

A Survey of Learning-based Method Name Prediction

HANWEI QIAN, State Key Laboratory for Novel Software Technology, Nanjing University, China and Engineering Research Center of Electronic Data Forensics Analysis, China

WEI LIU, City University of Macau, China

TINGTING XU, State Key Laboratory for Novel Software Technology, Nanjing University, China

JIE YIN, Engineering Research Center of Electronic Data Forensics Analysis, China

XIA FENG, City University of Macau, China

WEISONG SUN*, College of Computing and Data Science, Nanyang Technological University, Singapore

ZIQI DING, Engineering Research Center of Electronic Data Forensics Analysis, China

YUCHEN CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China

YUN MIAO, State Key Laboratory for Novel Software Technology, Nanjing University, China

JIAXUN LI, School of Mathematical Sciences, Soochow University, China

JIANHUA ZHAO, State Key Laboratory for Novel Software Technology, Nanjing University, China

CHUNRONG FANG, State Key Laboratory for Novel Software Technology, Nanjing University, China

The choice of method names significantly influences code comprehension and maintenance, posing a considerable challenge, especially for novice developers. Automating the prediction of appropriate method names based on the method code body has emerged as a promising approach to address this challenge. In recent years, numerous machine/deep learning-based method name prediction (MNP) techniques have been proposed. However, a comprehensive and systematic overview of these techniques is currently lacking, hindering future researchers from understanding the research status, development trends, challenges, and opportunities in this field. To fill this gap, in this paper, we conduct a systematic literature review on learning-based MNP studies. Specifically, we first perform a thorough review of the literature concerning publication venue, publication year, and contribution types. This analysis enables us to discern trends in studies related to MNP. Second, we

*Weisong Sun is the corresponding author.

Authors' addresses: **Hanwei Qian**, qianhanwei@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093 and Engineering Research Center of Electronic Data Forensics Analysis, Nanjing, Jiangsu, China, 210031; **Wei Liu**, weiliu980705@gmail.com, City University of Macau, Macau, China, 999078; **Tingting Xu**, xutt.924@gmail.com, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; **Jie Yin**, 402485838@qq.com, Engineering Research Center of Electronic Data Forensics Analysis, Nanjing, Jiangsu, China, 210031; **Xia Feng**, xiafeng@cityu.edu.mo, City University of Macau, Macau, China, 999078; **Weisong Sun**, weisong.sun@ntu.edu.sg, College of Computing and Data Science, Nanyang Technological University, Singapore, 50 Nanyang Avenue, Singapore, 639798; **Ziqi Ding**, antstardzq@gmail.com, Engineering Research Center of Electronic Data Forensics Analysis, Nanjing, Jiangsu, China, 210031; **Yuchen Chen**, yuc.chen@outlook.com, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; **Yun Miao**, miaoyun001my@gmail.com, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; **Jiaxun Li**, 20224207007@stu.suda.edu.cn, School of Mathematical Sciences, Soochow University, Soochow, Jiangsu, China, 215000; **Jianhua Zhao**, zhaojh@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; **Chunrong Fang**, fangchunrong@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Association for Computing Machinery.

1049-331X/2024/0-ART1 \$15.00

<https://doi.org/>

depict the general workflow of learning-based MNP techniques, which involves three consecutive subprocesses: context extraction, context preprocessing, and context-based prediction. Subsequently, we investigate contemporary techniques/solutions applied in the three subprocesses. Third, we scrutinize the widely used experimental databases, evaluation metrics, and replication packages utilized in MNP studies. Moreover, we summarize existing empirical studies on MNP to facilitate a quick understanding of their focus and findings for subsequent researchers. Finally, based on a systematic review and summary of existing work, we outline several open challenges and opportunities in MNP that remain to be addressed in future work.

CCS Concepts: • **Software and its engineering** → **Automatic programming**.

Additional Key Words and Phrases: Method Name Prediction, Method Name Suggestion, Method Name Recommendation, Machine Learning, Deep Learning, AI and Software Engineering

ACM Reference Format:

Hanwei Qian, Wei Liu, Tingting Xu, Jie Yin, Xia Feng, Weisong Sun, Ziqi Ding, Yuchen Chen, Yun Miao, Jiaxun Li, Jianhua Zhao, and Chunrong Fang, 2024. A Survey of Learning-based Method Name Prediction. *ACM Trans. Softw. Eng. Methodol.* 0, 0, Article 1 (2024), 64 pages.

1 INTRODUCTION

In numerous programming languages, such as Java, Python, and C++, methods (or called functions) are essential building blocks of programming languages, allowing developers to encapsulate logic, promote code reuse, and enhance the organization and readability of their code. Usually, each method has a method name that serves as an identifier for specific functionalities or operations within the code. Meaningful and succinct method names are significant in software development, as they can help developers quickly grasp the key functionality of the methods [90]. What's more, choosing good names can be especially critical for methods that are part of public APIs, as poor method names can doom a project to irrelevance [69].

However, constructing high-quality method names is often challenging, especially for inexperienced/novice developers [41]. On the one hand, novice developers may grapple with the intricacies of code abstraction and struggle to encapsulate the essence of a method's functionality in a concise and meaningful manner. On the other hand, they might also face difficulties in adhering to established naming conventions and patterns, which are crucial for maintaining consistency across a codebase. As a result, novice developers might unintentionally introduce ambiguity or inconsistency in method names, complicating collaboration within a development team. The art of naming methods involves not only the technical aspect of accurately representing the code's behavior but also considering the broader context of the application and its potential future modifications [4]. Method name prediction (MNP), also known as method name suggestion, is the task of suggesting appropriate names for methods based on the context of the method code [127]. The method code is also called target method [12]. By suggesting well-crafted, meaningful, and contextually relevant method names, MNP techniques guide best practices, helping developers learn effective naming conventions over time. MNP research focuses on developing advanced techniques for automatically generating method names for method code bodies. Given a target method, MNP techniques can automatically generate an accurate method name for it.

To fulfill the MNP task, numerous researchers have sequentially proposed a plethora of techniques. The origins of MNP research can be traced back to the work of Allamanis et al. [9], who trained an n-gram language model called giga-token on a GitHub Java corpus. During their analysis using the giga-token model, they discovered that method names are more predictable compared to type and variable names. Subsequently, the success of deep learning (DL) technology on many software engineering (SE) tasks [34, 43], a large number of researchers have widely adopted this technology to better solve MNP tasks [10, 31, 59, 127]. DL-based MNP techniques extensively employ DL-based encoder-decoder networks to train models capable of predicting names based on the method's

code body. The encoder is responsible for transforming the method code into a context vector (also known as an embedding), while the decoder uses this context vector to generate the predicted method names. Whether it is the early machine learning (ML)-based MNP technique or the recent DL-based MNP technique, we collectively call it the learning-based MNP technique. Despite the existence of numerous learning-based MNP studies, to the best of our knowledge, there is still a lack of systematic review of research on learning-based MNP. This hinders subsequent researchers from quickly understanding this field's progress, challenges, and research opportunities. Since the introduction of MNP, all related research has employed learning-based methods, and this trend is expected to continue in the future. Therefore, this paper focuses on learning-based MNP research.

We summarize and present the publication venue, public year, and contribution types (including new techniques and empirical studies). Then, based on an in-depth reading and review of these works, we outline the general workflow of learning-based MNP, which completes MNP through three core and consecutive subprocesses: context extraction, context preprocessing, and context-based prediction. We further investigate and summarize the optimization techniques/solutions proposed in existing MNP papers to improve the three subprocesses. In addition, to facilitate a quick evaluation of newly proposed MNP techniques by subsequent researchers, we systematically review and present the evaluation methods employed in existing studies, including experimental datasets, performance metrics, and replication packages. Likewise, we review and summarize existing empirical studies on MNP to help subsequent researchers understand their concerns and findings. Finally, drawing upon a comprehensive examination of the existing MNP literature, we delineate challenges and opportunities for future study. We envision that our findings will play a crucial role in guiding future research inquiries and advancements in this rapidly evolving field.

The main contributions of our survey include:

- **Survey Methodology.** We adopt a systematic approach to collect relevant papers documented in peer-reviewed journals and conferences up to June 1, 2024, as a starting point for future MNP research. We review 51 MNP studies that used ML/DL techniques in terms of publication trends, publication venues, and publication years.
- **Learning-based MNP.** We depict the general workflow of learning-based MNP. This workflow summarizes that learning-based MNP techniques typically achieve the generation of the predicted method names through the following three consecutive subprocesses: context extraction, context preprocessing, and context-based prediction. We further systematically examine and discuss 51 learning-based MNP techniques, employing a multidimensional analysis framework to investigate innovations across the three subprocesses.
- **Dataset, Metric, Replication Packages.** We compile commonly used 18 experimental datasets, 8 performance metrics, and 36 replication packages in MNP research.
- **Empirical studies.** We review and summarize existing empirical studies on MNP to help future researchers understand the concerns and findings of pioneering researchers.
- **Outlook and challenges.** We underscore challenges and opportunities in the learning-based MNP, drawing from our findings with the aim of fostering continued exploration in this field. Supplementary resources, datasets, and code related to our research are available at <https://github.com/software-theorem/MNPSurvey>.

Structure of the paper: The subsequent sections of this paper are structured as follows. Section 2 introduces the background information concerning the ML and DL techniques adopted in the MNP task. In Section 3, we present the survey methodology about how we collect relevant papers from several databases. Sections 4 – 6 provide a comprehensive summary of the primary research questions addressed within this study, along with corresponding findings. Section 7 thoroughly examines the challenges and prospects that lie on the horizon for the field of MNP studies, shedding

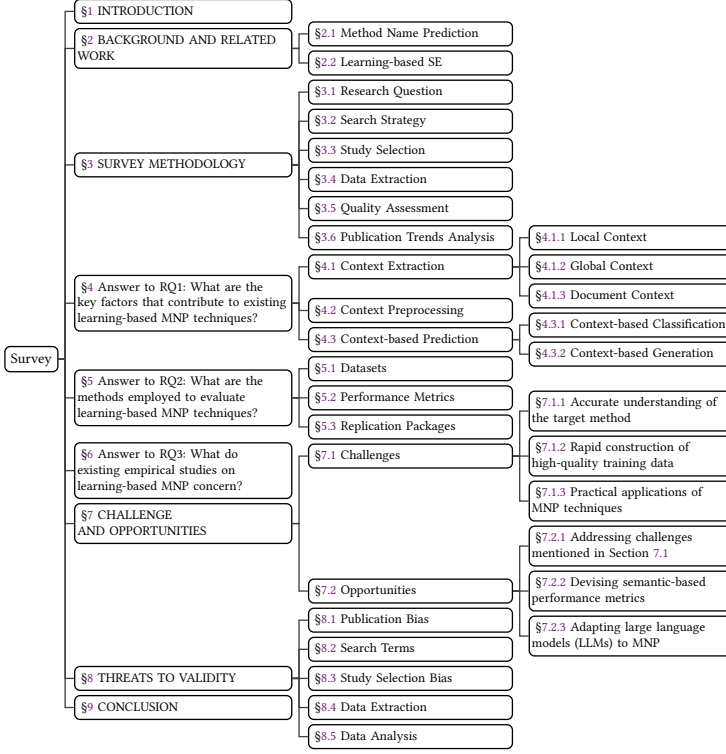


Fig. 1. Structure of the paper

light on potential opportunities for future research. The threats to validity of our survey are discussed in Section 8. Finally, Section 9 provides a conclusion of this survey. The detailed structural information of this paper is presented in Fig. 1.

2 BACKGROUND AND RELATED WORK

2.1 MNP

MNP aims to recommend a proper method name from a source code snippet or stripped binary code according to the extract method’s local context (e.g., method body) and global context (e.g., caller) feature. Fig. 2 illustrates an example. The target method code snippet c_1 presented in Fig. 2(a) is provided by the developer. “calculateArraySum” in Fig. 2(b) is a suitable method name that fulfills the developer’s requirement.

The tokens composing the identifier names in the contexts appear together regularly and naturally due to the developers’ intention to realize the method’s functionality. Specifically, developers tend to combine related identifier name tokens when writing code to achieve specific method functionalities, such as in the method `getHostAddress` used for retrieving the host address, which in code implementation might be a process to *get* the *local* Ethernet listening *address*. An abstract, concise method name captures this functionality. Therefore, the occurrence of tokens in identifier names within the context may influence the tokens in method names. Nguyen et al. [47] collect 2,127,355 files from GitHub, including 17,012,754 methods. Their empirical study shows that for a method, 65.0% of the tokens in its name also exist in the tokens of identifier names in its context.

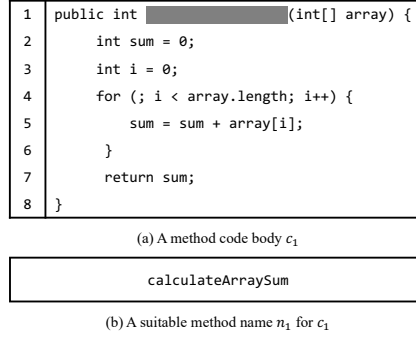


Fig. 2. Example of MNP

As a result, statistical models can be established using machine learning, deep learning, and similar methods to explore the relationship between method names and their context (including the method body). From a machine learning perspective, the primary objective of these models is to learn the latent relationship between method context and method names, formalized as $P(L_{name}|C_{context})$, where $C_{context}$ represents the context of the method, and L_{name} name corresponds to the target name of the method. These models derive a probability distribution model P from training data. Once trained, the models can predict the most probable method name given a specific method body context. MNP can be regarded as a generation task. Its objective is to generate an appropriate method name as output based on a given code snippet or context. In practice, MNP typically uses a fixed vocabulary as the target space for token prediction, thereby categorizing it as a form of classification task. It is generally believed that the closer the tokens predicted by a model are to the tokens of the actual method name, the higher the precision and recall, and the more valuable the predicted results. Researchers have developed various new techniques to enhance the MNP performance, such as extracting more static and dynamic information from code.

2.2 Learning-based SE

Due to the naturalness of software [39], driven by the accumulation of vast amounts of code in open-source repositories, the SE community leverages ML techniques to identify intriguing patterns and unique relationships within code data. This aims to automate or enhance many SE tasks typically performed by developers. Through meticulous feature engineering, researchers aim to identify significant data attributes capable of solving specific problems or automating particular tasks. As computational power advances and modern computer architectures increase available memory, ML methods have evolved to incorporate DL approaches. Enabled by vast repositories of available software code and rapid hardware advancements, DL techniques have made significant strides in various tasks within SE research. Many SE problems can naturally be framed as learning-based data analysis tasks, encompassing classification tasks that aim to categorize data instances into predefined classes, ranking tasks aimed at generating a ranking within a dataset, regression tasks used to assign real values to data instances, and generation tasks that aim to produce concise natural language descriptions. Bug detection [38, 44, 141] can be seen as a classification task for predicting whether a code segment contains bugs. Code search [99, 124] and bug localization [65, 66] in SE are ranking tasks. SE researchers also employ regression models to estimate the effort required for developing software systems [1, 56] and bug fixed times [24, 36], among others, which fall under time-series forecasting problems. Code summarization [43, 54, 100] is typically considered a

generation task, aiming to extract crucial information from code and generate advanced natural language descriptions to aid programmers in understanding the functionality of the code.

Many studies have observed the trend of applying learning-based methods in SE, summarized and synthesized existing research, and conducted an in-depth analysis of future developments [23, 71, 91, 116, 121, 130, 140]. For example, Wang et al. [116] review the literature on ML/DL in SE from 2009 to 2023, analyzing trends, complexity, and reproducibility challenges. It examines differences in ML/DL approaches to SE tasks and identifies key factors influencing model selection. Surveys on learning-based SE typically cover a wide range of topics but may not delve deeply into the specifics of each task. Surveys focused on specific SE tasks, on the other hand, provide more in-depth analysis and insights, serving as a valuable complement. Sun et al. [101] surveys code search techniques, focusing on query-end, code-end, which predominantly uses DL-based code representations, and match-end optimization. They provide a detailed analysis of each dimension and highlight open challenges and future research opportunities in code search. Zhang et al. [137] surveys learning-based automated program repair (APR) techniques, highlighting their use of DL to fix bugs through neural machine translation. It reviews the APR workflow, key components, datasets, and evaluation metrics, and discusses practical guidelines and open issues for future research and application. Omri et al. [73] review various software fault prediction approaches, highlighting how recent DL algorithms address the limitations of traditional feature-based methods by capturing semantic and structural information for more accurate predictions.

The emerging application of DL in SE shows significant potential for executing various traditional software development tasks. The current and future trend is to leverage learning-based methods to address SE challenges. Traditionally, MNP has relied on developers' experience for manual naming, but the advent of learning-based methods has made it possible to automate the suggestion and verification of method names. This paper focuses on the research of learning-based MNP tasks, examining the technologies, methods, models, datasets, and metrics used in various studies. It explores influential techniques and methods, analyzes the current challenges in the field, and discusses potential future trends and directions.

3 SURVEY METHODOLOGY

For a systematic review of MNP research, we follow the guidelines provided by Kitchenham and Charters [97] and Petersen et al. [80] to design the review protocol, which includes research questions, search strategies, study selection, data extraction, and quality assessment. Twelve researchers are involved in this study, all with experience in SE, with most having research experience in MNP. Seven graduate students collaborate throughout the study, responsible for paper collection, data extraction, and analysis, all under the guidance and supervision of experts in SE research. Additionally, five senior researchers contribute significantly by formulating and discussing research questions, addressing biases, and improving documentation for this study.

3.1 Research Question

We aim to summarize, categorize, and analyze empirical evidence from various published studies on learning-based MNP. To achieve this, we address three research questions (RQs):

- **RQ1.** *What are the key factors that contribute to existing learning-based MNP techniques?* This RQ aims to depict the general workflow of learning-based MNP and explore the key optimization techniques proposed in learning-based MNP studies.
- **RQ2.** *What are the methods employed to evaluate learning-based MNP techniques?* This RQ covers three fundamental areas of technique evaluation: databases, performance evaluation metrics, and replication packages.

- **RQ3.** *What do existing empirical studies on learning-based MNP concern?* This RQ discusses existing empirical studies that focus on evaluating learning-based MNP techniques.

By analyzing these RQs, we are able to find challenges and opportunities for learning-based MNP research. These findings are the foundation for creating a research roadmap for future endeavors.

3.2 Search Strategy

During the manual search phase, we select papers published since June 2020 from widely recognized and highly reputable venues in the SE field, such as the International Conference on Software Engineering (ICSE) and ACM Transactions on Software Engineering and Methodology (TOSEM). We meticulously review the initially collected papers' titles, abstracts, and keywords to identify additional keywords and phrases. Subsequently, through brainstorming sessions, we expand and refine the list of search strings, incorporating relevant terms, synonyms, and variations. This iterative process enables us to continually improve the search keyword list based on search outcomes, ensuring accurate capture of literature pertinent to MNP.

The final keywords used for our automated search include ("**method**" OR "**function**") AND ("**name**") AND ("**suggestion**" OR "**recommendation**" OR "**prediction**"). Subsequently, we conduct a total of $2 \times 1 \times 3 \times 4 = 24$ searches across four popular electronic databases in SE: ACM Digital Library, IEEE Xplore Digital Library, Scopus Digital Library, and Google Scholar. Following this, the first two authors manually inspect each paper to determine its relevance to our research scope.

The search was conducted on June 1, 2024, encompassing studies published until that date. After completing automated quests on four electronic databases, 185 relevant studies are obtained, as shown in Table 1. Following removing duplicate findings, 109 studies on MNP are retained. Fig. 3 illustrates the systematic methodology employed for collecting papers in the survey. In the initial stage of the process, the papers are categorized into 6 groups based on several criteria, including method name suggestion, function name suggestion, and other related categories. A manual review of the initial 500 records for each group is conducted to ensure the inclusion of relevant papers for further analysis. Subsequently, the papers are sourced from four major digital libraries. The following digital libraries are consulted: the ACM Digital Library (42 papers), the IEEE Xplore Digital Library (8 papers), the Scopus Digital Library (25 papers), and Google Scholar (110 papers). An automated search identifies 185 papers, which are then filtered to remove duplicates, yielding 169 papers. The selection process continues with the application of filters to the titles and abstracts of the papers, resulting in 109 papers. These are then subjected to a detailed full-text review, which narrows the number down to 32 papers. The snowballing technique is employed to identify additional relevant papers through the references of the selected papers, expanding the pool to 1,660 papers. During this phase, citations that are not initially identified are incorporated. A comprehensive and meticulous discussion and selection process ultimately result in the inclusion of 51 papers that meet all predefined criteria.

3.3 Study Selection

The preliminary selection process of the study involved summarizing all research published in SE related to MNP. We select the top 1,000 search results from each database and screen suitable studies by reviewing each study's title, keywords, and abstract. Subsequently, these studies underwent further screening based on detailed inclusion and exclusion criteria outlined in Table 2. Inclusion criterion I4 about minimum quality threshold will be discussed in detail in Section 3.5. Each study was evaluated independently by at least two reviewers. In cases of disagreement, senior researchers facilitated discussions to achieve consensus.

