oueslati H, Mezini M (2011) Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: 2011 33rd International Conference on Software Engineering (ICSE), IEEE, pp 241–250

Brown TB, Mann B, Ryder N, Subbiah M, Kaplan J, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, et al (2020) Language models are few-shot learners. arXiv preprint arXiv:200514165

Calcagno C, Distefano D, Dubreil J, Gabi D, Hooimeijer P, Luca M, O'Hearn P, Papakonstantinou I, Purbrick J, Rodriguez D (2015) Moving fast with software verification. In: NASA Formal Methods Symposium, Springer, pp 3–11

Callaú O, Robbes R, Tanter E, Röthlisberger D, Bergel A (2014) On the use of type predicates in object-oriented software: The case of smalltalk. In: Proceedings of the 10th ACM Symposium on Dynamic languages, pp 135–146

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 preprint arXiv:210703374

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. arXiv preprint arXiv:210307115

Clement CB, Lu S, Liu X, Tufano M, Drain D, Duan N, Sundaresan N, Svyatkovskiy A (2021) Long-range modeling of source code files with ewash: Extended window access by syntax hierarchy. arXiv preprint arXiv:210908780

Cordy JR, Roy CK (2011) The nicad clone detector. In: 2011 IEEE 19th International Conference on Program Comprehension, IEEE, pp 219–220

De Roover C, Lämmel R, Pek E (2013) Multi-dimensional exploration of api usage. In: 2013 21st International Conference on Program Comprehension (ICPC), IEEE, pp 152–161

DeFreez D, Thakur AV, Rubio-González C (2018) Path-based function embedding and its application to error-handling specification mining. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 423–433

Dietrich J, Schole H, Sui L, Tempero E (2017) Xcorpus–an executable corpus of java programs

Dyer R, Nguyen HA, Rajan H, Nguyen TN (2013) Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: 2013 35th International Conference on Software Engineering (ICSE), IEEE, pp 422–431

Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, et al (2020) Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:200208155

Godfrey MW, Zou L (2005) Using origin analysis to detect merging and splitting of source code entities. IEEE Transactions on Software Engineering 31(2):166–181

Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, et al (2020) Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:200908366

Gururangan S, Swayamdipta S, Levy O, Schwartz R, Bowman SR, Smith NA (2018) Annotation artifacts in natural language inference data. arXiv preprint arXiv:180302324

Habib A, Pradel M (2018) How many of all bugs do we find? a study of static bug detectors. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 317–328

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

Hellendoorn VJ, Proksch S, Gall HC, Bacchelli A (2019a) When code completion fails: A case study on real-world completions. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp 960–970

Hellendoorn VJ, Sutton C, Singh R, Maniatis P, Bieber D (2019b) Global relational models of source code. In: International conference on learning representations

Hindle A, Godfrey MW, Holt RC (2008) Reading beside the lines: Indentation as a proxy for complexity metric. In: 2008 16th IEEE International Conference on Program Comprehension, IEEE, pp 133–142

Hindle A, Barr ET, Gabel M, Su Z, Devanbu P (2016) On the naturalness of software. Communications of the ACM 59(5):122–131

Hora A, Silva D, Valente MT, Robbes R (2018) Assessing the threat of untracked changes in software evolution. In: Proceedings of the 40th International Conference on Software Engineering, pp 1102–1113

Hovemeyer D, Pugh W (2004) Finding bugs is easy. Acm sigplan notices 39(12):92–106

Husain H, Wu HH, Gazit T, Allamanis M, Brockschmidt M (2019) Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:190909436

Iyer S, Konstas I, Cheung A, Zettlemoyer L (2016) Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp 2073–2083

Kanade A, Maniatis P, Balakrishnan G, Shi K (2020) Learning and evaluating contextual embedding of source code. In: International Conference on Machine Learning, PMLR, pp 5110–5121

Karampatsis RM, Sutton C (2020) How often do single-statement bugs occur? the manysstubs4j dataset. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp 573–577