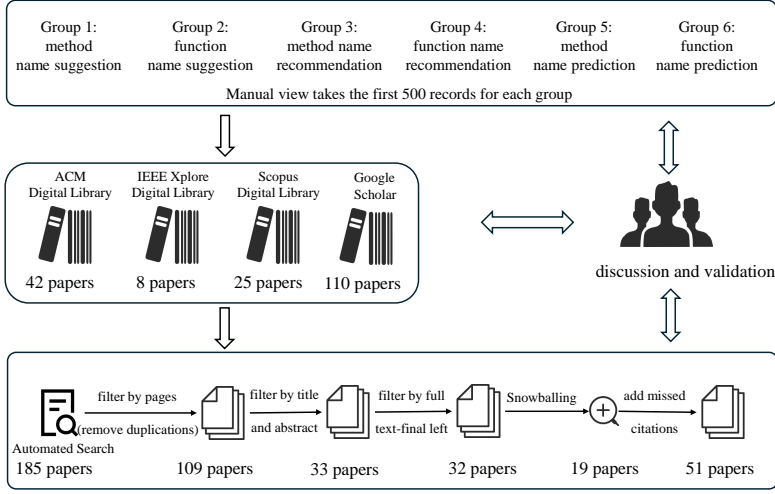


Fig. 3. General workflow of the paper collection

Table 1. The results of the retrieval and screening of papers related to MNP

Process	Studies
ACM Digital Library	42
IEEE Xplore Digital Library	8
Scopus Digital Library	25
Google Scholar	110
Automatic search from four electronic databases.	185
Removing duplicated studies.	109
Excluding primary studies based on title and abstract.	33
Excluding primary studies based on full text - final left	32
Snowballing	19
Add missed citations	51

Table 2. The inclusion and exclusion criteria

ID	Inclusion criteria
I1	The paper must be written in English.
I2	The paper must involve at least one machine/deep learning technique.
I3	The paper must be a peer-reviewed published full paper.
I4	The paper must pass the minimum quality threshold.
ID	Exclusion criteria
E1	The paper is a form of grey literature, e.g., a technical report or dissertation.
E2	The paper is explicitly a short paper, position paper, or editorial.
E3	After being extended into journal versions, conference papers are excluded.

To ensure comprehensive coverage of the field in our investigation, we employ a snowballing approach to gather further papers that our keyword searches might have missed. The snowballing method identifies citation-dependent papers to expand our paper collection. We utilize both

Table 3. Data extraction form

Item	Description	Association
Title	The title of the literature.	Publication Trend
Publication year	The publication year of the literature.	Publication Trend
Publication type	The type of the literature, e.g., new technique and empirical study.	Publication Trend
Prediction approach	Methods/techniques used for context extraction, context preprocessing, context encoding, context classifying/decoding	RQ1
Datasets	Focus on the dataset's type, language, scale, and source	RQ2
Performance metrics	Measurement for evaluating a technique.	RQ2
Replication package	Focus on availability, dependencies, and documentation.	RQ2
Programming languages	The programming languages used for performing MNP tasks	RQ3
Research questions	The key inquiries or investigation objectives posed in research	RQ3

backward and forward snowballing techniques. In the backward snowballing, we scrutinize the references of each collected paper and identify papers within our scope. In the forward snowballing, we use Google Scholar to find papers of interest that cited already collected papers. We iteratively repeat the snowballing process until no new relevant papers are added, reaching a stable state. Through this process, we retrieve an additional 19 papers, further enriching the coverage of the MNP literature.

Furthermore, to ensure the comprehensiveness and accuracy of our investigation, we also reached out to the authors of the collected papers. We provided them with our paper and requested them to review whether our depiction of their work was accurate. This interaction allowed us to understand their contributions better and make necessary revisions to our descriptions. Additionally, through these communications, authors directed our attention to additional papers, initially not included in our collection. Among the suggested papers, 32 met our inclusion criteria and were deemed relevant to our investigation. These 19 papers were subsequently added to our literature repository, further enhancing the coverage of MNP literature in our investigation.

3.4 Data Extraction

The purpose of data extraction is to collect data items that can address the research questions. Our data collection primarily focuses on publication information, study background, technical details, and experimental settings, as detailed in Table 3 of this paper. The data extraction process is divided among the seven authors of this study, with each paper being independently extracted by at least two authors and reviewed by their principal supervisor. Data items relevant to answering the three research questions are meticulously recorded, along with any information that aids in analyzing MNP techniques. After completing data extraction, we compare the extracted data and assess their consistency using Cohen's kappa coefficient [19]. If the two authors achieve a high agreement in extracting methods for predicting methods, their data extraction is considered reliable and consistent. If the coefficient score is low, relevant concepts and definitions are further discussed or adjusted until a consensus is reached. Other senior researchers randomly check the extraction results to mitigate bias.

3.5 Quality assessment

Quality assessment (QA) plays a role in selecting studies and interpreting results in systematic literature reviews [51]. By providing broader inclusion and exclusion criteria, QA helps to screen and identify appropriate studies. These assessment criteria are formulated following one of the most widely used quality assessment tools proposed by Dybå and Dingsøyr [26]. According to their approach, the assessment of research quality primarily focuses on rigor, credibility, and relevance. We develop four quality assessment criteria, detailed in Table 4 as QA1 to QA4.

Table 4. Checklist of Questions to Assess the Quality of Studies

ID	Quality assessment criteria
QA1	Does the predicted method names generated through code snippets?
QA2	Does the paper describe evaluation metrics?
QA3	Does the paper provide analysis after the performance evaluation?
QA4	Is the contribution of the research clearly stated?
QA5	Does the paper provide a publicly accessible link to the code?
QA6	Are all the links provided in the paper available?
QA7	Is the raw dataset retrieved from open source?
QA8	Are all the necessary components or environments for replication provided?
QA9	Does the paper provide sufficient detail to enable replication?
QA10	Does the paper provide all necessary details for reproducibility?

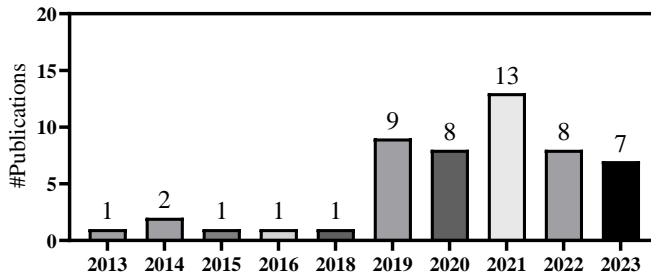


Fig. 4. Number of publications per year

Detailed quality assessment information is crucial for data synthesis and result interpretation. In SE, replicability and reproducibility are essential for determining the quality of original studies and testing their reliability. We design a quality checklist with QA5 to QA10 to generate quality data. This data is then used as evidence to support parts of the answers to RQ1 and RQ2. In Section 5, we provide a more detailed discussion of the answers to the checklist questions and their analysis.

Overall, these ten criteria provide a metric to assess whether the findings of a specific study can contribute valuable insights to the MNP field. Meeting the requirements of criteria QA1, QA2, QA3, and QA4 indicates passing the minimum quality threshold. In the quality assessment process, each study is independently evaluated by two reviewers based on ten criteria and undergoes validation.

3.6 Publication Trends Analysis

We analyze the publication information from the 51 MNP studies retrieved from Section 3 and discuss the main emerging publication trends.

Fig. 4 illustrates the yearly publication count for research papers. Fig. 5 shows the cumulative number of publications for the corresponding year. As displayed in Fig. 4 and 5, it is observed that the first MNP study was published in 2013. The studies on MNP gradually gained popularity after 2019, peaking in 2021. The papers published between 2019 and 2022 represent 88% of the total.

The 51 reviewed papers are published in different conferences and journals. From Fig. 6(a), it is observed that 82% of the papers are published in conference proceedings and 18% in journals. Additionally, Fig. 6(b) shows that out of the 51 articles, 43 belong to the category of new techniques (NT), 8 is an empirical study (ES) [58].

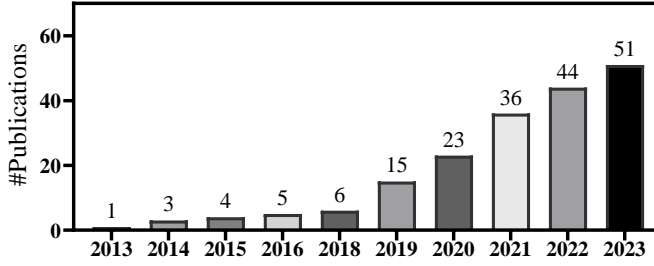


Fig. 5. Cumulative number of publications per year

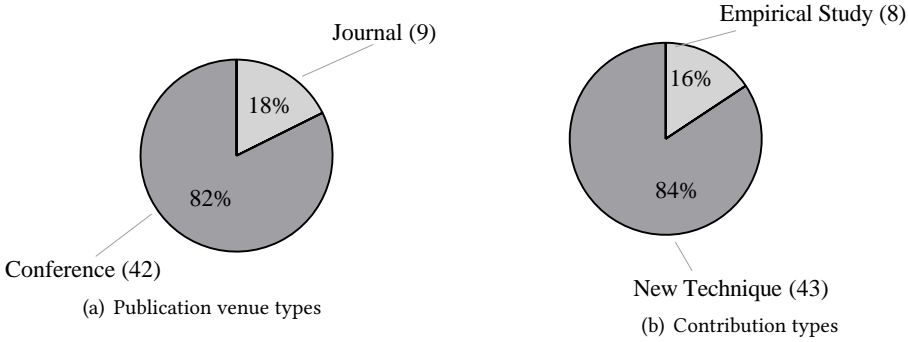


Fig. 6. Publication venues and contribution types

Table 5 lists the detailed publication venues of reviewed studies, covering various types of studies. Among the 30 publication venues, the most popular conferences for publishing these papers are ICSE, ICPC, MSR, PLDI, PACMPL, and ICLR. ICSE has the most significant number of presented papers among these.

Summary ► MNP gained attention in SE research literature in 2015, with its popularity peaking in 2021. Of the 42 papers, which represent 82% of the total, are published in conferences, while the remaining ones are in journals. Additionally, 43 papers, accounting for 84% of the total, focus on new MNP techniques. ◀

4 ANSWER TO RQ1: WHAT ARE THE KEY FACTORS THAT CONTRIBUTE TO EXISTING LEARNING-BASED MNP TECHNIQUES?

In this section, we summarize and present the modeling and auxiliary techniques used by the reviewed learning-based MNP studies. First of all, we outline the general workflow followed by learning-based MNP techniques. Fig. 7 depicts this workflow. It is observed that given a target method (usually a method code body) and the corresponding method context, the learning-based MNP technique generates the predicted method name for it through the following four consecutive subprocesses: context extraction, context preprocessing, context encoding, and context classifying/decoding. Context extraction is responsible for extracting useful information from the target

Table 5. Publication venues for method name prediction studies (NT: New Technique; ES: Empirical Study; TS: Total Studies)

Short Name	Full Name	NT	ES	TS
MSR	International Conference on Mining Software Repositories	4	0	4
ICPC	International Conference on Program Comprehension	5	0	5
PLDI	ACM-SIGPLAN Symposium on Programming Language Design and Implementation	3	0	3
ESEC/FSE	ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering	2	0	2
ICML	International Conference on Machine Learning	1	0	1
PACMPL	Proceedings of the ACM on Programming Languages	3	0	3
ICLR	International Conference on Learning Representations	3	0	3
ISSTA	International Symposium on Software Testing and Analysis	0	1	1
APSEC	Asia-Pacific Software Engineering Conference	1	0	1
PEPM	ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation	1	0	1
ICSE	International Conference on Software Engineering	6	0	6
SANER	IEEE International Conference on Software Analysis, Evolution, and Reengineering	2	0	2
STC	Annual Software Technology Conference	0	1	1
RL+SE&PL	International Workshop on Representation Learning for Software Engineering and Program Languages	0	1	1
QUATIC	Quality of Information and Communications Technology	0	1	1
COLING	International Conference on Computational Linguistics	1	0	1
ASE	International Conference on Automated Software Engineering	1	0	1
JSS	Journal of Systems and Software	1	0	1
PAKDD	Pacific-Asia Conference on Knowledge Discovery and Data Mining	1	0	1
NeurIPS	Neural Information Processing Systems	1	0	1
Inf Softw Technol	Information and Software Technology	0	1	1
AAAI	Association for the Advancement of Artificial Intelligence	1	0	1
ICCQ	International Conference on Code Quality	1	0	1
ISSRE	International Symposium on Software Reliability Engineering	1	0	1
Empir. Softw. Eng.	Empirical Software Engineering	1	0	1
COLA	Journal of Computer Languages	1	0	1
TOSEM	ACM Transactions on Software Engineering and Methodology	1	0	1
SSE	IEEE International Conference on Software Services Engineering	0	1	1
QRS	International Conference on Software Quality, Reliability and Security	1	1	2
IJCSIT	International Journal of Computer Science and Information Technologies	0	1	1

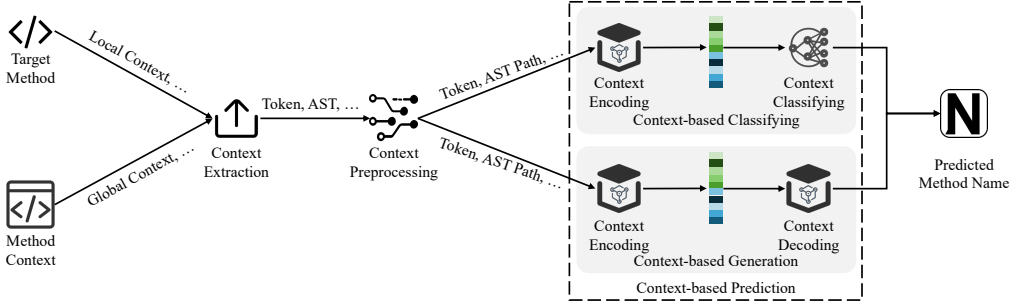


Fig. 7. Workflow of learning-based MNP

method and its contexts, such as Token and Abstract Syntax Tree (AST), which is input to subsequent processes to facilitate producing the predicted method name accurately, detailed in Section 4.1. Context preprocessing is responsible for further processing the information extracted in the context extraction process, aiming to explore rich context features better and enhance the accuracy of MNP. Common preprocessing methods involve converting AST or Graph into AST paths and Graph paths, detailed in Section 4.2. Context-based prediction is responsible for predicting suitable method names based on context features produced in the context preprocessing process. As shown in Fig. 7, existing context-based prediction techniques can be mainly divided into two categories: context-based classification and context-based generation. As its name implies, context-based

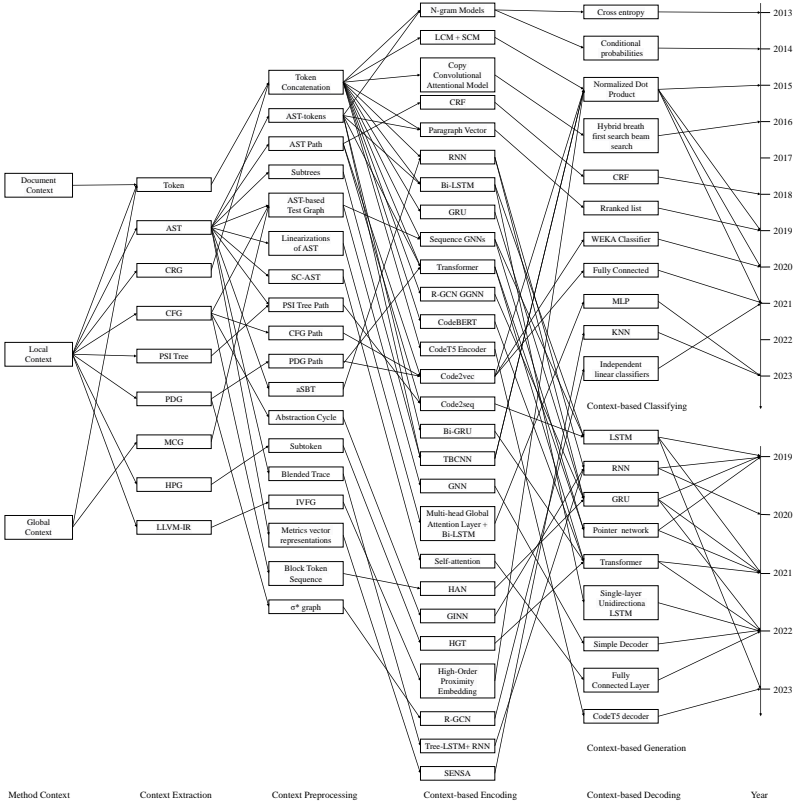


Fig. 8. Components of learning-based MNP

classification techniques train a classifier capable of classifying the input context features into a label that corresponds to a method name (detailed in Section 4.3.1), while context-based generation techniques train a generative model that can generate an appropriate method name based on the input context features (detailed in Section 4.3.2). Fig. 8 provides a detailed view of the entire process in a learning-based MNP task, illustrating all components involved and their interrelationships, from context extraction and preprocessing to context-based prediction.

4.1 Context Extraction

Fig. 9 presents the common useful information extracted by existing learning-based MNP techniques in the context extraction process, including local context, global context, and documentation context.

4.1.1 Local context. The local context is defined as the context information extracted from the target method itself [59, 118]. Local context (also called internal context [57]) can also be further fine-grained into implementation context [69] and interface context [69]. The implementation context illustrates how the method is implemented, including the names of the variables, fields, and methods that are used, accessed, or invoked in the method's body [69]. The interface context specifies the input and output of the method, including the parameters' types and the return type of the method [69]. Such information is used in the form of tokens (denoted as Token) in many learning-based MNP techniques. There are still many MNP studies that do not use the concept

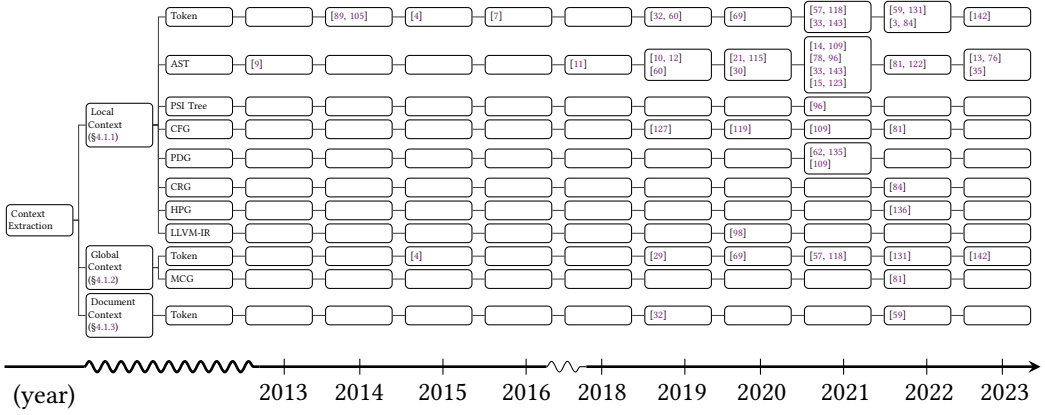


Fig. 9. Evolution of context extraction

of local context, but based on the concept of local context elaborated in the above two works, we can classify the context information used by these MNP studies to predict method names as local context. Such context information includes abstract syntax tree (AST) [10–14, 21, 76, 81, 96, 109, 115], program structure interface (PSI) tree [96], control flow graph (CFG) [64, 81, 109, 119, 127], program dependence graph (PDG) [62, 64, 109, 135], code relation graph (CRG) [84], and heterogeneous program (HPG) [136]. In the following paragraphs, we will introduce these local contexts and corresponding extraction methods.

• Token.

Definition. In learning-based MNP techniques, tokens serve as a common local context representation. Tokens are typically extracted from various elements within the method’s code, such as variable names, field names, method names, parameters, return types, and method calls. The extraction process involves breaking down these elements into individual tokens using conventions like CamelCase and underscores, then normalizing them to lowercase. Tokens can be categorized into different contexts, including implementation context (tokens from the method’s body) and interface context (tokens from parameters and return types). Additionally, some approaches consider a local context window around the token to be predicted, including tokens that appear within a specified range of positions before and after the target token.

Literature. In [69, 118, 131, 142], the authors extract tokens from the names of variables, fields, and methods that are used, accessed or invoked in the method’s body (i.e., implementation context), and tokens from the parameter’s types and the return type of the method (i.e., interface context) [69, 118, 131]. In [4], Allamanis et al. define the *local context* as the set of tokens within K positions before and after the code token t to be predicted. To extract tokens, they collect each method’s name, parameter names and types, return type, class name, and the names of variables, fields, and method calls within the method body. These names are tokenized using CamelCase and underscore naming conventions, and the tokens are normalized to lowercase. In [57], Li et al. extract tokens from the method body, argument types, and names, and the method’s return type. They break names into sub-tokens, which are collected in the order they appear in the source code. The internal context includes sub-tokens from the return type, parameter types, and names. Single-character (sub)tokens are removed. In [59], Liu et al. extract tokens from method names, identifiers, parameters, and return types by parsing the method into an AST. They split entities using camelcase and underscore conventions, lowercase them, and concatenate the sub-tokens in their original order to form a

sequential representation of the local feature. In [84], Qu et al. extract tokens directly from the method body fields and add them sequentially. Compound tokens are decomposed into sub-tokens, which are collected to represent the semantic context of each method. In [3], the authors extract the token sequence from the given body. They broke the code body into subtokens using BPE [93]. In [142], Zhu et al. utilize javalang and spiral to extract the names of program entities from identifier names and method signatures, then divide these names into sequences of sub-tokens.

• Abstract Syntax Tree (AST).

Definition. AST is derived from the parse tree through semantic analysis. A parse tree, also known as a concrete syntax tree, is a graphical representation of the syntactic structure of a program according to the rules of formal grammar. The AST is a simplified representation of the parse tree that focuses on the essential elements of the program's structure. Compared with the parse tree, the AST provides a more concise and structured representation of the program's semantics, making it easier to analyze and manipulate.