Karampatsis RM, Babii H, Robbes R, Sutton C, Janes A (2020) Big code!= big vocabulary: Open-vocabulary models for source code. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), IEEE, pp 1073–1085

Kitaev N, Kaiser L, Levskaya A (2020) Reformer: The efficient transformer. arXiv preprint arXiv:200104451

LeClair A, McMillan C (2019) Recommendations for datasets for source code summarization. arXiv preprint arXiv:190402660

LeClair A, Jiang S, McMillan C (2019) A neural model for generating natural language summaries of program subroutines. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp 795–806

Lin XV, Wang C, Zettlemoyer L, Ernst MD (2018) Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. arXiv preprint arXiv:180208979

Lopes CV, Maj P, Martins P, Saini V, Yang D, Zitny J, Sajnani H, Vitek J (2017) Déjàvu: a map of code duplicates on github. Proceedings of the ACM on Programming Languages 1(OOPSLA):1–28

Lämmel R, Pek E, Starek J (2011) Large-scale, ast-based api-usage analysis of open-source java projects. pp 1317–1324, DOI 10.1145/1982185.1982471

Ma Y, Dey T, Bogart C, Amreen S, Valiev M, Tutko A, Kennard D, Zaretzki R, Mockus A (2021) World of code: Enabling a research workflow for mining and analyzing the universe of open source vcs data. Empirical Software Engineering 26(2):1–42

Martins P, Achar R, Lopes CV (2018) 50k-c: A dataset of compilable, and compiled, java projects. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), IEEE, pp 1–5

McCoy RT, Pavlick E, Linzen T (2019) Right for the wrong reasons: Diagnosing syntactic heuristics in natural language inference. arXiv preprint arXiv:190201007

Milojkovic N, Caracciolo A, Lungu MF, Nierstrasz O, Röthlisberger D, Robbes R (2015) Polymorphism in the spotlight: Studying its prevalence in java and smalltalk. In: 2015 IEEE 23rd International Conference on Program Comprehension, IEEE, pp 186–195

Mir AM, Latoskinas E, Gousios G (2021) Manytypes4py: A benchmark python dataset for machine learning-based type inference. arXiv preprint arXiv:210404706

Mou L, Li G, Zhang L, Wang T, Jin Z (2016) Convolutional neural networks over tree structures for programming language processing. In: Thirtieth AAAI Conference on Artificial Intelligence

Palsberg J, Lopes CV (2018) Njr: A normalized java resource. In: Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, pp 100–106

Parr T (2013) The definitive ANTLR 4 reference. Pragmatic Bookshelf

Pietri A, Spinellis D, Zacchiroli S (2019) The software heritage graph dataset: public software development under one roof. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, pp 138–142

Pradel M, Sen K (2018) Deepbugs: A learning approach to name-based bug detection. Proceedings of the ACM on Programming Languages 2(OOPSLA):1–25

Prenner JA, Robbes R (2021) Automatic program repair with openai's codex: Evaluating quixbugs. 2111.03922

Prenner JA, Robbes R (2022) Making the most of small software engineering datasets with modern machine learning. IEEE Transactions on Software Engineering p in press

Puri R, Kung DS, Janssen G, Zhang W, Domeniconi G, Zolotov V, Dolby J, Chen J, Choudhury M, Decker L, et al (2021) Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. arXiv preprint arXiv:210512655

Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I, et al (2019) Language models are unsupervised multitask learners. OpenAI blog 1(8):9

Raemaekers S, Van Deursen A, Visser J (2013) The maven repository dataset of metrics, changes, and dependencies. In: 2013 10th Working Conference on Mining Software Repositories (MSR), IEEE, pp 221–224

Raychev V, Bielik P, Vechev M (2016) Probabilistic model for code with decision trees. ACM SIGPLAN Notices 51(10):731–747

Sajnani H, Saini V, Svajlenko J, Roy CK, Lopes CV (2016) Sourcerercc: Scaling code clone detection to big-code. In: Proceedings of the 38th International Conference on Software Engineering, pp 1157–1168

Sawant AA, Bacchelli A (2017) fine-grape: fine-grained api usage extractor–an approach and dataset to investigate api usage. Empirical Software Engineering 22(3):1348–1371

Schwarz N, Lungu M, Robbes R (2012) On how often code is cloned across repositories. In: 2012 34th International Conference on Software Engineering (ICSE), IEEE, pp 1289–1292

Spinellis D (2017) A repository of unix history and evolution. Empirical Software Engineering 22(3):1372–1404

Svajlenko J, Roy CK (2015) Evaluating clone detection tools with bigclonebench. In: 2015 IEEE international conference on software maintenance and evolution (ICSME), IEEE, pp 131–140

Tay Y, Dehghani M, Abnar S, Shen Y, Bahri D, Pham P, Rao J, Yang L, Ruder S, Metzler D (2020a) Long range arena: A benchmark for efficient transformers. arXiv preprint arXiv:201104006

Tay Y, Dehghani M, Bahri D, Metzler D (2020b) Efficient transformers: A survey. arXiv preprint arXiv:200906732

Tempero E, Anslow C, Dietrich J, Han T, Li J, Lumpe M, Melton H, Noble J (2010) The qualitas corpus: A curated collection of java code for empirical studies. In: 2010 Asia

Pacific Software Engineering Conference, IEEE, pp 336–345

Terra R, Miranda LF, Valente MT, Bigonha RS (2013) Qualitas. class corpus: A compiled version of the qualitas corpus. ACM SIGSOFT Software Engineering Notes 38(5):1–4

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 Transactions on Software Engineering and Methodology (TOSEM) 28(4):1–29

Utture A, Kalhauge CG, Liu S, Palsberg J (2020) Njr-1 dataset. DOI 10.5281/zenodo.4839913, URL https://doi.org/10.5281/zenodo.4839913

Vasic M, Kanade A, Maniatis P, Bieber D, Singh R (2019) Neural program repair by jointly learning to localize and repair. arXiv preprint arXiv:190401720

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems, pp 5998–6008

Wang K, Christodorescu M (2019) Coset: A benchmark for evaluating neural program embeddings. arXiv preprint arXiv:190511445

Wang Y, Du L, Shi E, Hu Y, Han S, Zhang D (2020) Cocogum: Contextual code summarization with multi-relational gnn on umls. Tech. rep., Microsoft, Tech. Rep. MSR-TR-2020-16, May 2020.[Online]. Available: https ...

Wei J, Goyal M, Durrett G, Dillig I (2020) Lambdanet: Probabilistic type inference using graph neural networks. arXiv preprint arXiv:200502161

Wolf T, Debut L, Sanh V, Chaumond J, Delangue C, Moi A, Cistac P, Rault T, Louf R, Funtowicz M, Brew J (2019a) Huggingface's transformers: State-of-the-art natural language processing. CoRR abs/1910.03771, URL http://arxiv.org/abs/1910.03771, 1910.03771

Wolf T, Debut L, Sanh V, Chaumond J, Delangue C, Moi A, Cistac P, Rault T, Louf R, Funtowicz M, et al (2019b) Huggingface's transformers: State-of-the-art natural language processing. arXiv preprint arXiv:191003771

Yao Z, Weld DS, Chen WP, Sun H (2018) StaQC: A systematically mined question-code dataset from Stack Overflow. In: Proceedings of the 2018 World Wide Web Conference, pp 1693–1703