The specific definitions and explanations of AST may slightly vary across different sources [10, 13, 21, 81, 96]. For example, Alon et al. [10] claim that an AST uniquely represents a code snippet in a given language and grammar. The leaves of the tree are called terminals and usually refer to user-defined values that represent identifiers and names from the code snippet. The non-leaf nodes are called nonterminals and represent a restricted set of structures in the language, e.g., loops, expressions, and variable declarations. Compton et al. [21] state that an AST is a tree-like representation of a code snippet. It does not contain every detail in the original source code (e.g., parentheses and comments) but rather, represents the important syntactic structures that make up the functionality of the code snippet.

Example. Fig. 11 illustrates an example of a code snippet along with its corresponding AST.

Literature. AST is also a widely used local context in existing learning-based MNP techniques [10–14, 21, 76, 81, 96, 109, 115]. Existing studies [58, 82] indicate that there are significant differences in ASTs generated by different parsers, resulting in notable variations in their impact on MNP. A suitable parser can help researchers entirely use the information in the AST [58]. Existing learning-based MNP techniques have tried a variety of AST parsers. For example, in [11], Alon et al. adopt different AST parsers for code in various programming languages, including JavaParser [40] for Java; UglifyJS for JavaScript; Python internal parser and AST visitor for Python; and Roslyn for C#. In [10, 12, 13, 47, 76, 81, 115, 119], the authors utilize JavaParser to parse each target method into an AST. In [21], Compton et al. utilize Spoon [77] to parse the method into the AST. Spoon is an open-source library for parsing, rewriting, transforming, and transpiling Java code. In [14], Bui et al. build their technique on the top of ASTs produced by srcML [20]. SrcML provides a combined vocabulary of AST node types for five main-stream languages, including Java, C, C++, C#, and Objective C. In [109], Vagavolu et al. parse each code snippet using a platform called Joern [129] to generate AST. In [96], instead of using existing AST parsers, Spirin et al. propose a new approach called PSIMiner to build AST. PSIMiner extracts Program Structure Interface (PSI) trees for the files by calling the IntelliJ Platform's API. To produce an AST, PSIMiner traverses PSI in the depth-first search order. For each node, it checks whether it should belong to the final AST by looking at its type. PSIMiner ignores some predefined types like PsiWhiteSpace because they do not carry any semantic information. Also, it ignores nodes defined in its JSON configuration file, which can be useful, for example, to remove comments from code. Further, PSIMiner converts the nodes to an internal representation that contains the original PSI objects alongside additional user-defined information obtained by calling the PSI API. In [59], the authors use Javalang to extract ASTs. Some MNP studies have investigated the differences in ASTs generated by different AST parsers and the impact on MNP performance. For example, in [58], Li et al. investigate the differences in ASTs generated by five AST parsers: JavaParser, ANTLR, Tree-sitter, Guntree [46], and Javalang. We

Table 6. AST parsers used in existing MNP studies

Parser Year	JavaParser	Spoon	srcML	Joern	PSIMiner	Tree-sitter	ANTLR	Gumtree	Javalang	JDT	Semantic	Astminer	fAST	Total
2013	0	0	0	0	0	0	0	0	0	[9]	0	0	0	1
2018	[11]	0	0	0	0	0	0	0	0	0	0	0	0	1
2019	[12] [10, 47]	0	0	0	0	0	0	[60]	0	0	0	0	0	4
2020	[115, 119] [30, 86]	[21]	0	0	0	0	0	0	0	0	0	0	0	5
2021	[33]	0	[14]	[109]	[96]	[78]	0	0	0	0	[143]	0	[15]	7
2022	[81]	0	0	0	0	0	0	0	[59]	0	0	[122]	0	3
2023	[13, 58] [82]	0	0	0	0	[58, 82]	[58]	[58]	[58, 82] [35]	[82]	0	0	0	4
Total	13	1	1	1	1	3	1	2	4	2	1	1	1	-

count the AST parsers used in MNP studies, and the results are shown in Table 6. It is observed that there are various AST parsers used in learning-based MNP research, among which JavaPaser is used most frequently.

In [78], Peng et al. generate the AST for each method using the Tree-Sitter parser [95], where all code tokens are mapped as terminals. They split sub-tokens following [12], where each code token is split according to code naming conventions, e.g., `setConnectionsPerServer` is split into `{set, connections, per, server}`. In [143], Zügner et al. extract source code from the dataset and use Semantic to obtain the relevant ASTs. In [60], Liu et al. use GumTree to obtain the ASTs [61]. In [9], Allamanis et al. tokenize and retrieve the AST of the code using Eclipse JDT. In [30, 33, 122], they use Javaparser to get the AST of the Java code. In [15], Bui et al. use fAST, an efficient parser [133], to parse code into ASTs in a binary format equivalent to SrcML [20]. In [35], Guo et al. obtain two versions of the method (before and after modification) and use the Javalang tool [106] to parse these methods to extract method names. If parsing errors occur, they apply regular expression-based rules to collect the method names.

• PSI Tree.

Definition. PSI [96] is a combination of the syntax tree structure of code and the interfaces to interact with its parts. PSI trees act as concrete syntax trees: apart from the AST structure they also carry information about punctuation marks, braces, and other formatting. A PSI tree contains all AST nodes but adds nodes introducing the internal structure, nodes for modifiers (even if they are not present), and other Java syntactic constructs.

Example. Fig. 10 illustrates the PSI Tree of the code snippet. The red and green nodes correspond to the Abstract Syntax Tree (AST), while the blue nodes are generated by the PSI.

Literature. Spirin et al. [96] propose a tool PSIMiner that can transform an AST into a Program Structure Interface (PSI) tree. They utilize the PST tree to enhance ASTs with identifier types to improve one of the existing MNP model code2seq [10].

• Control Flow Graph (CFG).

Definition. A control flow graph (CFG) is a directed graph where the nodes represent predicates and statements, and the edges connecting them indicate the transfer of control between those nodes. It describes the order in which program statements execute. Each edge also has a label that indicates the necessary conditions to execute that path.

Example. Fig. 11 illustrates an example of a code snippet along with its corresponding CFG.

Literature. Many MNP studies extract CFG from the target method and use it as the local context [64, 81, 109, 119, 127]. Existing MNP research mainly uses three tools to generate the CFG of the target method, including Spoon [119, 127], Joern [109], and Soot [64, 81].

• Program Dependence Graph (PDG).

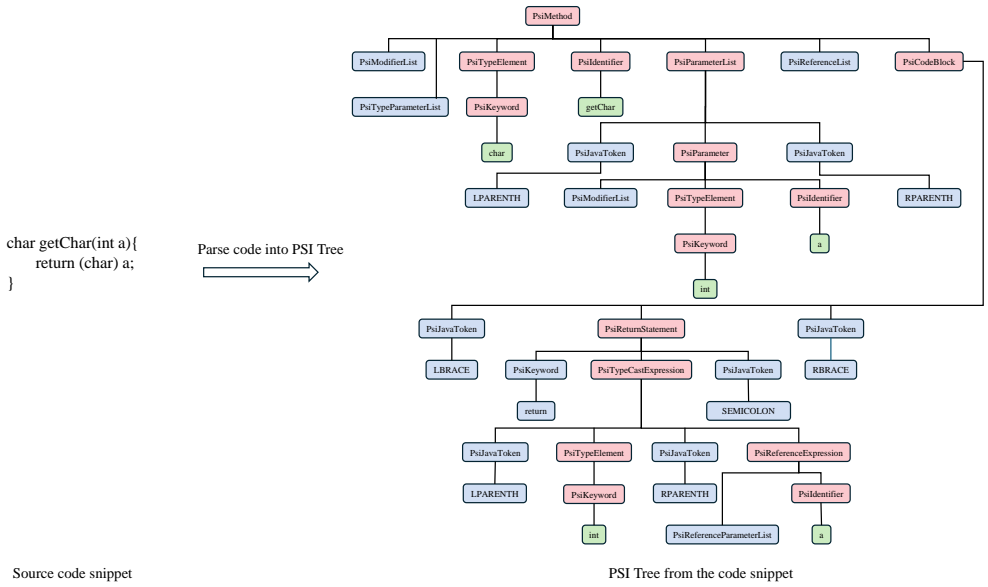


Fig. 10. Example of a code snippet and corresponding PSI Tree provided in [96]

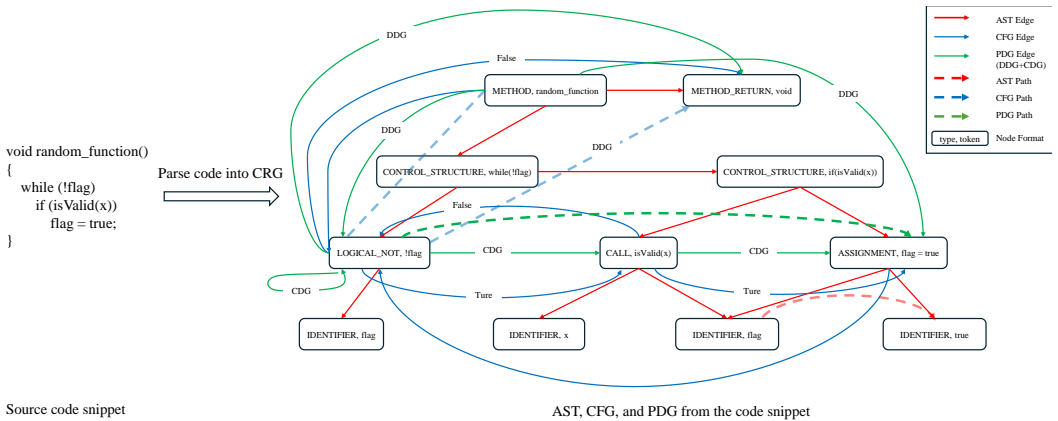


Fig. 11. Example of code snippet and corresponding AST, CFG, and PDG provided in [109]

Definition. A program dependency graph PDG is a directed graph where the nodes represent predicates and statements, and the edges connecting them indicate control dependencies and data dependencies between those nodes [109].

Example. Fig. 11 illustrates the AST, CFG, and PDG of the given code snippet. Each node in the figure contains two fields separated by a comma. The first field indicates the type of the node, while the second field represents the actual code token associated with the node. The advantages of PDG are of two types: control dependency edges representing how the result of a predicate influences the variable and data dependency edges that indicate how one variable affects another [109].

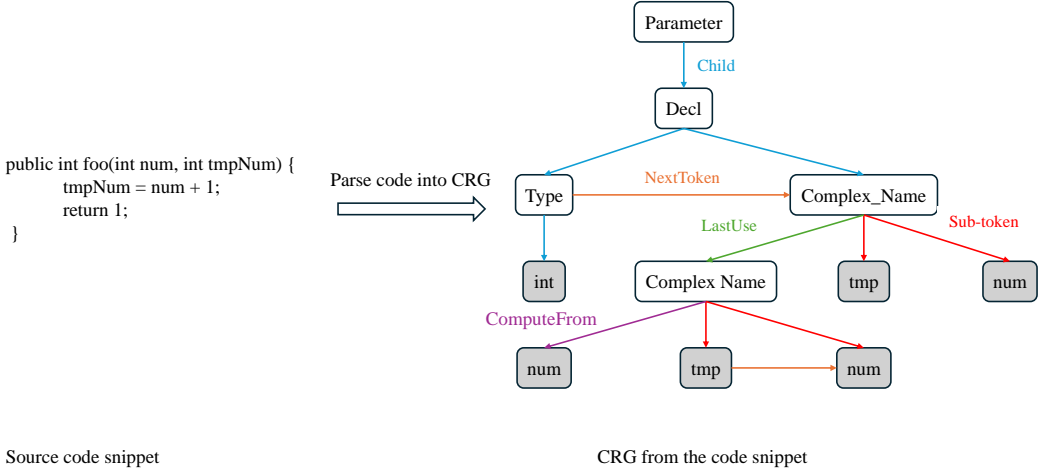


Fig. 12. Example of a code snippet corresponding CRG provided in [84]

Literature. Several studies take PDG into account when devising learning-based MNP techniques [62, 64, 109, 135]. To generate the PDG of the target method, Zhang et al. [135] use Soot [110] to compute the control flows and data flows for each target method at the Jimple statement level, and then construct PDG. Liu et al. [62] utilize Joern to generate PDG. Ma et al. [64] first convert the Java program to a Jimple intermediate representation and then employ Soot to obtain the PDG.

• Code Relation Graph (CRG).

Definition. The CRG retains the structure of the abstract syntax tree (AST) and includes data and control flows.

Example. Fig. 12 presents the CRG of the code snippet. They use the graph output format [6] from the code analysis tool fastast [133] to construct a CRG for each method, representing the structural context.

Literature. In [84], Qu et al. propose a code relation graph (CRG) to describe code structure. The CRG is built as follows: "Child" edges connect nodes according to the AST. Since tokens have no inherent order, "NextToken" edges link each token to its successors. To capture control and data flows, additional edges indicate variable use and updates. For a token, edges connect it to tokens where the variable was last used ("LastUse" edges). When an assignment is observed, the token is connected to all variable tokens in the expression using "ComputeFrom" edges. Return tokens are also connected to the method declaration with "ReturnTo" edges.

• Heterogeneous Program Graphs (HPG).

Definition. HPG is proposed by Zhang et al. [136], which uses type information in the chart to represent code and can explicitly provide the types of nodes and edges. An HPG of a certain code snippet is generated based on its AST according to the abstract syntax description language (ASDL) [114]. HPG can relieve the type missing issue in the homogeneous graph [136].

Example. Fig. 13 shows the HPG of the code snippet. Each node is assigned a type (left part) and a value (right part). Each edge has a type label. HPG is a format of heterogeneous graphs. A

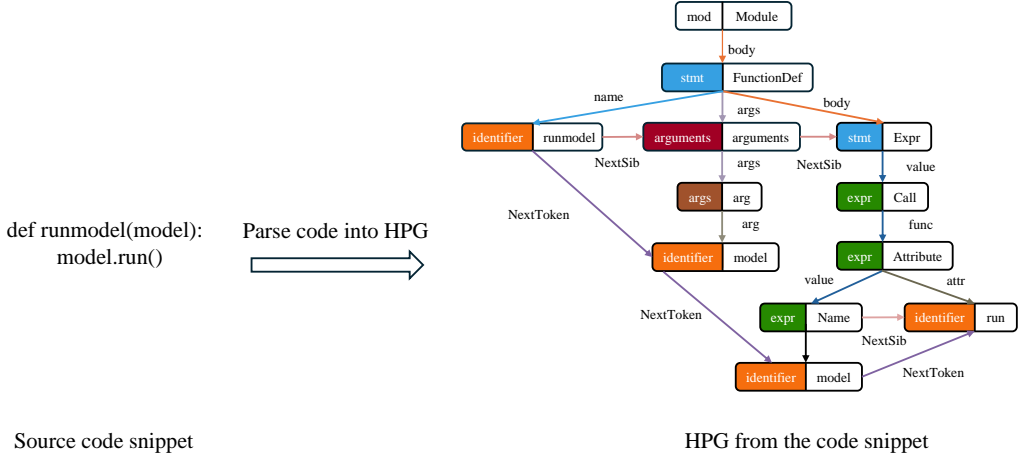


Fig. 13. Example of a code snippet and corresponding HPG provided in [136]

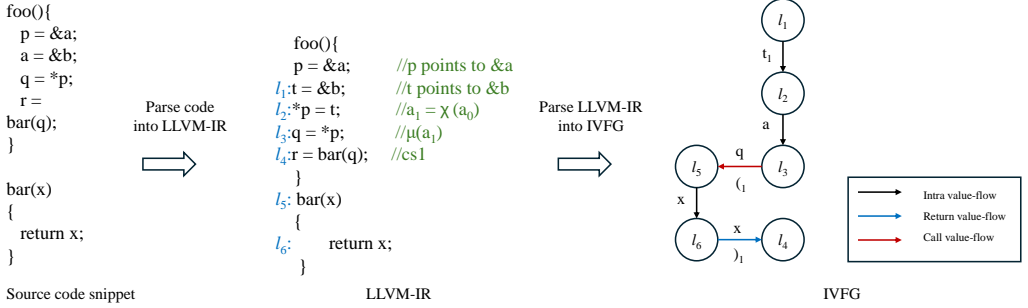


Fig. 14. Example of a code snippet and corresponding LLVM-IR and IVFG

heterogeneous graph is a graph consisting of different types of entities, i.e., nodes, and different types of relations, i.e., edges.

Literature. Zhang et al. verify the promoting effect of HPG on MNP tasks. To generate HPG, they implement HPG parsers for two popular programming languages, including Python and Java. The Python parser conforms to the official Python 3.7 ASDL grammar¹. The implemented Python parser can extract up to 23 types of nodes and 71 types of edges (forward). As for Java, they implement an HPG parser based on the AST parser Tree-sitter. They manually define the field type for Java and assign types to the edges in AST.

• LLVM-IR.

Definition. LLVM-Intermediate Representation (IR) is a uniform code representation generated by modern compilers like LLVM, which supports various programming languages through multiple front-ends. It is widely used for program analysis and transformation tasks, representing a program's code in a structured and language-agnostic manner.

¹<https://docs.python.org/3.7/library/ast.html#abstract-grammar>

Example. Fig. 14 is an example of a code snippet and corresponding LLVM-IR.

Literature. In [98], Sui et al. argue that LLVM-IR expresses the logical structure of a program through specific instruction types, enabling it to accurately capture information about variables, functions, and control flow.

4.1.2 Global context. The global context is defined as the context information extracted from the target method context, such as other methods that have a calling relationship with the target method [69, 118]. The global context includes enclosing context [57, 69, 118, 142], call relation [118], interaction context [57], sibling Context [57], and pattern guider [131]. The enclosing context refers to the enclosing class, including the class name's tokens [69]. Such context provides information on the general task/purpose of the class where the target method is implemented. In [57], the enclosing context for the target method includes the name of the enclosing class of the target method and the names of the program entities, method calls, field accesses, variables, and constants in the class. The definition of call relation is as follows: given two methods a and b , if b is triggered in the implementation context of a , then the call relation $a \rightarrow b$ is established where a is the caller while b is the callee [128]. Interaction context for the target method includes the names of the callers of the target method, the contents of the callers (i.e., bodies, interfaces, return types), the names of the callees, and the contents of the callees (bodies, interfaces, return types) [57]. The sibling context for the target method m includes the names of the methods in the same class with m , and the contents of sibling methods (including the bodies, interfaces, and return types) [57]. The pattern guide of a method is the name of its most similar method [131]. This represents the naming pattern of the method name. Like local context, global context is also used in the form of tokens in many learning-based MNP techniques [4, 57, 69, 118, 131, 142]. Some MNP studies do not use the concept of global context, but based on the concept of global context elaborated in the above endeavors, we can classify the context information used by these studies to predict method names as global context. Such global context information includes aggregated call graph (ACG) [132] and method call graph (MCG) [64, 81]. In the following paragraphs, we will introduce these global contexts and corresponding extraction means.

• Token.

Most global contexts are applied in various learning-based MNP studies in the form of tokens. For example, Nguyen et al. [69] extract enclosing context tokens from the class name.

Literature. In [118], to extract the global contexts, Wang et al. establish call relations via analyzing the names within each MethodInvocation AST node in the project. They respectively extract the entity names from the enclosing context, the interface context of the callees, and the implementation context of the callees, after which these names are broken into tokens based on the camel cases and underscore naming conventions. To restrict the length of the input sequence, they limit the number of tokens extracted from the implementation context of each callee method to ten. In [57], for the sibling context and enclosing context, the authors directly extract the names from the program entities, the return type, and the types in the interface. The names are broken into sub-tokens, which are collected into the sequence in the same appearance order in the source code. For the interaction context, they first build a call graph using Soot. They then identify the callers and callees for the target method. The sequences of sub-tokens are built in the same manner as in the previous contexts to form the feature for the interaction context. In [131], Yang et al. retrieve the most similar method for the target method from training data by estimating their body code similarity through Lucene, a widely used text search engine. They use the name of the most similar method as the pattern guider. Inspired by the work [69] that demonstrates the promise of using the token sequence of program entities for MNP, they extract corresponding context token sequences from all global contexts, including interface context, enclosing context, and pattern guider. In [4],

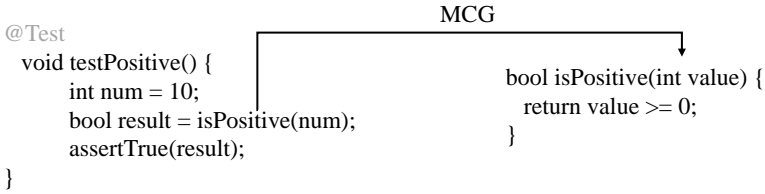


Fig. 15. An example of MCG provided in [81]

Allamanis et al. extract tokens from the names of the class and superclass of the target method. In [142], Zhu et al. believe that the contexts from enclosing classes could provide information on the method's function, while the contexts from siblings and other contextual method names help the model adapt to the naming conventions within specific projects. Therefore, they extract tokens from the enclosing class, including class names, siblings, and class attributes.

• Method Call Graph (MCG).

Definition. MCG allows us to determine which specific methods are called from the target method [81].

Example. Fig. 15 presents an example of the MCG for the given code snippet.

Literature. Ma et al. [64] integrate MCG as part of a program graph to train a generic code embedding model called GraphCode2Vec. They also verify the effectiveness of GraphCode2Vec on the MNP task. They build MCGs via class hierarchy analysis [22]. In [81], Petukhov et al. use Jimple together with Soot to build the call graph between the unit test method and the method under test.

In [132], Yonai et al. propose to enhance MNP by applying graph embedding techniques to the aggregated call graph (ACG). The key idea is to recommend names of methods whose function is similar to a target method in terms of a method embedding that expresses the function of a method, obtained from caller-callee relationships. An ACG is a directed graph similar to a call graph, but one node represents one set of methods with the same name, and a directed edge means there are one or more calls between two method sets. Compared with ordinary call graphs, the design of ACG has more advantages. Caller-callee relationships in the ACG are simplified by identifying nodes only by method names, and the computational cost for calculating embeddings on the ACG is reduced by decreasing the number of nodes and edges.

Although the ACG is the same as a call graph whose nodes are grouped by method names, the ACG is not built via the call graph, but directly from source code to avoid the difficulty of constructing a call graph. Yonai et al. devise an algorithm to build/calculate the ACG of the target method. For each of the method definitions extracted from the code corpus, the algorithm first obtains the name of the method and the set of all method names contained in the body. Then, it adds all those method names to the vertex set V_A of ACG. Next, edges from the name of the definition to each method name in the body are added to the edge set E_A of the ACG. The calculation of the ACG does not require semantic analysis, so the computational cost is low. Additionally, since each source file is processed independently, the process can be parallelized easily.

4.1.3 Documentation context. A method does not exist in isolation, a large number of associations can be found in the documentation. The documentation of the method can describe the functionality

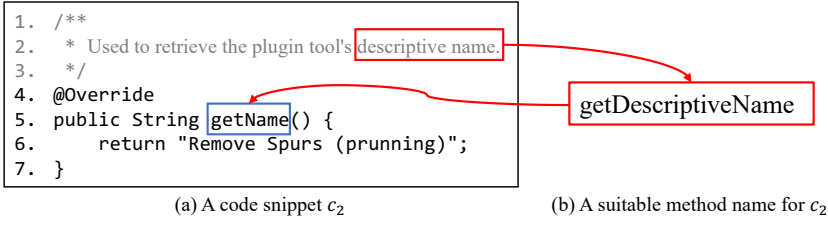


Fig. 16. An example where documentation can facilitate MNP

of the method and the role it plays in the project [59]. The documentation can also provide rich information about the methods, which will help in predicting method names. The work [59] provides a definition of documentation context, that is, the first sentence of the code documentation is informative, and many code summarization approaches use it as a code summary [43, 54, 100]. Following them, they use the tokens in the first sentence as the documentation context. Fig. 16 shows an example of using the documentation to facilitate MNP. It is observed that the body code of the method c_2 cannot provide enough information to suggest a suitable name, while the documentation of c_2 (lines 1–3 of Fig. 16(a)) can provide a useful indication for predicting the method name `getDescriptiveName` shown in Fig. 16(b).

For each method with a comment, to get its documentation context, the work [59] extracts the first sentence that appeared in its Javadoc description since it typically describes the functionalities of the method. Then they delete the punctuations and split the sentence with space to get words and lowercase the words. All the words are concatenated to form the documentation context. Gao et al. in [32] use Eclipse Java Development Tools to parse source files into abstract syntax trees, extract method declarations with Javadoc comments, derive method descriptions from the first sentence of comments, and prepare datasets with tokenized natural language method descriptions and method names split into subtokens.

4.2 Context Preprocessing

In this section, we summarize and exhibit the preprocessing methods for the aforementioned local and global contexts used by the reviewed MNP studies. Fig. 17 presents the context preprocessing methods proposed by existing MNP studies. Note that not all contexts have corresponding preprocessing. For example, Qu et al. [84] and Yonai et al. [132] directly feed CRG (Code Relation Graph) and ACG (Aggregated Call Graph) respectively to the subsequent MNP models for learning without special preprocessing. We will introduce existing preprocessing methods according to their preprocessing objects, i.e., contexts.

• Token.

After extracting (sub)tokens from different local, global, or documentation contexts, a common preprocessing is to concatenate tokens from these different sources.

① Token Concatenation.

Definition. Token concatenation is a technique where tokens from different contexts are combined into a sequential representation. Nguyen et al. define the process of token concatenation for three distinct contexts: the implementation (IMP), interface (INF), and enclosing (ENC) contexts. Tokens from these contexts are concatenated into a sequence, separated by periods (.). For the interface (INF) context, the input and output parts are separated by commas (.). In both the implementation (IMP) and interface (INF) contexts, the tokens representing names and types are arranged according

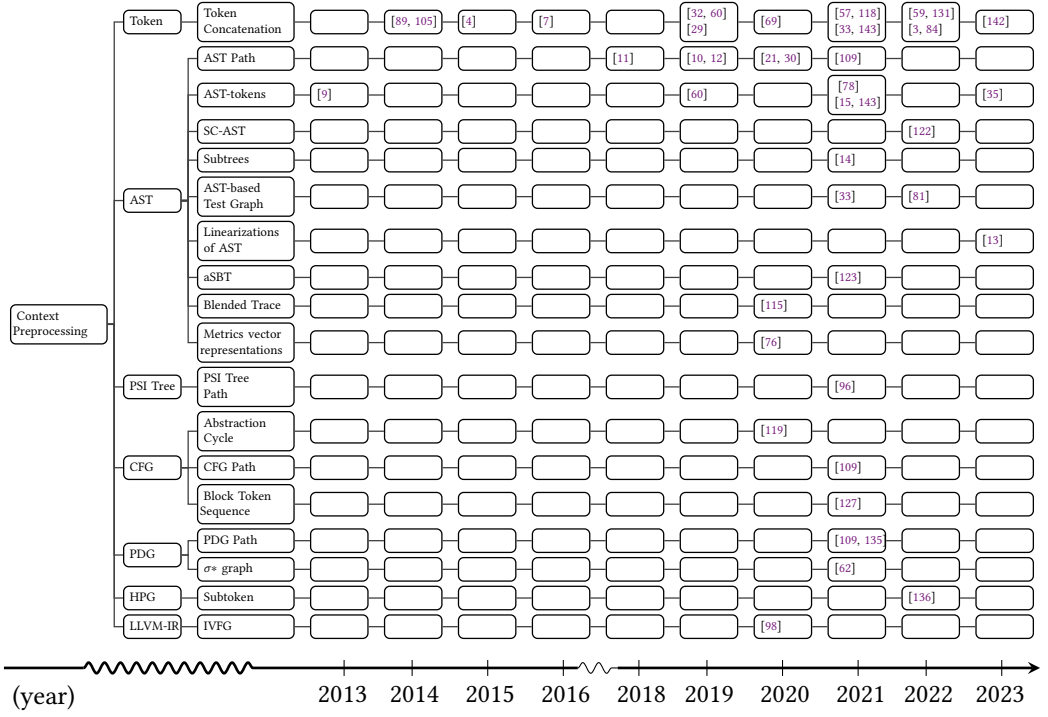


Fig. 17. Evolution of context preprocessing

to their order of appearance in the code. This method of concatenation forms a structured sequence used for further processing.

Literature. Nguyen et al. have employed token concatenation in various studies. In [57, 69, 118, 131], they concatenate tokens from the IMP, INF, and ENC contexts to form a sequential representation, with tokens separated by periods and structured according to their respective contexts. Additionally, Zhu et al. [142] emphasize the importance of token sequence length in the performance of models, limiting the total sequence length to 512 tokens to prevent longer contexts from overshadowing shorter ones.

Other studies [3, 4, 59, 84] follow a similar approach by concatenating subtokens in the order they appear in the source code to form the local feature's sequential representation. Tokens are ranked by their probabilities of forming the method name based on their context. For tied tokens, their order in the callee method's token sequence is used to rank them further. Additionally, these works assign each token an indicator based on the context from which it is extracted, with up to 35 different context-indicator pairs (e.g., <Enclosing context, ClassName>, <Callee implementation context, ReturnStatement>).

• Abstract Syntax Tree (AST).

Several AST preprocessing methods proposed in existing MNP research are summarized below.

① AST Path.

Definition. Alon et al. [11] propose a method of decomposing abstract syntax trees (ASTs) into paths to capture and generalize meaningful properties across programs. An AST path is a sequence of nodes connected by movement directions (up or down) within the tree. These paths can represent recurring patterns in code while preserving distinct characteristics between programs. An AST

path-context is defined as a triplet consisting of an AST path and the values at its start and end nodes, with these values generalized through an abstraction function to improve the model's ability to generalize across different codebases.

Literature. In their original work [11], Alon et al. introduce two key optimization strategies to handle the computational complexity of processing AST paths and to enhance prediction performance. The first strategy involves different abstractions, such as encoding the full path without symbols (No-arrows), discarding path directions and node order (Forget-order), focusing on specific nodes (First-top-last, First-last, Top), or treating all paths the same (No-paths). The second strategy introduces two hyperparameters, max_length and max_width , which limit the number of extracted paths, with their optimal values determined by grid search on a validation set.

This method of AST decomposition has been employed in their later works, code2vec [12] and code2seq [10], and has been adopted by other researchers such as Compton et al. [21] and Vagavolu et al. [109], indicating its influence in the field of program representation and prediction.

② Statement-Centered Abstract Syntax Tree (SC-AST).

Definition. SC-AST is a specialized form of AST that focuses on program statements. Introduced in [122], SC-AST is constructed by extracting statement nodes from a standard AST and adjusting their structure. In this process, each statement is treated as the root node of a subtree, and the direction of edges within these subtrees is reversed to highlight the relationships between statements. This reconfiguration helps capture detailed statement-level information more effectively.

Literature. The concept of SC-AST was proposed by Zhang et al. [122] to address the limitations of traditional ASTs in capturing statement-level details. By focusing on program statements and reversing subtree edge directions, SC-AST enables improved analysis of relationships between statements. This technique is particularly useful in tasks that involve learning statement representations, such as using Gated Graph Neural Networks (GGNN) for further analysis. The SC-AST method enhances the model's ability to learn from the structure and semantics of program statements, offering better insights compared to conventional AST representations.

③ Subtrees.

Definition. Subtrees are smaller, meaningful segments of ASTs used to represent code in a more granular way. Bui et al. [14] conceptualize subtrees in ASTs as analogous to “words” in documents. These subtrees are extracted by traversing an AST and selecting specific nodes that meet certain conditions, such as nodes of the type `expr`. The selected subtrees form a vocabulary, and this vocabulary is used to encode ASTs of any size through a self-supervision mechanism. The subtrees provide a fine-grained representation of the code, facilitating learning embeddings for parsable code snippets.

Literature. In [14], Bui et al. utilize subtrees preprocessed from AST to learn the source code encoder under a self-supervision mechanism. They consider ASTs analogous to documents and subtrees in ASTs analogous to “words” in documents, and adapt the idea of document embedding to learn embeddings of ASTs of any size by using an encoder that can encode ASTs of any parsable code snippet. For each AST, they identify a set of subtrees, and all of the subtrees are accumulated into a vocabulary of subtrees.

They implement the extraction of subtrees by traversing ASTs. During traversing an AST, every visited node satisfying a certain condition, e.g., of the type `expr`, leads to a subtree rooted at the visited node. In their experiments, they chose to select the subtrees whose root nodes are of the types `{expr_stmt, decl_stmt, expr, condition}`. They consider these relatively fine-grained code elements because they are usually meaningful but yet still small enough to be considered as the frequent “words” in the vocabulary of subtrees from a large code base. Such small code elements often have similar meanings when their syntactical structure is similar even though their textual appearance may be different (due to different identifier names, such as `int n = arr.length` versus

`int m = x.length`). In addition, they also consider nodes that represent a single keyword, e.g., `if`, `for`, `while`, and these nodes can be seen as the subtrees with size = 1. They do not consider too coarse-grained subtrees, such as the whole `if`, `while`, `for` statements, as those subtrees are often big so that (1) each of them, as an individual vocabulary word, may appear too infrequent in the code base for the encoder to learn a meaningful representation for it directly; (2) syntactical differences among big subtrees do not necessarily mean the corresponding code has vary meanings, while the encoder may have a harder time to recognize the semantic similarity among them.

④ AST-based Test Graph.

Definition. An AST-based Test Graph is a graph representation built from the abstract syntax tree (AST) of a unit test method and the method under test (MUT). Petukhov et al. [81] construct this graph by first extracting the AST from the source code file containing the unit test and then augmenting it with semantic edges, such as data-flow and control-flow connections between variables. The test graph is further refined by cutting a subgraph related to the unit test body and adding control-flow edges that capture the execution flow. The graph is expanded by including the ASTs of the MUTs, which are identified through a call graph, and connecting them to the test graph via new control-flow edges.

Literature. Petukhov et al. [81] introduce the concept of an AST-based Test Graph to enhance the training of a Method Name Prediction (MNP) model, specifically aimed at generating names for unit test methods. They build the test graph by first extracting the AST from the source code and then adding semantic edges to represent data-flow and control-flow dependencies. To focus on the unit test body, they cut a relevant subgraph from the AST and augment it with additional control-flow edges that reflect the execution progress. The method under test (MUT) is identified via a call graph, and the process is repeated for each MUT, after which the unit test and MUT graphs are connected. This comprehensive graph structure helps to capture both the structural and semantic relationships between unit tests and the methods they are testing, facilitating better name generation.

⑤ Linearizations of AST.

Definition. Linearization of ASTs is a process where ASTs are converted into a sequential format to represent code statements. In the fold2vec model proposed by Bertolotti et al. [13], the linearization process involves parsing a method into an AST, splitting it into subtrees (each representing a statement), and then linearizing these subtrees using pre-order traversal. Non-terminal and terminal nodes are registered during this traversal, and integer literals are removed for efficiency. This tree unfolding technique, also called Contextual Unfolding (CU), provides distinct sequences of terminal and non-terminal nodes to capture both the structural and topical aspects of the code.

Literature. In [13], Bertolotti et al. propose a model called fold2vec, which uses linearized ASTs to represent code statements. They evaluate fold2vec on the MNP task. The process to extract linearized ASTs involves: 1) Parsing the method into an AST. 2) Splitting the AST into sub-trees, each representing a statement. 3) Linearizing these sub-trees in two ways: pre-order visits registering non-terminals and terminals. 4) Tokenizing terminals and removing integer literals. Tree unfoldings, or linearizations, are used to create a hybrid tree representation by folding sub-trees into more informative nodes, thus reducing tree height. These are specific tree unfoldings for ASTs, derived from sub-trees of each statement AST. There are two types of Contextual Unfoldings (CUs): one for non-terminal nodes (e.g., `IfStmt`, `ForStmt`) and one for terminal nodes (e.g., identifiers, literals, keywords). These provide different information to the neural network, capturing code structure and topic, respectively [72]. Fold2vec defines helper functions to calculate CUs. Given a node sequence, these functions return only terminal or non-terminal nodes. They are defined inductively over the length of their inputs.

Additionally, a function linearizes a statement tree via a pre-order visit, skipping sub-trees of other statements. This function is applied to the root node of every statement sub-tree. The extraction function combines these helper functions to calculate token sequences for fold2vec. It starts from the AST root node and uses pre-order traversal, producing CUs at each statement node and concatenating them in the output. They also split terminal nodes into sub-tokens (e.g., “toString” becomes “to” and “string”) using ronin [45] to keep the embedding table size manageable. Integer literals are removed from the output to reduce the word embedding size. Information on parameters and return types is included, but method declarations are omitted for simplicity.

⑥ Advanced structure-Based Traversal (aSBT).

Definition. ASBT is an enhanced method for traversing ASTs that generates two linear sequences: one capturing the tokens in the code and the other representing the type information. These sequences provide a detailed view of both the code structure and its syntactical elements. In aSBT, these two sequences are combined into a single embedded sequence through an embedding layer, which integrates both token and type information for further processing.

Literature. Xie et al. [123] introduced the aSBT method to improve the representation of ASTs in code analysis. Using aSBT, they transformed the AST into two sequences: one sequence represents the code tokens, while the other captures the type information from the AST. These sequences are then embedded together into a single aSBT token embedded sequence using an embedding layer. This enhanced traversal technique allows for a richer representation of the code, capturing both syntactic and structural features more effectively compared to traditional AST traversal methods, thus improving performance in tasks such as code understanding and prediction.

⑦ Blended Trace.

Definition. Blended Trace is a sequence that combines symbolic and concrete execution details of a program. Each entry pairs a symbolic statement with the actual program states observed during execution, capturing both the code structure and specific execution details.

Literature. In [115], the Blended Trace is introduced to represent how symbolic statements map to real program states. By combining symbolic and concrete execution details, it captures both the structure of the code and the specific execution details, providing a comprehensive view of program behavior.

⑧ Metrics Vector Representations.

Definition. Metrics Vector Representations quantify aspects of source code quality, such as size, complexity, cohesion, and coupling, independent of lexical properties

Literature. In [76], Metrics Vector Representations are introduced to measure different characteristics of source code quality. By focusing on functional and structural properties, these metrics provide a straightforward, effective, and cost-efficient way to vectorize source code, avoiding the influence of lexical properties.

• PSI Tree.

① PSI Tree Path.

Definition. PSI Tree Path refers to the process of preprocessing the PSI tree into paths. This involves handling types in the same manner as other parts of the *path-context*, where type names are split into subwords.

Literature. In [96], Spirin et al. follow [10] also preprocess the PSI tree into paths. They preprocess the PSI tree into paths and suggest treating types similarly to other elements of the *path-context* by splitting type names into subwords.

• Control Flow Graph (CFG).

Existing MNP studies mainly adopt the following methods to preprocess CFG, including Abstraction Cycle and CFG Path.

① Abstraction Cycle.

Definition. Abstraction Cycle in the GINN involves three operators: partitioning, heightening, and lowering. These operators iteratively abstract and recover the CFG to learn semantic embeddings until sufficient propagation is achieved.

Literature. In [119], Wang et al. propose GINN, which uses an abstraction cycle to learn semantic embeddings from CFGs. The cycle employs partitioning, heightening, and lowering operators to iteratively process the CFG, ensuring comprehensive message passing between nodes.

② CFG Path.

Definition. CFG Path is a sequence in a CFG that starts at a node, traverses intermediate nodes and ends at an END node or a loop. It captures control flow patterns, characterized by node types and the direction of the last node.

Literature. In [109], Vagavolu et al. enhance the code2vec model [12] by incorporating CFG paths. These paths capture control flow patterns, including different loop executions and conditionals, to improve semantic representation in code analysis.

• Program Dependence Graph (PDG).

Existing MNP studies mainly preprocess PDG into PDG paths [109, 135]. In [62], Liu et al. perform a simple preprocessing on the PDG, including splitting each node's feature into subtokens based on the delimiter “.” and enriching PDG with inverse edges. Ma et al. [64] directly use the raw PDG without any preprocessing.

① PDG Path.

Definition. PDG Path is a sequence of nodes and edges on a PDG, where nodes represent statements and edges represent control or data flow. PDG paths are generated using a random walk strategy, treating each path as a sentence for model training. Nodes are represented by their statements, and edges by their flow types. The resulting sequences are pre-processed into a set of sentences for each method.

Literature. In [135], Zhang et al. introduce Meth2Seq, which represents methods as collections of PDG paths. These paths are generated using a random walk strategy and pre-processed into sentences for model training. Similarly, Vagavolu et al. [109] preprocess PDGs into PDG paths but define PDGs, PDG paths, and PDG path contexts differently. They represent PDG paths as sequences of node types and movement directions, with all movement directions standardized as downward.

② σ Graph.

Definition. σ Graph is a graph designed to analyze program structures for detecting security vulnerabilities and concurrency issues. It includes σ_0 graphs, which represent control and data flow, and σ_1 graphs, which add detailed semantic information.

Literature. In [62], σ graphs are introduced to enhance program analysis. σ_0 graphs depict control and data flow, while σ_1 graphs add semantic details. These graphs are used for detailed method-level analysis to detect security vulnerabilities and concurrency issues.

• Heterogeneous Program Graphs (HPG).

① Subtoken.

Definition. As mentioned in Section 4.1, HPG is proposed by Zhang et al. [136]. After building HPG, to capture the semantics and to avoid the out-of-vocabulary (OOV) problem caused by large vocabulary size, they employ the subtoken technique in HPG, by splitting the terminal nodes into multiple parts based on the camel case or the snake case [4].

Literature. Zhang et al. [136] introduce a new node type, subtoken, and a new edge type, subtoken_of, into HPG. The original terminal nodes are split into multiple subtoken nodes by the value, and new subtoken_of edges are inserted to indicate the origin of the subtoken nodes. For example, the identifier node with the value “train_model” is split into two subtoken nodes – “train” and “model”, with two subtoken_of edges inserted from the new subtoken terminal

nodes to the original “train_model” node. In addition, the reversed edge of subtoken_of is also inserted into HPG during splitting. Since there may be multiple tokens to be split, they devise two schemes: shared subtoken and independent subtoken. In the shared subtoken scheme, the identical subtoken node is shared among different identifier nodes. The “model” subtoken is shared by two identifier nodes, i.e., “train_model” and “test_model”. This scheme may effectively reduce the size of the graph, and previous work [5] has demonstrated that a similar strategy is possible to get good semantic representations on the subtoken nodes. In the independent subtoken, they treat the subtoken nodes of each identifier independently. The subtoken nodes of “train_model” and “test_model” are independently separated. This scheme may obtain polysemous representations for the same subtoken, but it may suffer from large graph sizes as there are too many subtoken nodes.

- **LLVM-IR.**

- ① **Interprocedural Variable Flow Graph (IVFG).**

Definition. IVFG is a cross-procedural graph based on LLVM-IR for analyzing program control flow and alias-aware data flows. It captures def-use chains of variables using a multi-edged directed graph, with nodes representing instructions and edges denoting def-use relationships

Example. Fig. 14 shows a code fragment and its corresponding LLVM-IR and IVFG, with variable type information omitted for simplicity. In this example, certain variables are indirectly accessed, such as when a store operation involves a top-level pointer in LLVM’s partial SSA form [53].