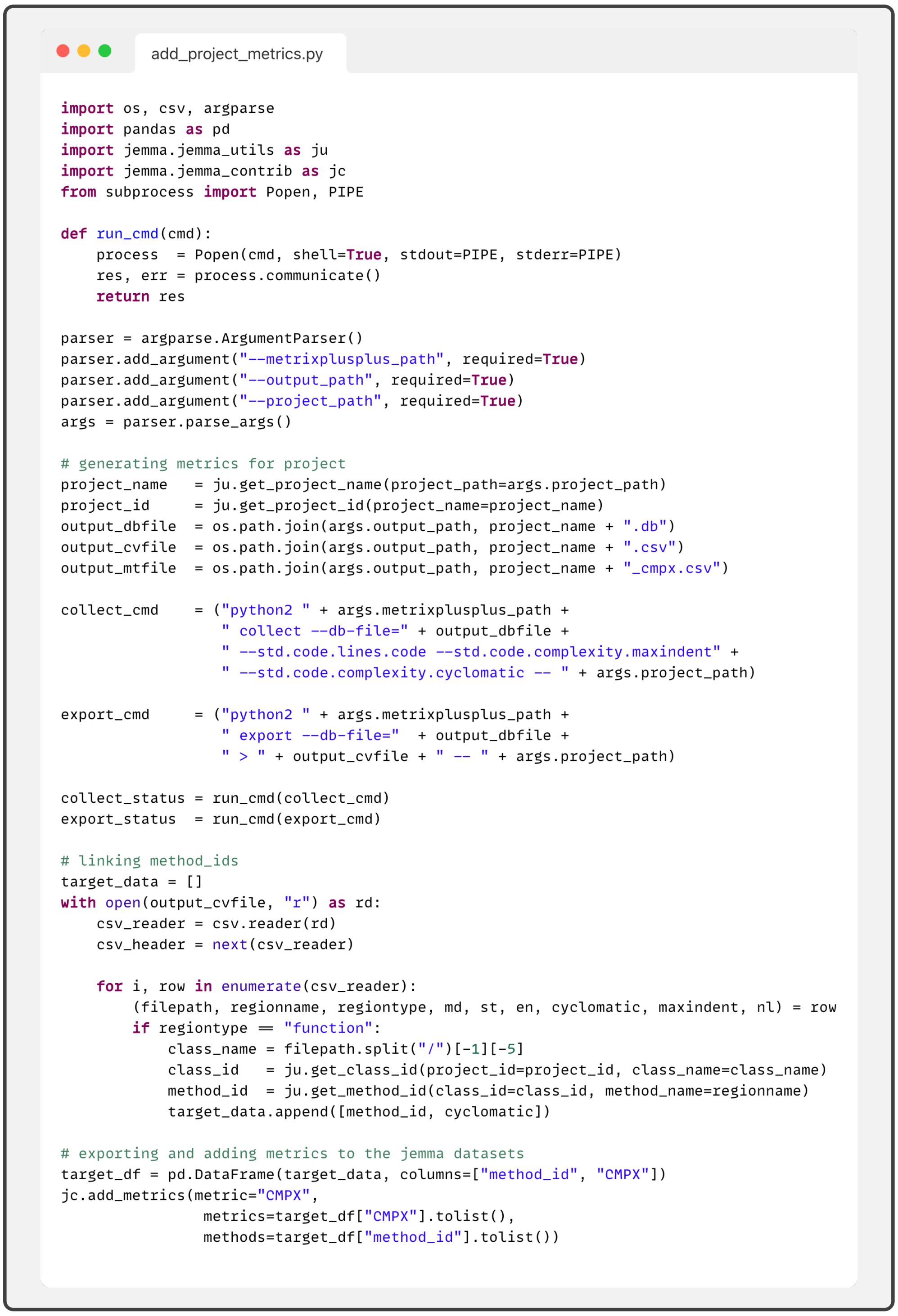
Yin P, Deng B, Chen E, Vasilescu B, Neubig G (2018) Learning to mine aligned code and natural language pairs from stack overflow. In: 2018 IEEE/ACM 15th international conference on mining software repositories (MSR), IEEE, pp 476–486

Yu T, Zhang R, Yang K, Yasunaga M, Wang D, Li Z, Ma J, Li I, Yao Q, Roman S, et al (2018) Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. arXiv preprint arXiv:180908887

Zavershynskyi M, Skidanov A, Polosukhin I (2018) Naps: Natural program synthesis dataset. arXiv preprint arXiv:180703168