Literature. In [98], IVFG is introduced for analyzing control flow and data flows in programs using LLVM-IR. This method is detailed further in [94], using SSA form for top-level variables and interprocedural memory SSA form for address-taken variables.

4.3 Context-based Prediction

4.3.1 Context-based Classification.

Context-based classification techniques typically build an MNP model based on an encoder and a classifier. The encoder is a neural network that can encode the original method code body or intermediate representations (i.e., context preprocessing results) into a fixed-length vector representation (i.e., embedding). To build such an encoder, existing MNP techniques have tried various neural network architectures and pre-trained models. The classifier is also a neural network that can predict the embedding produced by the encoder to the label corresponding to the word. Below we will introduce in detail the design schemes of the encoder and classifier that appear in existing MNP studies.

- **Context Encoding**

Fig. 18 displays the evolution over time of the context encoding techniques (i.e., encoders) proposed in existing context-based classifying MNP techniques. We group these encoders based on the context information processed by them.

- **Token.**

- ① **Logbilinear Context Model (LCM) + Subtoken Context Model (SCM).**

Definition. LCM and SCM are two neural model-based encoders for code tokens proposed by Allamanis et al. [4]. The LCM maps each target token and context to vectors in a D -dimensional space called embeddings. It predicts token appearance by comparing the embeddings of context and target tokens, using both local and global contexts. The local context is similar to a logbilinear Language Model (LM), while the global context uses binary feature functions based on context tokens. The SCM predicts new identifier names by breaking them into subtokens using camel case and underscores. It predicts a sequence of subtokens to form a full identifier, using embeddings to represent and combine the context and subtoken influences.

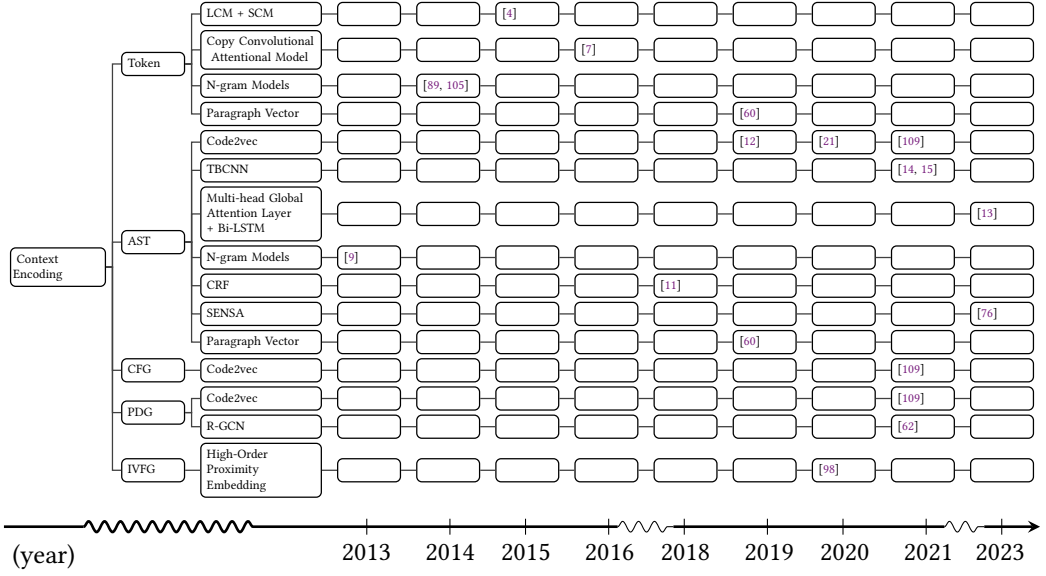


Fig. 18. Evolution of context encoding used in context-based classifying techniques

Literature. Allamanis et al. [4] propose the LCM and the SCM for code token prediction. The LCM builds on the logbilinear model [67], mapping tokens and contexts to embeddings and predicting token appearance based on the similarity of these embeddings. It uses both local and global contexts to form the context representation. The SCM addresses the challenge of predicting new identifier names by splitting them into subtokens and predicting these subtokens sequentially. It uses a logbilinear model to convert scores into probabilities, with embeddings representing surrounding tokens and previous subtokens. This approach allows the SCM to effectively predict new identifiers by leveraging subtoken information.

② Copy Convolutional Attentional Model.

Definition. Copy Convolutional Attentional Model is a neural network model proposed by Allamanis et al. [7] for generating method names from code snippets. It uses a convolutional network to extract features and an attention mechanism to predict each subtoken of the method name. The model also includes a copying mechanism to handle out-of-vocabulary tokens.

Literature. In [7], Allamanis et al. introduce a model for method name generation that processes code subtokens using convolutional networks and attention to predict method name subtokens sequentially. It incorporates a coping mechanism to manage out-of-vocabulary tokens, effectively summarizing code snippets into method names.

③ N-gram Model.

Definition. The N-gram model is a generative probabilistic model used to predict the probability of the next word based on the preceding n-1 words. It simplifies the modeling process by assuming that only the previous n-1 words are needed to predict the current word, thereby calculating the joint probability of the entire sequence of words. The parameters of the N-gram model are estimated by counting the frequency of n-grams in the training corpus. However, smoothing methods are typically employed to avoid the zero probability issue.

Literature. In [105], Suzuki et al. applied the N-gram model to estimate the probability distribution of token sequences in code. The N-gram model can predict the next possible token by analyzing

the probability distribution of tokens within code snippets. When applied to MNP, the N-gram model can recommend the next word for a method name by calculating the conditional probability for a given prefix, thus generating a comprehensible and complete method name.

④ Paragraph Vector.

Definition. Paragraph Vector is a technique used to embed method names into numerical vector spaces. This technique is part of a broader approach that processes method names and bodies from a code corpus to identify inconsistent method names and suggest appropriate ones.

Literature. In [60], they use unsupervised learning to embed method names with the paragraph vector technique. Method body tokens are embedded using Word2Vec and CNNs, with padding for varying token lengths. The resulting vector spaces help retrieve similar method bodies and suggest appropriate method names.

• Abstract Syntax Tree (AST).

① Code2vec.

Definition. Code2vec is a technique for encoding AST paths into vector representations. It learns embeddings for paths, names, and tags, and uses a fully connected layer and an attention mechanism to aggregate multiple context vectors into a single vector representing a code snippet.

Literature. In [12], Alon et al. propose code2vec to encode AST paths. The method involves learning embeddings for paths, names, and tags, and using a fully connected layer to combine these embeddings into context vectors. An attention mechanism aggregates these context vectors into a single vector representation of the code snippet. Several subsequent works [21, 109] also adopt code2vec-based encoders to encode AST paths.

② Tree-Based Convolutional Neural Network (TBCNN).

Definition. TBCNN is a neural network designed to encode ASTs by capturing their structural information. It converts AST nodes into vector representations, applies a tree-based convolution kernel to extract structural features, uses dynamic pooling to gather information, and processes the features through hidden and output layers to obtain the AST representation.

Literature. In [14], Bui et al. adopt TBCNN [68] to encode AST. TBCNN employs an unsupervised pre-training approach to learn vector representations of program symbols. These vectors are then used in a tree-based convolutional layer for supervised learning. The convolutional layer applies fixed-depth feature detectors over the tree, functioning like convolution with finite support kernels, to capture the structural information of the AST.

③ Multi-head Global Attention Layer + Bi-LSTM.

Definition. The Multi-head Global Attention Layer combined with Bi-LSTM is an encoder architecture for processing Contextual Unfoldings (CUs) of ASTs. Terminal tokens are processed using a multi-head global attention layer to obtain weighted sums, while non-terminal tokens are processed through a Bi-LSTM to preserve positional information. The intermediate representations of terminals and non-terminals are concatenated and refined using a self-attention layer.

Literature. In [13], Bertolotti et al. devise an encoder built on a multi-head global attention (GA) layer and Bi-LSTM to encode CUs. CUs are just tree unfoldings designed to deal with the AST, detailed in Section 4.2. They first convert terminal and non-terminal tokens into embeddings obtained from the vector table E of 10,000 entries of 100 floats. E is trained with back-propagation and the rest of the network. Terminal tokens are processed with the GA layer to obtain weighted sums, while non-terminal tokens are processed through a Bi-LSTM to maintain positional information. The intermediate representations of terminals and non-terminals are concatenated to form Contextual Unfolding Intermediate Representations (CUIRs), which are further refined using a self-attention (SA) layer to enhance the overall representation by mixing relevant CUIRs.

• Control Flow Graph (CFG).

① Code2vec.

Definition. In [109] Vagavolu et al. encode CFG paths utilizing code2vec as the encoder. The detailed design of code2vec has been discussed in Section 4.3.1–① Context Encoding–• Abstract Syntax Tree (AST).

Literature. Vagavolu et al. [109] encode CFG paths using code2vec, focusing on context encoding via ASTs. Ma et al. propose GraphCode2Vec [64], a GNN-based encoder for CFGs. GraphCode2Vec leverages four GNN architectures (GCN [50], GraphSAGE [37], GAN [112], GIN [126]) to create dependence embeddings for program instructions through message-passing. These embeddings are aggregated via global attention pooling to form program-level dependence embeddings, emphasizing important nodes. The final program embedding is obtained through concatenation, which preserves feature information. GraphCode2Vec employs self-supervised pre-training strategies, including node classification, context prediction, and variational graph encoding (VGAE) [49], allowing the model to learn from raw datasets without human supervision.

- **Program Dependence Graph (PDG).**

- ① **Code2vec.**

Literature. In [109], Vagavolu et al. encode PDG paths utilizing code2vec as the encoder. The detailed design of code2vec has been discussed in Section 4.3.1–① Context Encoding–• Abstract Syntax Tree (AST).

- ② **Relational Graph Convolutional Network (R-GCN).**

Definition. An R-GCN is designed to process heterogeneous graphs containing multiple node and edge types. Each node is initialized with a feature vector, and the R-GCN updates node representations by aggregating information from neighboring nodes through a rule that involves normalization and learnable parameters. A nonlinear activation function is applied during this process, and the final layer's representation serves as the output.

Literature. In [62], Liu et al. use an R-GCN-based encoder to process PDG. They work with a heterogeneous graph that has multiple node and edge types. Each node is associated with a feature vector. The R-GCN updates node representations through a specific rule involving neighboring nodes, normalization, and learnable parameters, with a nonlinear activation function applied. The initial node features are used to start the process, and the final layer's representation is considered the final output. For more details on the normalization method, they refer to [92]. In [64], Ma et al. also utilize GraphCode2Vec as an encoder to encode PDG. The detailed design of GraphCode2Vec has been discussed in Section 4.3.1–① Context Encoding–• Control Flow Graph (CFG).

- **IVFG**

- ① **High-Order Proximity Embedding.**

Definition. High-Order Proximity Embedding is a method for analyzing source code by capturing high-order proximity relationships. It involves compiling code into LLVM-IR, performing pointer analysis to generate a memory SSA form and an IVFG, and using matrix decomposition techniques like the Katz Index and Singular Value Decomposition to achieve high-order proximity embedding.

Literature. In [98], Sui et al. propose Flow2vec, a method for high-order proximity embedding to analyze source code. The process includes compiling code into LLVM-IR, generating a memory SSA form and IVFG through pointer analysis, and resolving indirect function calls. The method uses the Katz Index and SVD to decompose the proximity matrix, efficiently approximating embedding vectors for sparse graphs and preserving context-sensitive value-flows.

- ② **Context Classifying**

Fig. 19 presents the evolution over time of the context classifying techniques (i.e., classifier) proposed in existing context-based classifying MNP techniques. Since the design of the classifier is context-independent, we will independently introduce each type of classifier as follows.

- **Conditional Random Field (CRF).**

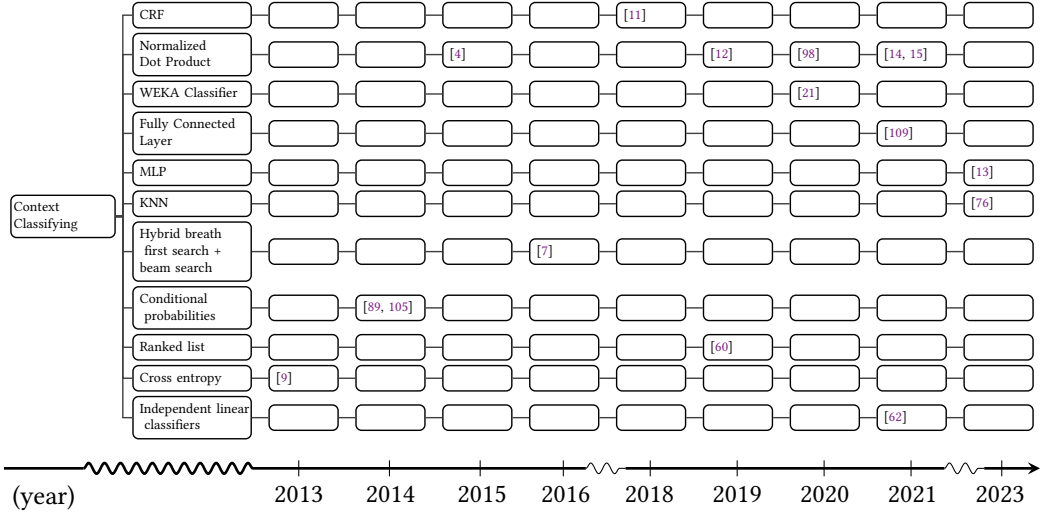


Fig. 19. Evolution of context classifying used in context-based classifying techniques

Definition. CRF is a probabilistic model used for predicting structured outputs, represented as a conditional distribution with an associated graphical structure. It is commonly applied in fields like natural language processing, bioinformatics, and computer vision. CRFs model the relationships between variables using factors that measure compatibility, ensuring the distribution sums to 1 through a normalization factor.

Literature. Alon et al. [11] use a CRF-based classifier to predict method names from AST path embeddings. Building on the work [11], they adapt CRFs to use AST paths instead of original factors. CRFs are implemented using Nice2Predict, with added support for unary factors to capture relationships between different occurrences of the same identifier. This approach allows CRFs to output single predictions for each program element and suggest top candidate names for methods.

• Normalized Dot Product.

Definition. The Normalized Dot Product is a technique used to predict method names by calculating the dot product between a code vector and tag embeddings, followed by softmax normalization. This method leverages the relationship between code representations and label distributions to make predictions.

Literature. Alon et al. [12] introduce the use of the normalized dot product in code2vec to predict method names, hypothesizing that label distribution can be inferred from code AST paths. The predicted distribution is obtained by applying softmax normalization to the dot product between the code vector and tag embeddings. Bui et al. [14] and Ma et al. [64] adopt this method using their pre-trained encoders, InferCode and GraphCode2Vec, respectively. Allamanis et al. [4] also use a similar approach, employing a logbilinear context model and a subtoken context model of code, with beam search to find the most probable subtoken sequences.

• WEKA Classifier.

Definition. The WEKA classifier is a machine learning framework used to perform classification tasks. It can process numerical embeddings, such as those generated by code2vec, to predict target labels like method names.

Literature. Compton et al. [21] utilize WEKA to predict method names from code vectors produced by code2vec. They test various classifiers and find that a linear SVM, implemented with WEKA's LibLinear, provides the best performance for this task.

- **Fully Connected Layer.**

Definition. A Fully Connected Layer is a neural network layer where each neuron connects to every neuron in the previous layer. Combined with a softmax activation function, it generates a probability distribution for classification tasks such as predicting method names.

Literature. Vagavolu et al. [109] use a fully connected layer with softmax activation to predict method names from code vectors generated by code2vec. The layer outputs a probability distribution over method names, with the weight matrix rows corresponding to the labels.

- **Multi-Layer Perceptron (MLP).**

Definition. A MLP is a neural network with multiple dense layers, each applying a linear transformation and a non-linear activation function. It is used for tasks like classification by transforming input vectors into output vectors.

Literature. Bertolotti et al. [13] use an MLP layer in Fold2vec to predict method names from code vectors. The MLP reduces the hidden representation to a 300-dimensional vector, followed by a linear transformation to map this vector to an output space with 261,245 method name labels. Each label is assigned a score, and a softmax layer normalizes these scores to probabilities, producing a list of method names sorted by likelihood.

- **K-Nearest Neighbors Algorithm (KNN).**

Definition. The KNN algorithm is a method for classification and regression that ranks data points based on their proximity using distance metrics like Euclidean or Manhattan distance.

Literature. Parsa et al. [76] use KNN to predict method names by measuring the similarity between a target method and other methods in a dataset. They rank the top 10 suggested names based on similarity and the average similarity of methods within the same class. The approach normalizes vectors to ensure unit norms and computes similarity using various distance metrics. Each method is represented as a vector of software metrics, and the names of other methods in the enclosing classes are considered to find the most similar methods in a reference dataset.

- **Others.**

In [89, 105], they predict the next word in a method name by calculating the conditional probabilities of candidate words based on the given prefix and then suggest the next word in descending order of these probabilities. In [8], they predict a full method name by using a hybrid approach that combines breadth-first search and beam search, where we iteratively generate and evaluate subtoken sequences based on conditional probabilities, maintaining a priority queue of the top predictions and pruning less likely candidates to efficiently find the best k suggestions. In [60], they predict method names by generating a ranked list of similar names based on method body similarities, utilizing four distinct ranking strategies that consider factors such as similarity scores, group sizes, and group averages to refine the suggestions. In [62], they predict method names by treating them as sequences of subtokens and using independent linear classifiers to predict each subtoken, with attention pooling to generate a single method-level embedding for the prediction. In [9], they use cross entropy to predict method names by evaluating how well a language model predicts each token in a sequence, with lower cross entropy indicating higher confidence and better predictive accuracy for the method name.

4.3.2 Context-based Generation.

This type of MNP technique typically trains a method name generation model based on an encoder-decoder architecture. Similar to context-based classification, the encoder of the generation

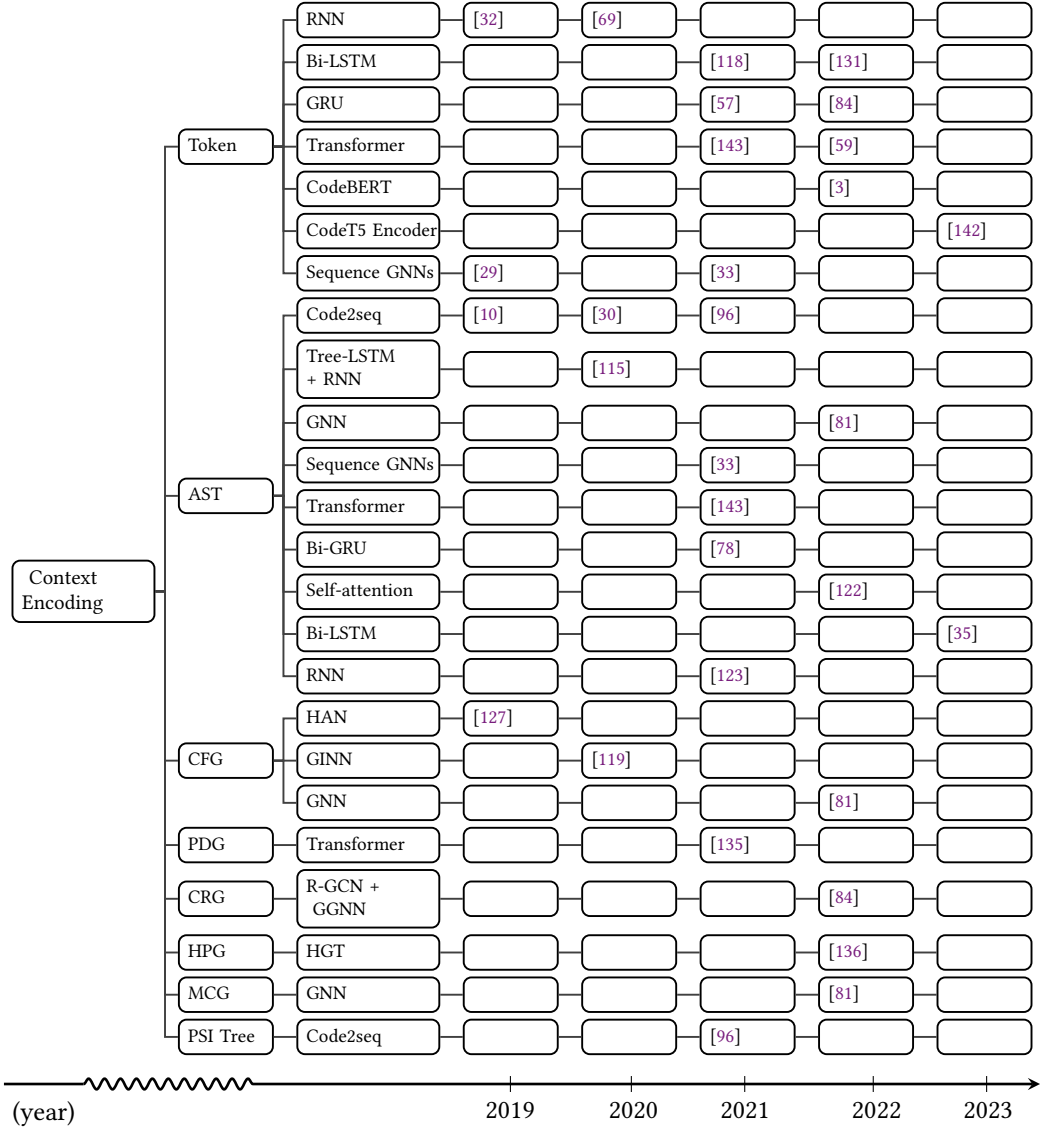


Fig. 20. Evolution of context encoding used in context-based generation techniques

model is also a neural network that can encode the original method code body or intermediate representations (i.e., context preprocessing results) into an embedding. To build such an encoder, existing learning-based MNP techniques have tried various neural network architectures and pre-trained models. The decoder is also a neural network that can decode the embedding produced by the encoder to the predicted method name. Below we will introduce in detail the design schemes of encoder and decoder that appear in existing MNP studies.