Zhou Y, Liu S, Siow J, Du X, Liu Y (2019) Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. arXiv preprint arXiv:190903496

Appendix A Code Examples



add_project_metrics.py

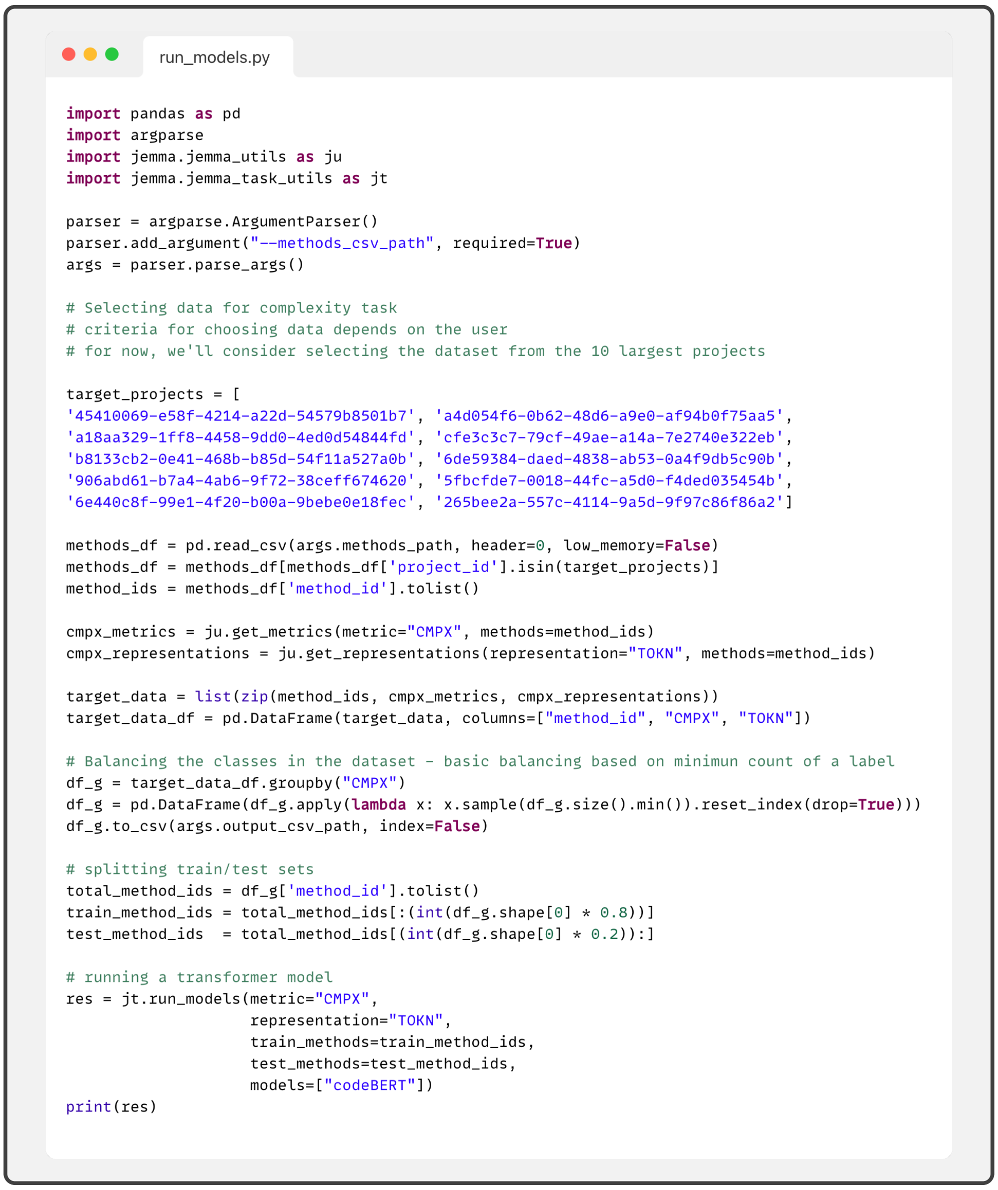
```python
import os, csv, argparse
import pandas as pd
import jemma.jemma_utils as ju
import jemma.jemma_contrib as jc
from subprocess import Popen, PIPE

def run_cmd(cmd):
    process  = Popen(cmd, shell=True, stdout=PIPE, stderr=PIPE)
    res, err = process.communicate()
    return res

parser = argparse.ArgumentParser()
parser.add_argument("--metrixplusplus_path", required=True)
parser.add_argument("--output_path", required=True)
parser.add_argument("--project_path", required=True)
args = parser.parse_args()

# generating metrics for project
project_name   = ju.get_project_name(project_path=args.project_path)
project_id     = ju.get_project_id(project_name=project_name)
output_dbfile  = os.path.join(args.output_path, project_name + ".db")
output_cvfile  = os.path.join(args.output_path, project_name + ".csv")
output_mtfile  = os.path.join(args.output_path, project_name + "_cmpx.csv")

collect_cmd    = ("python2 " + args.metrixplusplus_path +
                  " collect --db-file=" + output_dbfile +