❶ Context Encoding

Fig. 20 presents the evolution over time of the encoders devised in context-based generation MNP techniques. We group these encoders based on the context information they encode.

- **Token.**

- ① **RNN.**

Definition. An RNN processes sequential data by embedding input tokens as vectors and encoding them into hidden representations

Literature. Nguyen et al. [69] use an RNN-based encoder in their seq2seq model for method name prediction, embedding input tokens and encoding them into hidden representations.

- ② **Bidirectional Long Short-Term Memory (Bi-LSTM).**

Definition. A Bi-LSTM processes data in both forward and backward directions, capturing context from past and future states to generate hidden states and final encoded sequences.

Literature. Wang et al. [118] and Yang et al. [131] use a single-layer Bi-LSTM for encoding in their models. The Bi-LSTM processes input context tokens and pattern guider tokens into vectors, generating hidden states that are combined to form the final encoded sequences. They use separate LSTMs for encoding context and pattern guide sequences in their experiments.

- ③ **Gate Recurrent Unit (GRU).**

Definition. A GRU is a type of RNN that uses gating mechanisms to control the flow of information, making it effective for handling sequential data. GRUs convert input sequences into hidden vectors, which can be used for various downstream tasks.

Literature. Li et al. [57] use a GRU-based context encoder in DeepName. They generate sub-token vectors with GloVe [79], pad sequences for uniform length, and use multiple GRUs to encode different context structures. The GRU outputs are combined into hidden state vectors, which are processed by an attention layer to emphasize important sub-tokens. In [84], Qu et al. follow the work [57] and also adopt GRU to build the token encoder.

- ④ **Transformer.**

Definition. Transformer is a neural network architecture composed of multiple self-attention and fully connected layers. It uses a multi-head attention mechanism to transform input sequences into output vectors, which are then processed by a feed-forward network to produce the final output.

Literature. Liu et al. [59] use the Transformer architecture to develop an encoder for generating method names from various contexts, including local, project-specific, and documentation. The Transformer employs self-attention layers with multi-head attention to process input vectors, followed by a fully connected feed-forward network to refine the data and produce the final output.

- ⑤ **CodeBERT.**

Definition. CodeBERT is a pre-trained model designed for code representation. It is trained with two main objectives: masked language modeling (MLM) on bimodal data of natural language and programming language pairs, and replaced token detection (RTD) on unimodal data, such as code without paired natural language texts.

Literature. CodeBERT is used as a token encoder in various MNP studies, such as those by Ahmed et al. [3]. The model is pre-trained with MLM, a technique proven effective in models like BERT RoBERTa, and RTD, which leverages large amounts of code data. These pre-training objectives enable CodeBERT to effectively capture the semantics of code for downstream tasks.

- ⑥ **CodeT5 Encoder.**

Definition. CodeT5 is a pre-trained model built on an encoder-decoder architecture, similar to the T5 model. It aims to derive generic representations for programming and natural languages through pre-training on unlabeled source code. CodeT5 uses an Identifier-aware Denoising Pre-training (IDP) objective, which includes Masked Span Prediction (MSP), Identifier Tagging (IT), and Masked Identifier Prediction (MIP).

Literature. Zhu et al. [142] use CodeT5 to encode tokens for MNP. Unlike the encoder-only CodeBERT, CodeT5 employs an encoder-decoder architecture. It is pre-trained with IDP, which involves MSP for predicting masked spans, IT for identifying code tokens, and MIP for masking

all identifiers in the code segment. These pre-training tasks help CodeT5 capture the semantics of code and natural language effectively.

⑦ Sequence GNNs.

Definition. Sequence GNNs integrate sequence encoders with GNNs to capture both sequential and structural information from data. This approach enables comprehensive representations for tasks involving complex relationships within sequences. Sequence encoders, like bidirectional RNNs, map sequences to representations, which are then processed by GNNs. A weighted averaging mechanism computes graph-level representations using a learnable function and logistic sigmoid.

Literature. In [29], GGNNs combined with sequence encoders process sequences and binary relationships, extending to additional nodes like meta-nodes from syntax trees. Efficient processing of large graphs involves flattening graphs in a minibatch into a single graph and using TensorFlow's unsorted segment operations for softmax computations. Ge et al. [33] used a hybrid approach with GGNNs for message passing on graph-structured data and BiLSTM for serializing node representations, capturing complex structures and sequential relationships in source code for better understanding and generation tasks.

• AST.

To convert ASTs or preprocessing results of ASTs (discussed in Section 4.2) into embeddings, existing MNP studies have devised various encoders. The following content will introduce the design of these AST encoders in detail.

① Code2seq.

Definition. As mentioned before, Code2seq is a model that uses an encoder-decoder architecture to generate sequences from code. The encoder processes each AST path individually using a Bi-LSTM, which captures the sequence of nodes and their child indices. Nodes and subtokens are represented with learned embeddings, and the final state of the Bi-LSTM encodes the entire path. The decoder then uses these encoded vectors to produce the target sequence.

Literature. Code2seq, introduced by Alon et al. [10], differs from code2vec by using an encoder-decoder architecture. The encoder generates a vector for each AST path with a Bi-LSTM, capturing node sequences and child indices. Nodes and subtokens are represented with learned embeddings, and the final state of the Bi-LSTM encodes the path. Subtoken vectors are summed to represent the full token. Code2seq combines path and token representations by concatenating them and applying a fully connected layer for further processing. Spirin et al. [96] also use code2seq to encode AST paths in their work.

② Tree-LSTM + RNN.

Definition. A Tree-LSTM and RNN-based encoder processes ASTs and program states. Tree-LSTM recursively updates parent node hidden states from child nodes to embed statements, while RNNs embed program states using variable embeddings. An attention mechanism fuses features to compute embeddings for ordered pairs, and a final RNN captures the flow of the blended trace.

Literature. Wang et al. [115] develop an AST encoder combining Tree-LSTM and RNN to handle blended traces. They use Tree-LSTM for statement embeddings and RNNs for program state embeddings, applying attention to feature fusion. A third RNN captures the overall trace flow, and max pooling creates the final program embedding.

③ Graph Neural Network (GNN).

Definition. GNNs encode graphs by updating node states through message-passing. Each node has an initial state, and messages are exchanged between nodes using a linear function. Node states are updated by aggregating these messages and applying a GRU function. The process is repeated for a fixed number of time steps.

Literature. Petukhov et al. [81] apply GNNs to encode AST-based test graphs. They use message-passing and GRU functions for state updates and introduce an attention-based aggregation layer.

Node states are grouped by type, averaged, and passed through an attention mechanism before being sent to an encoder to reduce output vector size.

④ Bi-directional GRU.

Definition. Bi-directional GRUs process sequences in both forward and backward directions, capturing information from both ends. In path encoding, the final GRU state represents the structural relationship between code tokens or between tokens and the root of a syntax tree.

Literature. Peng et al. [78] use Bi-directional GRUs for encoding paths within syntax trees to capture code structure. They apply relative path encoding for token-to-token paths and absolute path encoding for token-to-root paths, integrating both into the Transformer model's self-attention. Two models are proposed: TPTrans with relative path encoding and *TPTrans* – α combining both encoding methods.

• Control Flow Graph (CFG).

To convert CFGs or preprocessing results of ASTs (discussed in Section 4.2) into embeddings, existing MNP endeavors have designed diverse encoders. The following content will introduce the design of these CFG encoders in detail.

① Hierarchical Attention Network (HAN).

Definition. A HAN applies attention mechanisms at multiple levels, such as tokens and blocks, to encode complex structures like CFGs. It uses a bidirectional GRU to capture context within tokens, followed by attention to assign importance to both tokens and blocks.

Literature. Xu et al. [127] propose HAN to encode CFGs for method name prediction. They embed tokens in basic blocks using a bidirectional GRU and apply attention at the token and block levels. This hierarchical approach generates method representations by adaptively weighting important elements.

② Graph Interval Neural Network (GINN).

Definition. As mentioned before, GINN is a neural network designed to encode semantic embeddings from CFGs. It uses three operators—partitioning, heightening, and lowering—to process node vectors across graph intervals, with each operator manipulating node vectors to capture semantic information.

Literature. Wang et al. [119] introduce GINN to encode CFGs through a principled abstraction method. The partitioning operator computes node vectors for each interval, while the heightening and lowering operators map node vectors between graphs in both directions. These operators, with weighted state vectors, help capture the semantic structure of the CFG.

③ Graph Neural Network (GNN).

Definition. The definition was mentioned earlier, with detailed information discussed in Section 4.3.2–Context Encoding–Graph Neural Network (GNN).

Literature. In [81], Petukhov et al. encode CFG utilizing a GNN-based encoder. The detailed design of GNN has been discussed in Section 4.3.2–① Context Encoding–• AST.

• PDG.

① Transformer.

Definition. The definition was mentioned earlier, with detailed information discussed in Section 4.3.2–Context Encoding–Transformer.

Literature. In [135], Zhang et al. propose a hybrid code representation learning model named Meth2Seq. The encoder of Meth2Seq is built on a Transformer. They utilize the encoder to encode PDG paths and generate a sequence of vectors for each method. The detailed design of GNN has been discussed in Section 4.3.2–① Context Encoding–• Token.

• Code Relation Graph (CRG).

① Relational Graph Convolutional Network (R-GCN) + Gated Graph Sequence Neural Network GGNN.

Definition. R-GCN handles graphs with multiple edge types by accounting for the influence of different relations on nodes. GGNN introduces time-step characteristics in graph message passing, allowing the node hidden states to reflect sequence information over time.

Literature. Qu et al. [84] build a CRG encoder using R-GCN and GGNN. R-GCN extracts features from CRGs, managing the effects of various edge types. To reduce overfitting, weight regularization techniques like basis and block diagonal decomposition are applied. GGNN adds time-step features during message passing, enhancing node representations and allowing better integration of structural and semantic features.

- **Heterogeneous Program Graphs (HPG).**

- ① **Heterogeneous Graph Transformer (HGT).**

Definition. As mentioned before, an HGT processes HPGs by aggregating information from neighboring nodes using mutual attention, accounting for both node and edge types. It includes positional encoding, node embeddings, and attention-based message passing to generate node representations.

Literature. Zhang et al. [136] develop an HGT-based encoder for HPGs. It uses positional encoding from depth-first AST traversal to capture node order and heterogeneous mutual attention to aggregate messages, considering node and edge types. The attention mechanism assigns weights to neighboring nodes, allowing the encoder to generate accurate node representations across multiple layers.

- ② **Context Decoding**

Fig. 21 presents the evolution over time of the context decoding techniques (i.e., decoder) designed by existing context-based generation MNP techniques. Like the classifiers in the context-based classifying techniques, the design of the decoder is context-independent, therefore we independently summarize each decoder as follows.

- **Recurrent neural network (RNN).**

Definition. The definition was mentioned earlier, with detailed information discussed in Section 4.3.2–Context Encoding–RNN.

Literature. Nguyen et al. [69] and Wang et al. [115] both use RNN-based decoders for predicting method names. Nguyen’s decoder predicts each token’s probability using the RNN’s recurrent state, previous token, and an attention-based context vector derived from the encoder’s hidden states. Similarly, Wang’s decoder generates method names by decoding program embeddings, with attention applied to blended traces. Both approaches recompute the context vector for each token, with attention weights calculated by a feedforward neural network. In [119], Wang et al. also adopt RNN as the decoder when they apply their model GINN to the MNP task. The detailed design of GINN is discussed in Section 4.3.2–Context Encoding–Control Flow Graph (CFG).

- **Long Short-Term Memory (LSTM).**

Definition. The definition of LSTM was mentioned earlier, with detailed information discussed in Section 4.3.2–Context Encoding–Long Short-Term Memory (LSTM).

Literature. Wang et al. [118] and Alon et al. [10] both use LSTM-based decoders for method name generation. Wang’s decoder predicts tokens either from a fixed vocabulary or by copying from the input, using the context vector from a Bi-LSTM encoder. The decoder calculates the probability of generating or copying tokens at each step. Alon, in contrast, uses an LSTM decoder that generates tokens based on previously predicted tokens, initializing the state by averaging path representations. The attention mechanism helps select relevant paths during the decoding process, similar to seq2seq models. In [96], Spirin et al. follow the code2seq provided in [10], thus leveraging the same decoder implementation, i.e., LSTM.

- **Single-layer Unidirectional LSTM.**

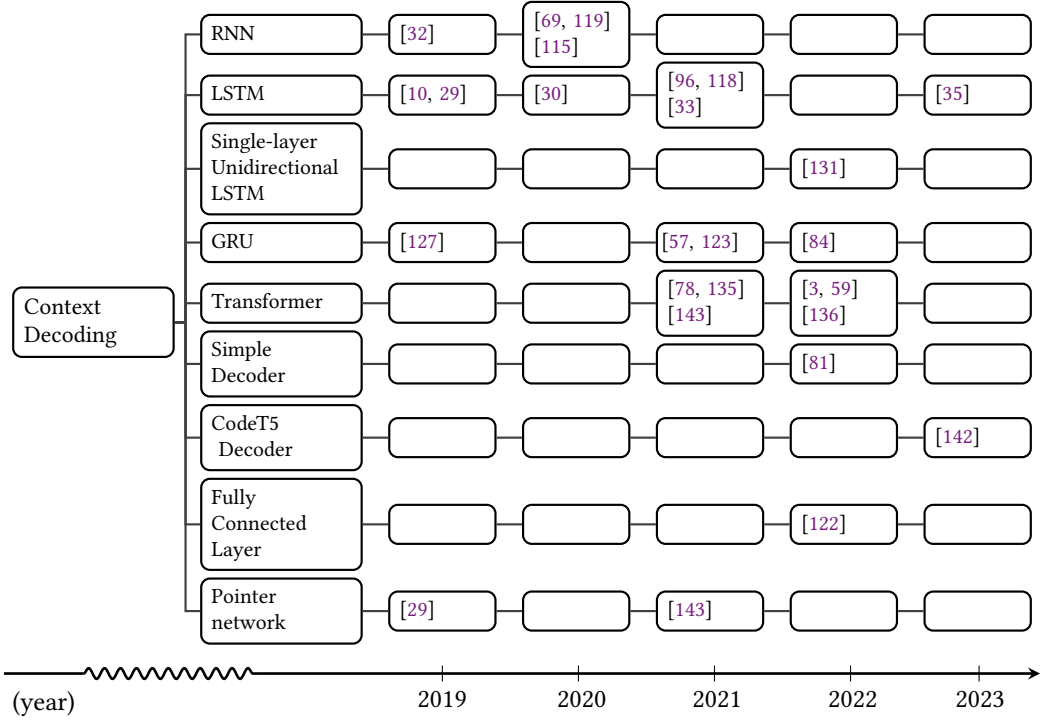


Fig. 21. Evolution of context decoding used in context-based generation techniques

Definition. A single-layer unidirectional LSTM is a RNN with one layer that processes sequences in a single direction, updating its hidden state based on the previous hidden state and current input, either from the ground truth during training or the previously emitted word during testing.

Literature. Yang et al. [131] propose a method name decoder using a single-layer unidirectional LSTM. During decoding, the hidden state is updated using the previous hidden state and a word embedding—ground-truth during training and the previously emitted word during testing. The model can copy tokens from the input or generate them from a fixed vocabulary. The generation probability is based on the pattern guider-context vector, the decoder state, and the decoder input, indicating the likelihood of generating tokens from the fixed vocabulary. The complement represents the likelihood of copying tokens from the input. The final output probability combines these two probabilities. Attention mechanisms cover both the input method context and pattern guide token sequences. For out-of-vocabulary (OOV) tokens, the generation probability is zero, and the model copies tokens based on the attention distribution, enabling OOV token generation.

• GRU.

Definition. The definition was mentioned earlier, with detailed information discussed in Section 4.3.2—**Context Encoding—Gate Recurrent Unit (GRU)**.

Literature. Li et al. [57] and Qu et al. [84] both use GRU-based decoders in their models. Li et al.’s DeepName integrates a Non-copy mechanism with GRUs and attention to emphasize important sub-tokens, combining CopyNet and Non-copy scores to predict tokens. Qu et al.’s SGMNG also uses GRUs, with a decoding attention mechanism and a pointer network for copying out-of-vocabulary words. The SGMNG decoder combines attention outputs with the hidden state to generate the target sequence.

• Transformer.

Definition. The definition was mentioned earlier, with detailed information discussed in Section 4.3.2–Context Encoding–Transformer.

Literature. Liu et al. [59] develop GTNM, a Transformer-based model that uses three attention layers to predict method names from context vectors and previously generated subtokens. Zhang et al. [136] utilize a Transformer for decoding Heterogeneous Program Graphs, enhanced with a pointer network to manage out-of-vocabulary words by combining probabilities of generating and copying tokens. Similarly, Ahmed et al. [3] and Zhang et al. [135] also build the decoder on the Transformer architecture.

• Simple Decoder.

Definition. A simple decoder in the context of neural networks typically consists of a hidden layer and an output layer. The hidden layer processes the input to generate a representation that is used both for predicting the current token and for providing feedback to the network to assist in predicting the subsequent token.

Literature. In [81], Petukhov et al. implement a simple decoder that consists of a hidden layer and an output layer. The output of the hidden layer is used for the prediction of the current token and as the feedback to the neural network for the prediction of the next token.

• CodeT5 Decoder.

Definition. In CodeT5, the encoder processes and encodes the input code into contextual embeddings, while the decoder uses these embeddings to generate the output sequence, ensuring that the generated output is contextually aligned with the input code. Detailed information about the encoder is discussed in Section 4.3.2–Context Encoding–CodeT5 Encoder.

Literature. As mentioned in Section 4.3.2–① Context Encoding–• Token., Zhu et al. [142] utilize CodeT5 to solve the MNP task. Different from CodeBERT which only has an encoder, CodeT5 consists of an encoder and a decoder. Therefore, Zhu et al. directly adopt the decoder provided in CodeT5 to predict method names.

• Other.

The Fully Connected Layer in a neural network is a layer where each neuron is connected to every neuron in the previous layer, allowing it to learn complex relationships by performing a weighted sum followed by an activation function. It is commonly used in the final stages of the network for making predictions or classifications. The Pointer Network is a neural network architecture that employs attention mechanisms to generate sequences of discrete items or indices from an input sequence. It is particularly suited for combinatorial optimization tasks, as it enables the model to dynamically "point" to specific elements in the input, facilitating the handling of variable-length outputs.


4.4 Discussion

The progress of MNP is closely linked to advancements in ML techniques. The pursuit of better performance is almost the central theme of all MNP-related research papers. To enhance performance, MNP researchers continuously explore and experiment with various approaches. In context extraction, researchers early recognized the complex relationships between method names and code, making it crucial to model the various associations between method names and other code elements. Initially, Alon et al. [10] use local context for MNP. Later, Nguyen et al [69] identify method names that also have associations with global context factors, such as the class name, and performed frequency-based analysis. Subsequently, Liu et al. [59] begin incorporating document information for prediction. As more information was extracted, the performance metrics of MNP improved.

Unlike natural language, code encompasses not only lexical features but also rich syntactic and semantic features. Researchers believe that capturing more of these features can enhance MNP performance. Therefore, for MNP tasks, researchers have employed various methods, including extracting n-gram features, utilizing syntactic structures such as AST and CFG, and applying semantic structures like PDG. The application of these methods has achieved some success.

When selecting encoding-decoding models, researchers tend to explore various innovative network architectures, such as TBCNN and GINN, to effectively capture relevant features and find the most suitable solutions for their research problems. MNP research heavily relies on advancements in learning technologies. In the early stages of DL, methods, and techniques are still evolving and lack mature standards and universal approaches. Consequently, researchers often design network architectures specifically tailored to the characteristics and requirements of different SE tasks.

With the widespread adoption of pre-trained models, an increasing number of SE tasks are beginning to utilize large models, including in the field of MNP. As shown in Fig. 20, Transformer models begin to be introduced to MNP tasks starting in 2021. By 2023, larger pre-trained models based on Transformers, such as CodeT5, also start to be applied, leading to significant performance improvements. The introduction of attention mechanisms indeed enhances performance; early customized MNP networks often require additional integration of attention mechanisms, whereas large Transformer-based models inherently include this mechanism, making them highly suitable for the task. Moreover, these large models typically handle tokens, and the recent success of large language models (LLMs) and the simplicity of token representation make extracting features from code easier. As model size increases, lexical features of code may, to some extent, replace the role of syntactic and semantic features.

 **Summary** ► Existing MNP techniques can be mainly divided into two categories: context-based classification and context-based generation. Key factors influencing the first category of MNP techniques include context extraction, context preprocessing, context encoding, and context classifying. Key factors affecting the second category of MNP techniques include context extraction, context preprocessing, context encoding, and context decoding. From this, it can be seen that the key difference between these two categories of techniques lies in whether MNP is regarded as a classification task or a generation task. ◀

5 ANSWER TO RQ2: WHAT ARE THE METHODS EMPLOYED TO EVALUATE LEARNING-BASED MNP TECHNIQUES?

In this section, we investigate and summarize the key approaches used to evaluate the 51 MNP techniques, including aspects of the datasets, performance metrics, and replication packages.

5.1 Datasets

Different datasets pose different challenges and tasks for MNP. Fig. 22 shows the distribution of programming languages in the dataset. It is observed that Java is the most popular language with 51 studies using Java. After Java, Python is the next most popular language with 7 studies using it. Fig. 23 displays the distribution of the number of study instances (i.e., methods or files) in the dataset. Most researchers explicitly count the number of methods or files in the dataset. Counting the number of methods is the dominant approach (47 studies provide the number of methods in the dataset). Some studies provide details on the number of methods or files used for each stage of model training.

Quantitatively, the number of methods used in MNP is concentrated in the interval of 100k~100m. Among them, there are 16 studies in the range of 100k~1m and 18 studies in the range of 10m~100m.

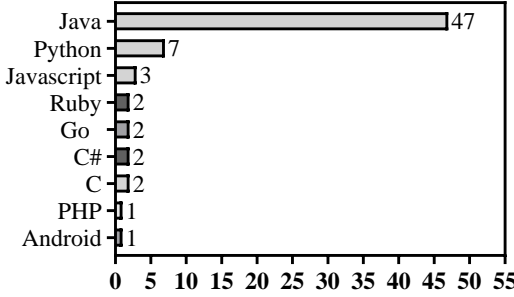


Fig. 22. Programming language

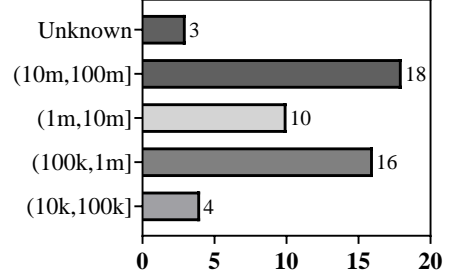


Fig. 23. Data set scale

Fig. 24. Data set language and scale

These two intervals account for 72% of the total. Table 7 shows the dataset size for different study instances. It can be observed that most of the MNP studies have constructed larger datasets ($> 1m$).

Table 7. Dataset scale in different code granularities (i.e., method and file)

Scale	[10k,100k]	[100k,1m]	[1m,10m]	[10m,100m]	Unkown	Total
Method	4	16	9	18	0	47
File	0	0	1	0	3	4
Total	4	16	10	18	3	51

Table 8 overviews the different data sources. It is divided into several different categories: mainly from GitHub [10], Apache [115], Stanford Open Graph Benchmark (OGB) [62], CodeXGLUE[3], Google [115], and CodeSearchNet [136], etc. Among them, GitHub is the most popular, with 40 MNP papers using methods collected from GitHub, from which they mainly select some of the top-ranked and most popular projects. Additionally, Java-large, Java-small, and Java-med are used in many studies and are representative to a certain extent.

5.2 Performance Metrics

In this Section, we will provide an overview of the metrics used in the MNP works and clarify the relevant definitions. Additionally, we will illustrate the differences between various metrics with concrete examples.

5.2.1 Overview of Metrics. Existing MNP studies employ various performance evaluation metrics as listed in Table 9 to measure the effectiveness of an MNP technique. In terms of performance evaluation, most of the studies mainly measure precision, recall, and F1-score, and some studies creatively use exact matching accuracy and so on.

- **Precision (P).** P refers to the proportion of correctly predicted method name tokens from the total number of predicted method name tokens. It measures the accuracy of the MNP technique.

- **Recall (R).** R is the proportion of correctly predicted method name tokens to the total number of tokens in the target method name. It measures the recall ability of the MNP technique.

- **F1-score (F1).** The F1 is the harmonic mean of precision and recall and provides a comprehensive measure of system performance. A high F1-score indicates a good balance between precision and recall.

Table 8. A summary overview of the different data sources

Year	Name	URL	Source	Number
2019	Java-small	https://github.com/tech-srl/code2vec http://github.com/tech-srl/code2seq	GitHub	[10, 12, 14, 64, 96, 119] [57, 59, 84, 118, 136] [30, 58, 82, 111, 142] [15, 33, 122]
2019	Java-med		GitHub	[10, 12, 47, 115] [15, 57, 76, 118, 142]
2019	Java-large		GitHub	[10, 12, 21, 86, 115] [13, 15, 30, 57, 96, 118, 142]
2019	codeattention	https://groups.inf.ed.ac.uk/cup/codeattention/	GitHub	[132]
2019	CodeSum2	https://github.com/XuSihan/CodeSum2	GitHub	[127]
2019	CoderPat	https://github.com/CoderPat/structured-neural-summarization	GitHub	[29]
2019		https://drive.google.com/file/d/1fNSmPluXbhQ9cfcAHkTtNrHID8Jkh2XD	GitHub	[32]
2020	Mnire	https://sonvnguyen.github.io/mnire/#datasets	GitHub	[57, 59, 69, 118, 131, 142]
2020	Flow2vec	https://github.com/artifact4oopsla20/Flow2Vec?tab=readme-ov-file#21-code-classification-and-summarization-table-2	GitHub	[98]
2021	OGB	https://ogb.stanford.edu/	Stanford OGB	[62]
2021	Meth2Seq	https://meth2seq.github.io/meth2seq/	GitHub	[135]
2021	Mocktail Dataset	https://archive.org/download/mocktail-dataset-method-naming-tse	GitHub	[109]
2021	Python Summary Dataset	https://github.com/EdinburghNLP/code-docstring-corpus	GitHub	[109]
2022	CodeSearchNet	https://github.com/github/CodeSearchNet	GitHub	[78, 136, 143]
2022	Java2Graph	https://github.com/kk-arman/graph_names/	GitHub	[81]
2022	CodeXGLUE	https://microsoft.github.io/CodeXGLUE/	GitHub	[3]
2023	JavaRepos	https://github.com/TruX-DTF/debug-method-name	Apache, Spring, Hibernate, and Google	[60, 117]
2023	150k Python Dataset	https://www.sri.inf.ethz.ch/py150	GitHub	[82]

Table 9. Performance metrics used by MNP studies

Metric	Range	Study	Number
P R F_1	[0, 1]	[10, 12, 14, 21, 47, 69, 86, 109, 115, 119, 127] [57, 59, 62, 64, 81, 96, 118, 131, 135, 136] [3, 4, 13, 15, 35, 58, 76, 82, 85, 117, 122, 142] [7, 29, 30, 60, 78, 98, 105, 111, 134, 143]	43
EM	0,1	[7, 11, 32, 57, 59, 69, 111, 127, 131, 134, 135, 142]	12
ROUGE	[0,1]	[29, 33, 84, 119, 123]	5
BLEU	[0,1]	[32, 123]	2
ED	[0, +∞)	[32, 111]	2
Accuracy	[0,1]	[30, 35, 60]	3
AED RED	[0, +∞)	[35]	1
METEOR	[0,1]	[123]	1

In the evaluation process, the method name ground-truth (N_g) and the predicted method name (N_p) are treated as a pair, and the following calculations are performed:

$$P = \frac{|token(N_g) \cap token(N_p)|}{|token(N_p)|} \quad (1)$$

$$R = \frac{|token(N_g) \cap token(N_p)|}{|token(N_g)|} \quad (2)$$

$$F1 = 2 \times \frac{P \times R}{P + R} \quad (3)$$

where $token(\cdot)$ is a function that returns the (sub)tokens in the method name \cdot , and the absolute value symbol $|\cdot|$ returns the number of tokens. Similar to F1, the Modified-F1 (F1**) score is also used to evaluate method names. The Modified-F1 (F1**) score is calculated from the modified unigram precision and unigram recall. This prevents models that repeatedly output subwords in the gold function name from receiving unreasonably high scores.

• **Exact match (EM).** The EM is whether the model's predictions exactly match the true results. For tasks such as sequence annotation, each annotation is required to be correct to be considered a match. Specifically, if the method name predicts N_p matches the method name ground-truth N_g exactly, the exact matching degree is 1, otherwise 0.

$$EM = \begin{cases} 1, & \text{if } N_g == N_p \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

• **Recall-Oriented Understudy for Gisting Evaluation (ROGUE).** ROGUE is a set of metrics for evaluating text summarization systems. ROGUE scores typically include variants such as ROGUE-N and ROGUE-L, where N is the size of the n-gram (n consecutive words), and L is the size of the Longest Common Subsequence(LCS). ROGUE-N evaluates the performance of a model by comparing the similarity of the generated summaries to a reference summary.

$$ROGUE - N = \frac{|token_n(N_g) \cap token_n(N_p)|}{|token_n(N_g)|} \quad (5)$$

where n stands for the length of the n-gram, n -gram and $|token_n(N_g) \cap token_n(N_p)|$ is the maximum number of n-grams co-occurring in a candidate summary and a set of reference summaries.

ROUGE-L ROUGE-N focuses on the overlap of n-grams to measure the similarity between generated text and reference text at the phrase or n-gram level, while ROUGE-L emphasizes the Longest Common Subsequence (LCS) to evaluate the coherence and order of the text. Its calculation formula is as follows:

$$R_{lcs} = \frac{LCS(N_g, N_p)}{|token(N_g)|} \quad (6)$$

$$P_{lcs} = \frac{LCS(N_g, N_p)}{|token(N_p)|} \quad (7)$$

$$F_{lcs} = \frac{(1 + \beta^2)R_{lcs}P_{lcs}}{R_{lcs} + \beta^2P_{lcs}} \quad (8)$$

where $LCS(N_g, N_p)$ is the length of the longest common subsequence of N_g and N_p . R_{lcs} and P_{lcs} denote recall and precision. The last F_{lcs} is Rouge-L. β is a parameter that adjusts the relative importance of precision and recall. If β is set to 1, the F_{lcs} represents the harmonic mean of precision and recall, and its calculation is the same as that of the F1-score.

• **Bilingual Evaluation Understudy (BLEU).** BLEU is a commonly used metric in machine translation tasks to measure the accuracy of translated candidate sentences against reference sentences. It compares how many words in the target sequence appear in the decoded sequence. In

the MNP task, the predicted method names and the actual method names are treated as candidate sentences and reference sentences, respectively. The score is calculated as follows:

$$BLEU = BP \cdot \exp \left(\sum_{n=1}^N w_n \log(N_p(n)) \right) \quad (9)$$

where $N_p(n)$ is the modified n-gram precision calculated by dividing the clipped count of n-grams in the candidate sentence by the count of n-grams in the reference sentence.

To reduce the bias for short candidate sentences, a brevity penalty is introduced:

$$BP = \begin{cases} 1 & \text{if } |N_p| > |N_g| \\ e^{(1-|N_g|/|N_p|)^r} & \text{if } |N_p| \leq |N_g| \end{cases} \quad (10)$$

where r is the length of the reference sequence and c is the length of the candidate sequence. It calculates the 1-gram, 2-gram, 3-gram, and 4-gram matching accuracy between candidate and reference sequences.

• **Edit Distance (ED).** ED is a commonly used metric for measuring the difference between two strings, calculating the minimum number of operations required to transform the generated string into the target string. In the MNP task, ED is used to compare the differences between the predicted and ground truth method names, counting the necessary insertions, deletions, and single-character substitutions.

• **Metric for Evaluation of Translation with Explicit ORDERING (METEOR).** METEOR is a commonly used metric for evaluating machine translation quality. It measures translation accuracy and completeness by combining the harmonic mean of precision and recall. METEOR not only focuses on word-level matches but also includes phrase matching and accounts for variations in word forms (such as stemming and lemmatization) and word order issues, providing a more comprehensive evaluation of translation quality compared to traditional metrics like BLEU. In the MNP task, the predicted method names and ground truth names correspond to the translated output and reference answers, respectively.

$$F_{mean} = \frac{10 \cdot P \cdot R}{R + 9 \cdot P} \quad (11)$$

$$Penalty = 0.5 \left(\frac{\#chunks}{\#unigrams_matched} \right)^3 \quad (12)$$

$$METEOR_Score = F_{mean} \cdot (1 - Penalty) \quad (13)$$

where *unigram* refers to the smallest unit of a single word and typically does not require special configuration. P and R are computed based on unigram precision and recall. METEOR first calculates the harmonic mean F_{mean} of precision and recall. It then adjusts this value based on the number of matched *unigrams* and *chunks*, resulting in a final score that reflects both precision and recall. In practical applications, the length of *chunks* is usually set between 1 and 4 words.

• **Accuracy.** Accuracy is a metric used to assess the performance of classification models, indicating the proportion of correct predictions among all predictions made. In the MNP task, accuracy measures the proportion of generated method names that are identical to the ground truth method names. It is calculated by dividing the number of correctly generated method names by the total number of test instances.

$$Accuracy = \frac{\text{Number of EM Predictions}}{\text{Total Number of Predictions}} \quad (14)$$

• **Average Edit Distance (AED) and Relative Edit Distance (RED).** AED is a metric for measuring edit operations, used to calculate the average number of edits required to transform the generated name into the target name. RED is used in method name consistency detection scenarios, and, compared to absolute edit distance, RED better reflects the relative differences and improvements between the newly generated name and the original name.

Among the metrics mentioned earlier, AED and RED are similar to the Accuracy metric in that they are calculated over the entire test set. In contrast, other metrics are computed for individual method names, with the dataset-level metrics obtained by averaging the results for each method name. Specifically, for a given test set with N samples, the AED and RED of an approach are defined as follows:

$$AED = \frac{1}{N} \sum_{k=1}^N ED(N_p^{(k)}, N_g^{(k)}) \quad (15)$$

$$RED = \frac{1}{N} \sum_{k=1}^N \frac{ED(N_p^{(k)}, N_g^{(k)})}{ED(N_{orig}^{(k)}, N_g^{(k)})} \quad (16)$$

where $ED(\cdot)$ is the edit distance function, N_p is the generated method name, N_{orig} is the original method name that has not been modified before consistency checking, and N_g is the ground truth method name.

5.2.2 Metrics Examples. To clarify the metrics, providing examples is very helpful. We will present examples to visually demonstrate the calculation process and practical application of each metric, thereby aiding in the understanding of their actual significance and use.

Assuming the dataset contains two records: for the first record, the ground truth method name $N_g^{(1)}$ is "calculateArraySum" and our predicted method name $N_p^{(1)}$ is "calculateSum"; for the second record, the ground truth method name $N_g^{(2)}$ is "calculateTotalSumforArrayElements" and our predicted method name $N_p^{(2)}$ is "calculateSumforArrayElements." We evaluate the performance of the predictions by calculating various metrics.

To calculate the metrics, method names need to be split into individual tokens and converted to lowercase. For names using snake case, underscores should be removed. Table 10 presents some important intermediate results in the calculation of the metrics. For example, from the table, we see that $|token(N_g^{(1)})| = 3$, $|token(N_p^{(1)})| = 2$, and $|token(N_p^{(1)}) \cap token(N_g^{(1)})| = 2$. Using Formulas 1 and 2, precision $P = 1$ and recall $R = 0.66$ can be easily calculated, leading to an F1-score of 0.8.

In other examples, such as "calculateSum" and "calculateArraySum", the names are not identical, resulting in an EM score of 0. The method name "calculateSum" requires the insertion of 5 characters, "Array" to transform into the target name "calculateArraySum", so the ED between them is 5. Similarly, the edit distance between "calculateSumforArrayElements" and "calculateTotalSumforArrayElements" is also 5, so the AED of these two records is 5. In [35], RED is used for checking name consistency. Assuming the original name is "calculate", the edit distance between "calculate" and "calculateArraySum" is 5, while the edit distance between "calculate" and "calculateSumforArrayElements" is 24, resulting in a RED value of approximately 0.42. We present the calculated results of the examples in Table 11. To provide a clear visual representation of each metric, we have listed the value ranges for each metric in the first row to facilitate intuitive comparison.

5.3 Replication Packages

In this section, we explore how often the 51 reviewed MNP techniques share replication packages in their papers. Replicability refers to the ability of other researchers to obtain the same results using

Table 10. Metric Calculation Details for Examples

<i>Record</i> ₁	<i>Record</i> ₂
$N_g^{(1)} = \text{"calculateArraySum"}$	$N_g^{(2)} = \text{"calculateTotalSumforArrayElements"}$
$N_p^{(1)} = \text{"calculateSum"}$	$N_p^{(2)} = \text{"calculateSumforArrayElements"}$
$\text{token}(N_g^{(1)}) = \{\text{'calculate'}, \text{'Array'}, \text{'Sum'}\}$	$\text{token}(N_g^{(2)}) = \{\text{'calculate'}, \text{'total'}, \text{'sum'}, \text{'for'}, \text{'array'}, \text{'elements'}\}$
$ \text{token}(N_g^{(1)}) = 3$	$ \text{token}(N_g^{(2)}) = 6$
$\text{token}(N_p^{(1)}) = \{\text{'calculate'}, \text{'sum'}\}$	$\text{token}(N_p^{(2)}) = \{\text{'calculate'}, \text{'Sum'}, \text{'for'}, \text{'array'}, \text{'elements'}\}$
$ \text{token}(N_p^{(1)}) = 2$	$ \text{token}(N_p^{(2)}) = 5$
$\text{token}(N_p^{(1)}) \cap \text{token}(N_g^{(1)}) = \{\text{'calculate'}, \text{'sum'}\}$	$\text{token}(N_p^{(2)}) \cap \text{token}(N_g^{(2)}) = \{\text{'calculate'}, \text{'sum'}, \text{'for'}, \text{'array'}, \text{'elements'}\}$
$ \text{token}(N_p^{(1)}) \cap \text{token}(N_g^{(1)}) = 2$	$ \text{token}(N_p^{(2)}) \cap \text{token}(N_g^{(2)}) = 5$
$\text{token}_2(N_g^{(1)}) = \{\text{'calculate array'}, \text{'array sum'}\}$	$\text{token}_2(N_g^{(2)}) = \{\text{'calculate total'}, \text{'total sum'}, \text{'sum for'}, \text{'for array'}, \text{'array elements'}\}$
$ \text{token}_2(N_g^{(1)}) = 2$	$ \text{token}_2(N_g^{(2)}) = 5$
$\text{token}_2(N_p^{(1)}) = \{\text{'calculate sum'}\}$	$\text{token}_2(N_p^{(2)}) = \{\text{'calculate sum'}, \text{'sum for'}, \text{'for array'}, \text{'array elements'}\}$
$ \text{token}_2(N_p^{(1)}) = 1$	$ \text{token}_2(N_p^{(2)}) = 4$
$\text{LCS}(N_p^{(1)}, N_g^{(1)}) = \text{'calculate sum'}$	$\text{LCS}(N_p^{(2)}, N_g^{(2)}) = \text{'calculate sum for array elements'}$
$ \text{LCS}(N_p^{(1)}, N_g^{(1)}) = 2$	$ \text{LCS}(N_p^{(2)}, N_g^{(2)}) = 5$
$ N_g^{(1)} = 2$	$ N_g^{(2)} = 6$

Table 11. Metric Values for Two Example

Metric	P	R	F1	EM	ROUGE-L	BLEU	ED	Accuracy	AED	RED	METEOR
Range	[0,1]	[0,1]	[0,1]	0,1	[0,1]	[0,1]	[0, +∞)	[0,1]	[0, +∞)	[0, +∞)	[0,1]
<i>Record</i> ₁	1	0.66	0.80	0	0.80	0.61	5	0	5	0.42	0.34
<i>Record</i> ₂	1	0.83	0.91	0	0.91	0.58	5	0	5	0.42	0.82

the artifacts provided by the authors. Reproducibility, however, involves obtaining the same results using generally the same methods, but not necessarily the original artifacts. Both replicability and reproducibility are crucial for assessing the quality of original research and validating its credibility, as they enhance our confidence in the results and help us distinguish between reliable and unreliable findings. This is particularly important for learning-based methods, whose performance often heavily depends on factors such as datasets, data processing methods, and hyperparameter settings.

As mentioned in Section 3.5, we consider replicability and reproducibility as significant factors in evaluating research quality. We begin by systematically reviewing each paper to ensure that the provided code links are publicly accessible and valid, and we record the availability of each link. Next, we verify whether the datasets used in the papers are sourced from open repositories; if they are private or require special permissions, we mark these as exceptions. We then assess whether the papers include all necessary components and environments for replication, such as experimental setups, dependencies, and configuration files. Additionally, we evaluate whether the papers provide sufficient details to support replication, including experimental methods, parameter settings, and data processing steps. Finally, we compile all this information, calculate the compliance of each paper with the standards, and summarize the results. Based on the statistical and summary results, we have created Fig. 27 to present the data visually. To facilitate future MNP studies, we provide the usable replication packages in Table 12.