                  " --std.code.lines.code --std.code.complexity.maxindent" +
                  " --std.code.complexity.cyclomatic -- " + args.project_path)

export_cmd     = ("python2 " + args.metrixplusplus_path +
                  " export --db-file="  + output_dbfile +
                  " > " + output_cvfile + " -- " + args.project_path)

collect_status = run_cmd(collect_cmd)
export_status  = run_cmd(export_cmd)

# linking method_ids
target_data = []
with open(output_cvfile, "r") as rd:
    csv_reader = csv.reader(rd)
    csv_header = next(csv_reader)

    for i, row in enumerate(csv_reader):
        (filepath, regionname, regiontype, md, st, en, cyclomatic, maxindent, nl) = row
        if regiontype == "function":
            class_name = filepath.split("/")[-1][-5]
            class_id   = ju.get_class_id(project_id=project_id, class_name=class_name)
            method_id  = ju.get_method_id(class_id=class_id, method_name=regionname)
            target_data.append([method_id, cyclomatic])

# exporting and adding metrics to the jemma datasets
target_df = pd.DataFrame(target_data, columns=["method_id", "CMPX"])
jc.add_metrics(metric="CMPX",
                metrics=target_df["CMPX"].tolist(),
                methods=target_df["method_id"].tolist())
```

Snippet A.1 Defining and adding new property to `JEMMA`: This snippet shows how to add the output of the *Metrix++* code analysis tool as a new property to the `JEMMA` dataset.

gen_comp_representations.py

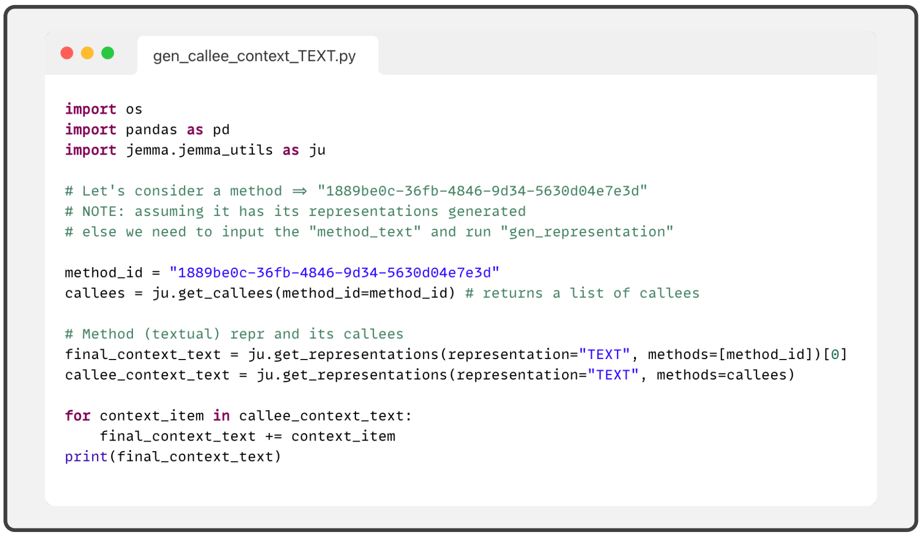
```python
import pandas as pd
import argparse
import jemma.jemma_utils as ju
import jemma.jemma_task_utils as jt

parser = argparse.ArgumentParser()
parser.add_argument("--methods_csv_path", required=True)
args = parser.parse_args()

methods_df = pd.read_csv(args.methods_path, header=0, low_memory=False)
method_ids = methods_df['method_id'].tolist()
metrics = ju.get_metrics(metric="COMP", methods=method_ids)
representations = ju.get_representations(representation="TEXT", methods=method_ids)

for (method_id, comp_metric, repr) in zip(method_ids, metrics, representations):
    pattern = '.' + comp_metric + '('
    mask = '<MASK>'

    repr = []
    for line in repr.split('\n'):
        if pattern in line:
            line = line.replace(comp_metric, mask)
            repr.append(line)
            break # we mask just the first call encountered
        repr.append(line)

    comp_repr = '\n'.join(repr)
    tokn_repr = jt.gen_representation(representation="TKNB",
                                      method_id=method_id,
                                      custom_method_text=comp_repr) # tokens

    c2vc_repr = jt.gen_representation(representation="C2VC",
                                      method_id=method_id,
                                      custom_method_text=comp_repr) # code2vec

    c2sq_repr = jt.gen_representation(representation="C2SQ",
                                      method_id=method_id,
                                      custom_method_text=comp_repr) # code2seq

    ftgr_repr = jt.gen_representation(representation="FTGR",
                                      method_id=method_id,
                                      custom_method_text=comp_repr) # ft graph
```

Snippet A.2 Generating new representations for a masking task: This example shows how to generate new representations for a masking task.



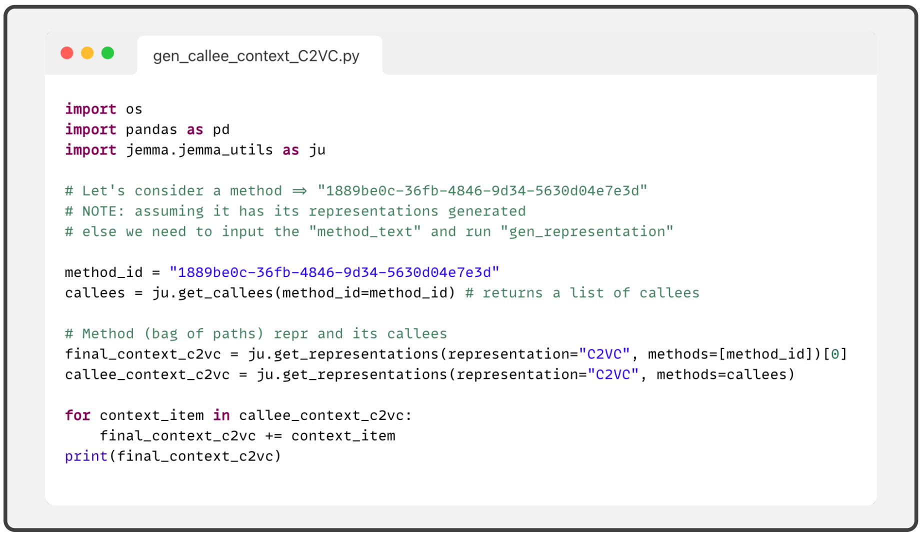
```python
import pandas as pd
import argparse
import jemma.jemma_utils as ju
import jemma.jemma_task_utils as jt

parser = argparse.ArgumentParser()
parser.add_argument("--methods_csv_path", required=True)
args = parser.parse_args()

# Selecting data for complexity task
# criteria for choosing data depends on the user
# for now, we'll consider selecting the dataset from the 10 largest projects

target_projects = [
'45410069-e58f-4214-a22d-54579b8501b7', 'a4d054f6-0b62-48d6-a9e0-af94b0f75aa5',
'a18aa329-1ff8-4458-9dd0-4ed0d54844fd', 'cfe3c3c7-79cf-49ae-a14a-7e2740e322eb',
'b8133cb2-0e41-468b-b85d-54f11a527a0b', '6de59384-daed-4838-ab53-0a4f9db5c90b',
'906abd61-b7a4-4ab6-9f72-38ceff674620', '5fbcfde7-0018-44fc-a5d0-f4ded035454b',
'6e440c8f-99e1-4f20-b00a-9bebe0e18fec', '265bee2a-557c-4114-9a5d-9f97c86f86a2']

methods_df = pd.read_csv(args.methods_path, header=0, low_memory=False)
methods_df = methods_df[methods_df['project_id'].isin(target_projects)]
method_ids = methods_df['method_id'].tolist()

cmpx_metrics = ju.get_metrics(metric="CMPX", methods=method_ids)
cmpx_representations = ju.get_representations(representation="TOKN", methods=method_ids)

target_data = list(zip(method_ids, cmpx_metrics, cmpx_representations))
target_data_df = pd.DataFrame(target_data, columns=["method_id", "CMPX", "TOKN"])

# Balancing the classes in the dataset - basic balancing based on minimun count of a label
df_g = target_data_df.groupby("CMPX")
df_g = pd.DataFrame(df_g.apply(lambda x: x.sample(df_g.size().min()).reset_index(drop=True)))
df_g.to_csv(args.output_csv_path, index=False)

# splitting train/test sets
total_method_ids = df_g['method_id'].tolist()
train_method_ids = total_method_ids[:(int(df_g.shape[0] * 0.8))]
test_method_ids  = total_method_ids[(int(df_g.shape[0] * 0.2)):]

# running a transformer model
res = jt.run_models(metric="CMPX",
                    representation="TOKN",
                    train_methods=train_method_ids,
                    test_methods=test_method_ids,
                    models=["codeBERT"])
print(res)
```

Snippet A.3 Running a Transformer model: Evaluating a transformer model on a prediction task, specifically the cyclomatic complexity prediction task.



```python
import os
import pandas as pd
import jemma.jemma_utils as ju

# Let's consider a method ⇒ "1889be0c-36fb-4846-9d34-5630d04e7e3d"
# NOTE: assuming it has its representations generated
# else we need to input the "method_text" and run "gen_representation"

method_id = "1889be0c-36fb-4846-9d34-5630d04e7e3d"
callees = ju.get_callees(method_id=method_id) # returns a list of callees

# Method (textual) repr and its callees
final_context_text = ju.get_representations(representation="TEXT", methods=[method_id])[0]
callee_context_text = ju.get_representations(representation="TEXT", methods=callees)

for context_item in callee_context_text:
    final_context_text += context_item
print(final_context_text)
```

Snippet A.4 (a) Building a Context: This example shows how to combine a textual representation of a method with additional context from its direct callees.



```python
import os
import pandas as pd
import jemma.jemma_utils as ju

# Let's consider a method ⇒ "1889be0c-36fb-4846-9d34-5630d04e7e3d"
# NOTE: assuming it has its representations generated
# else we need to input the "method_text" and run "gen_representation"

method_id = "1889be0c-36fb-4846-9d34-5630d04e7e3d"
callees = ju.get_callees(method_id=method_id) # returns a list of callees

# Method (bag of paths) repr and its callees
final_context_c2vc = ju.get_representations(representation="C2VC", methods=[method_id])[0]
callee_context_c2vc = ju.get_representations(representation="C2VC", methods=callees)

for context_item in callee_context_c2vc:
    final_context_c2vc += context_item
print(final_context_c2vc)
```

Snippet A.4 (b) Building a Context: This example shows how to combine a code2vec representation of a method with additional context from its direct callees.