As shown in Fig. 25, 11 papers, accounting for approximately 22%, did not provide links to the research results. Among the 40 papers that did provide links, the links to 4 papers are no longer

Table 12. Replication package links provided in MNP studies

Year	Technique	URL	Study
2015	-	https://groups.inf.ed.ac.uk/cup/naturalize/#team	[4]
2016	-	https://groups.inf.ed.ac.uk/cup/codeattention/	[7]
2019	-	https://github.com/SerVal-DTF/debug-method-name	[60]
2019	-	https://github.com/CoderPat/structured-neural-summarization	[29]
2019	Code2vec	https://github.com/tech-srl/code2vec	[12, 115]
		https://github.com/basedrhys/obfuscated-code2vec	[21]
		https://github.com/mdrafiqulrabin/handcrafted-embeddings	[86]
		https://cazzola.di.unimi.it/fold2vec.html	[58]
		https://github.com/NobleMathews/mocktail-blend	[109]
2019	Code2seq	https://github.com/tech-srl/code2seq	[10]
2019	Mercem	https://groups.inf.ed.ac.uk/cup/codeattention	[132]
2019	HeMA	https://github.com/Method-Name-Recommendation/HeMa	[47]
2019	HAN	https://github.com/XuSiHan/CodeSum2	[127]
2020	GINN	https://github.com/GINN-Imp/GINN	[119]
2020	Mnire	https://doubledoubleblind.github.io/mnire	[69]
2020	Flow2vec	https://github.com/artifact4oops/la20/Flow2Vec?tab=readme-ov-file#21-code-classification-and-summarization-table-2	[98]
2021	TPTrans	https://github.com/AwdHanPeng/TPTrans	[78]
2021	InferCode	https://github.com/bdqngghi/infercode	[14]
2021	-	https://github.com/css518/Keywords-Guided-Method-Name-Generation	[33]
2021	-	www.daml.in.tum.de/code-transformer	[143]
2021	-	https://github.com/mdrafiqulrabin/tnpa-generalizability	[85]
2021	TreeCaps	https://github.com/bdqngghi/treecaps	[15]
2021	PSIMiner	https://github.com/JetBrains-Research/code2seq	[96]
2021	Meth2Seq	https://meth2seq.github.io/meth2seq/	[135]
2021	UniCoRN	https://github.com/dmlc/dgl/tree/master/examples/pytorch/rgcn-hetero	[62]
2021	Cognac	https://github.com/ShangwenWang/Cognac	[118]
2021	DeepName	https://github.com/deepname2021icse/DeepName-2021-ICSE	[57]
2022	GraphCode2Vec	https://github.com/graphcode2vec/graphcode2vec	[64]
2022	HGT	https://github.com/IBM/Project_CodeNet/issues/29	[136]
2022	GNNs	https://github.com/kk-aman/graph_names	[81]
2022	NamPat	https://github.com/cqu-isse/NamPat	[131]
2022	SGMNG	https://github.com/QZH-eng/SGMNG	[84]
2022	CodeBERT	https://github.com/microsoft/CodeBERT/tree/master/GraphCodeBERT/codesearch	[3]
2022	GTNM	https://github.com/LiuFang816/GTNM	[59]
2022	PCAN	https://github.com/HongliangLiang/pcan	[122]
2023	SENSA	https://github.com/mzakeri/SENSA	[76]
2023	Fold2Vec	https://cazzola.di.unimi.it/fold2vec.html	[13]
2023	Mario	https://github.com/ShangwenWang/Mario	[117]
2023	AUMENA	https://figshare.com/s/0382ba979d970b4c2b2	[142]
2023	-	https://github.com/software-theorem/mnp	[82]

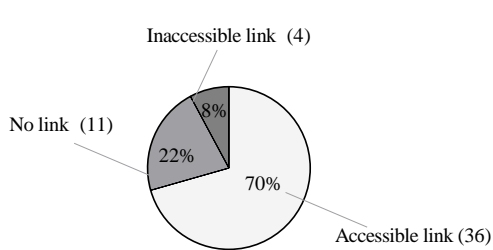


Fig. 25. Accessible of papers

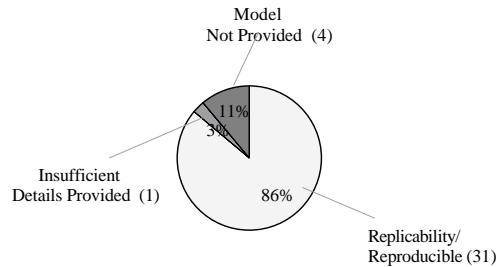



Fig. 26. Replicability of papers

Fig. 27. Accessible and Replicability filtering process

valid, representing about 8%. The remaining 36 papers offered valid links, and we further analyzed

their replicability. As depicted in Fig. 26, all 36 papers explicitly provided specific datasets. However, 4 papers did not provide model files, and 1 paper lacked sufficient details, corresponding to approximately 11% and 3%, respectively. The remaining 31 papers also included detailed documentation.

 **Summary** ► The databases predicted by the method names are built with large-scale source code written in Java from public code repositories (e.g., GitHub). The more used indicators of judgment in MNP are Precision, Recall, and F1-score. 31 papers, comprising 51% of the total, offer replication packages to facilitate result reproducibility. ◀

6 ANSWER TO RQ3: WHAT DO EXISTING EMPIRICAL STUDIES ON LEARNING-BASED MNP CONCERN?

Although MNP is an emerging research area in SE, a variety of learning-based MNP techniques have been successively proposed and continuously achieved promising results in terms of the accuracy of predicted names over the past six years. In addition to developing new MNP techniques that address technical challenges, the learning-based MNP research field is benefiting from several empirical studies. These empirical studies systematically explore the impact of various components (e.g., AST representation), providing insights into future learning-based MNP work. We summarize existing empirical studies in Table 13 and discuss them in detail as follows.

6.1 Generalization Performance of Models

Jiang et al. [47] conduct an empirical study on code2vec, exploring its performance across various datasets and investigating the underlying reasons for both its successes and failures. Specifically, they investigate the following research questions: 1) How well does code2vec work on datasets other than the one employed by the original evaluation conducted by the authors of code2vec? They evaluate code2vec on a new dataset as well as its original dataset employed in paper [12]. The results show that code2vec overall is accurate, and switching to a new large-scale dataset does not result in a significant reduction in the performance of code2vec. 2) How well does code2vec work with more realistic settings? They evaluate code2vec on a new dataset with different settings, i.e., file-based validation, project-based validation, and project-based non-overriding validation. The results demonstrate that the performance of code2vec decreases significantly in more realistic settings. However, its overall performance is still promising. 3) Can code2vec generate method names correctly when the given method bodies do not contain method name tokens? They evaluate the performance of code2vec in generating seen and unseen method name tokens, respectively. For each of the method names in the new dataset, we split it into tokens according to the Camel-Case naming convention. A token t from method name mn is a seen token if t appears (case insensitive) in the body named by mn . Otherwise, it is an unseen token. They assess how often code2vec can successfully recommend seen/unseen method name tokens during project-based non-overriding validation. They conclude that code2vec works well in generating unseen method name tokens. 4) Where and why does code2vec work? They conduct another manual analysis on successful cases. They observe that code2vec works well on getter/setter methods and delegations. The rationale for the success is that such methods have common structures in their ASTs. 5) Where and why does code2vec fail? They conduct another manual analysis on one thousand failed cases. They identify the main reasons for the failure as follows: First, out-of-vocabulary method names are the primary for the failure. A method name is out of vocabulary if and only if none of the methods in the training set is named with it. As a result, code2vec often fails. Using open vocabularies or minting method names by combining tokens may help to further improve performance. The second reason is that the exploited method features are often insufficient to infer method names. To further

Table 13. A summary and comparison of empirical studies in learning-based MNP

Year	Study	Language	Main Research Content (Research Questions)
2019	Jiang et al. [47]	Java	1) How well does code2vec work on datasets other than the one employed by the original evaluation conducted by the authors of code2vec? 2) How well does code2vec work with more realistic settings? 3) Can code2vec generate method names correctly when the given method bodies do not contain method name tokens? 4) Where and why does code2vec work? 5) Where and why does code2vec fail? 6) Is code2vec useful for developers? How often does code2vec recommend correctly for methods that are challenging to name manually?
2020	Rabin et al. [86]	Java	1)How does the performance of SVM models built using code2vec embeddings compare to SVM models trained on handcrafted features for MNP?
2020	Zaitsev et al. [134]	Pharo	1)Which model performs better in the MNP task, the extractive model based on TF-IDF with n-grams or the abstractive model based on sequence-to-sequence neural networks?
2021	Rabin et al. [85]	Java	1) How do the transformations impact the predictions of neural program models in the single-place transformed dataset? 2) When do the transformations affect neural program models the most? 3) How does the method length impact the generalizability of neural program models? 4) What are the trends in types of changes? 5) How do the transformations affect the precision, recall and F_1 -score of the neural program models?
2022	Varner et al. [111]	Java	1)Which model performs better in the MNP, the RNN encoder-decoder model with attention or the Transformer model?
2023	Wang et al. [117]	Java	1) How pervasive are proximate class pairs? 2) How is the relation between the field-irrelevant method names of a class and those of its proximate classes? 3) How is the similarity between field-relevant method names of a class and those of its proximate classes with respect to its non-unique fields? 4) How is the effectiveness of Mario? 5) How is the effectiveness of Mario on field-relevant method names and field-irrelevant method names respectively?
2023	Li et al. [58]	Java	1)How ASTs generated from different parsing tools could affect the performance of machine learning models depending on an input of code structure representation.
2023	Qian et al. [82]	Java, Python	1) Does AST improve the performance of MNP? 2) How do AST parsing methods affect the performance of MNP? 3) How do AST preprocessing methods affect the performance of MNP? 4) How do AST encoding methods affect the performance of MNP?

improve performance, additional information should be utilized, e.g., their enclosing classes. The third reason is improper method names in the testing set. During the evaluation, all recommended names are compared against the original names associated with testing methods, i.e., the code2vec fails if the recommended name is different from the original one.

Rabin et al. [86] conduct an empirical study to have a better understanding of the contents of code2vec neural source code embeddings. They use code2vec embeddings to create binary SVM classifiers and compare their performance with the handcrafted features on the MNP task. The results of a small case study suggest that the handcrafted features can perform very close to the highly-dimensional code2vec embeddings, and the information gains are more evenly distributed in the code2vec embeddings compared to the handcrafted features. They also find that the code2vec embeddings are more resilient to the removal of dimensions with low information gains than the handcrafted features.

In [85], Rabin et al. explore how semantic-preserving transformations affect the generalizability of neural program models. They assess the performance of code2vec, code2seq, and GGNN on three Java datasets, evaluating model predictions before and after transformations. The study reveals that GGNN is more sensitive to changes caused by semantic-preserving transformations compared to code2vec and code2seq. The latter models exhibit less variation in prediction outcomes with increased dataset sizes. Additionally, all-place transformations tend to affect code2vec and code2seq more than single-place transformations, while GGNN remains relatively stable. The study also finds that code2vec and code2seq perform better with longer methods, whereas GGNN's performance is less influenced by method length. Transformations generally reduce model performance, with code2vec and code2seq being more affected but compensating with larger datasets. Although semantic-preserving transformations impact classic metrics such as precision, recall, and F1-score, these changes do not correlate with the new metric, Prediction Change Percentage (PCP).

6.2 Role and Impact of Components

AST. Li et al. [58] investigate the source code engineering impacts (mainly AST parser) towards machine learning-based software services (mainly MNP task). They evaluate AST parser generators towards the impacts on the prediction model of code2vec for the prediction task of the method name in Java language. Like Utkin et al. [108], they also utilize five metrics: TS, TD, BF, UTP, and UTK, to compare the ASTs generated by five AST parsers: JavaParser, ANTLR, Tree-sitter, Guntree, and Javalang. Their experimental results on the Java-small dataset show that ASTs generated using different parsing parsers differ greatly in terms of source code structures and contents. This difference could significantly influence the performance of the trained model code2vec.

Qian et al. [82] also believe that it is a complex process to manipulate AST, including AST parsing, AST preprocessing, and AST encoding, of which a change in the scheme may change the AST embeddings and thus affect the performance of MNP. Therefore, they aim to dive deep into the problem and answer: *Does AST help to improve MNP? How do AST parsing/preprocessing/encoding methods affect MNP?* To achieve this, they conduct a comprehensive empirical study to systematically investigate the impact of the sub-processes of AST usage on MNP performance. Specifically, they first depict the workflow of the AST processing and usage in MNP. The utilization of AST involves intricacies encompassing three crucial sub-processes: AST parsing, AST preprocessing, and AST encoding. Then, they carry out experiments on two popular programming datasets, i.e., Java-small [4] and 150k Python Dataset [88]. The experiments involving four AST parsing methods: JDT, Javaparser, Javalang, and Tree-sitter; three AST preprocessing methods: original AST, AST path [11], and program graph [6]; and five AST encoding methods: BiLSTM, code2vec, code2seq, Tree-LSTM, and GGNN [6]. Finally, they obtain the following major findings about the current AST for MNP: 1) AST can indeed improve MNP; The strategic combination of path information within AST and token information from the code can notably influence MNP performance. 2) The selection of AST parsing methods bears a considerable influence on the overall performance of MNP. Currently, JDT stands out as the superior choice, outperforming Javaparser, Javalang, and Tree-sitter in promoting MNP. 3) AST preprocessing plays a pivotal role in the usage of AST, and a well-designed preprocessing scheme significantly influences MNP's efficacy. Currently, AST Path emerges as a particularly promising choice for MNP. 4) AST encoding undoubtedly warrants greater attention as it plays a crucial role in effectively translating ASTs into meaningful representations for MNP. Different AST preprocessing results necessitate distinct AST encoding networks or models to achieve optimal outcomes. Currently, code2seq performs best in promoting MNP for AST Path.

Model. In [111], Varner et al evaluate the Transformer and RNN model architectures for three distinct input types. Subsequently, the performance is evaluated across the six distinct experiments. The first input types employ solely the documentation tokens. The second input types utilize

the enclosing class tokens, input parameter tokens, return type tokens, and body tokens, all concatenated together as a single input. The third input type is concatenating the input sequences employed in the initial experiment with the corresponding input sequences utilized in the second experiment. It is anticipated that the third experiment will yield the most favorable results. All of the contexts employed in the input were combined using the period token. The results demonstrate that a model utilizing all of the aforementioned contexts will exhibit superior performance compared to a model employing any subset of the contexts. Moreover, the study demonstrates that the Transformer model outperforms the RNN model in this scenario.

Zaitsev et al. [134] explore two approaches to take the MNP task: one based on TF-IDF [87] and the other using a deep recurrent neural network [104]. The first approach combines TF-IDF with an n-gram language model to perform extractive text summarization. This method identifies keywords from the method's source code and arranges them in a meaningful sequence. The second approach employs an attention-based sequence-to-sequence neural network for abstractive summarization, generating method names from words that do not appear in the source code. In this context, the method name serves as the generated summary, while the method body is the original text. The effectiveness of these approaches is assessed by comparing the generated method names to the actual names assigned by programmers, which are used as the benchmark.

6.3 Performance in MNP Practice

Jiang et al. [47] investigate how often code2vec works when it is strongly needed, specifically assessing its practical utility for developers. The investigation is conducted as follows: First, they invite six developers involved in a commercial project for the evaluation. The commercial project has been released recently by a giant of IT industry to conduct large-scale software refactorings. Second, each of the participants randomly selected one hundred methods developed by himself/herself. Third, they request all participants to score the difficulty in naming sampled methods. The scores rank between one and five (i.e. 5-point scale [52, 63]) where one represents the least difficulty and five represents the highest difficulty. Fourth, they apply code2vec of project-based non-overriding validation to the scored methods and validate the recommendations against manually constructed names. They find that code2vec rarely works when it is strongly needed. Consider code2vec requires a large number of high-quality databases, while the computational process consumes time and resources. To further investigate the possibility of designing simple alternative approaches, they propose a heuristics-based approach to recommending method names according to given method bodies. And their evaluation results show that it outperforms code2vec significantly, and improves precision and recall by 65.25% and 22.45%, respectively.

Wang et al. [117] find that existing literature approaches assume the availability of method implementation to infer its name. However, methods are named before their implementations. Therefore, they first introduce a new MNP scenario where the developer requires an MNP tool to predict names for all methods that are likely to be implemented within a newly defined class. They observe that for classes whose names are similar, the names of their methods, no matter whether they are related to the fields, may be similar to some extent. To validate this observation, they conduct a large-scale empirical analysis on 258K+ classes from real-world projects to validate their findings. To perform the empirical analysis, they analyze 258,321 classes and investigate three research questions: 1) How pervasive are proximate class pairs? 2) How is the relation between the field-irrelevant method names of a class and those of its proximate classes? 3) How is the similarity between field-relevant method names of a class and those of its proximate classes with respect to its non-unique fields? They conclude the main findings obtained through the empirical analysis as follows: [F1] Proximate class pairs are pervasive among real-world projects. [F2] A considerable percentage of tokens composing the names of a class's field-irrelevant methods can be found in

the names of its proximate classes' field-irrelevant methods. [F3] The similarity between a class's method names of its non-unique fields and those of its proximate classes is extremely high. Then, supported by the empirical findings, they propose a hybrid big code-driven approach, namely Mario, to predict method names based on the class name. Experiment results demonstrate that Mario is effective in predicting method names and offers comparable or higher performance to code2seq that leverages implementation details.

 **Summary** ► Existing empirical studies on MNP primarily focus on the following aspects:

- 1) **Generalization Performance of Models.** Code2vec performs well on large-scale datasets, with embeddings showing similar performance to handcrafted features and greater resilience;
- 2) **Role and Impact of Components.** AST parsing and encoding methods are crucial for MNP performance, with the JDT parser and AST Path preprocessing showing the best results, while the Transformer model outperforms RNNs;
- 3) **Performance in MNP Practice.** The practical effectiveness of code2vec is limited. ◀

7 CHALLENGE AND OPPORTUNITIES

7.1 Challenges

7.1.1 Accurate understanding of the target method. For MNP models, whether they are classification or generation models, an accurate understanding of the target method is the prerequisite for generating accurate method names. As mentioned in Section 5, given a target method along with available supplementary information, MNP techniques typically design four processes/components to predict an suitable and accurate name for it. In the four processes, the first three aim to facilitate an understanding of the target method. It is evident that accurately understanding the target method is an extremely challenging task. The answers to **RQ2** demonstrate that existing MNP studies has designed a diverse range of solutions for the first three processes. Hence, the challenges of accurate understanding of the target method can be further attributed to the difficulties in accurate context extraction, preprocessing, and encoding.

Accurate Context Extraction. In Section 4.1, we meticulously sort out the various context information that existing research has extracted and utilized to enhance MNP accuracy, including local context (e.g., Token and AST), global context (e.g., method call graph), and documentation context (e.g., method comment). However, there are still many challenges in adequately extracting context to achieve optimal MNP performance. First, not every known context's real contribution to MNP has been thoroughly validated and explained. Second, when multiple contexts are used simultaneously, the extent to which these contexts collectively enhance/inhibit MNP is also unknown. Finally, whether there are other useful context information yet to be discovered remains unknown.

Accurate Context Preprocessing. Similarly, in Section 4.2, we present various context preprocessing techniques employed in existing MNP research. We salute the researchers for their dedicated explorations into preprocessing for different contexts. These preprocessing techniques often have the purpose of simplification, denoising, and extraction of key information. However, how to accurately preprocess context to achieve these purposes while ensuring MNP performance still faces many challenges. First of all, currently, only a subset of preprocessing techniques for AST has received attention and validation, while preprocessing techniques for other contexts have not all been thoroughly validated and explained in terms of their real contributions to MNP. Secondly, many preprocessing techniques are extremely complex, and replicating them can be extremely challenging if the original implementation code is not publicly available. Lastly, it is unknown whether there are other useful preprocessing techniques for different contexts waiting to be discovered.

Accurate Context Encoding. Context encoding utilizes various neural network-based encoders to convert the context or corresponding preprocessing results into embeddings. It is evident that neural networks play a crucial role in this process. As shown in Section 4.3.1–**Context Encoding** and 4.3.2–**Context Encoding**, existing MNP research has introduced or proposed various neural network-based encoders. Some encoders are specifically designed for the results of specific context preprocessing (e.g., HAN and HGT), while others directly leverage some general neural network models (e.g., RNN and Bi-LSTM). However, designing/selecting an appropriate neural network architecture to build an encoder faces many challenges. Firstly, some studies proposing dedicated neural network architectures do not discuss the feasibility of directly transferring general neural network architectures. In other words, whether proposing a dedicated neural network architecture is necessary and can enhance MNP performance has not been sufficiently verified and explained. Secondly, dedicated neural network architectures are often highly complex, and replicating these networks can be extremely challenging if their implementation code is not publicly available. Lastly, the choice of neural network architecture needs to consider the practical available resources at hand, including data resources and hardware (especially GPU devices).

7.1.2 Rapid construction of high-quality training data. It is well known that the performance of neural network models depends on both the scale and quality of the training data, and MNP models are no exception. With the popularity of the open-source development paradigm, numerous open-source platforms (e.g., GitHub) now contain vast amounts of open-source code, making the construction of large-scale training datasets for MNP models no longer out of reach. However, unfortunately, not all code in open-source projects is of high quality. This makes the rapid construction of large-scale and high-quality training datasets quite challenging. As mentioned in Section 5.1, there are currently several widely used large-scale datasets for training and evaluating MNP models. However, the quality and diversity of these datasets have not been evaluated and verified. This challenge can be further attributed to the difficulties in assessing data quality and diversity.

Data Quality Assessment. Based on our comprehensive review of 32 MNP studies, we find that the field still lacks practical metrics for assessing dataset quality. The design of such metrics is challenging because there is not a strict one-to-one relationship between the method name and the method code body (such as overloaded methods). Furthermore, in addition to the local context contained in the method body itself, numerous MNP techniques also rely on some global contexts, such as callee methods and the class in which the target method appears. There is also a lack of practical, available metrics for evaluating the quality of these global contexts. Last but not least, relying on manual inspection for large datasets is time-consuming and impractical. How to devising automated quality assessment methods based on metrics is also a challenging task.

Data Diversity Assessment. Likewise, we find no existing MNP studies that take sample diversity into account when constructing datasets. Sample diversity is crucial for developing effective, fair, and robust MNP models that can perform well in a variety of real-world MNP situations. Certainly, it is undeniable that assessing method diversity in MNP datasets is a challenging task. Accomplishing such a task requires considering the characteristics of the programming language. For example, there are various types of methods that can be defined in Java, such as instance methods, static methods, getter and setter methods, constructor Methods, and varargs methods. In Python, developers can define instance methods, class methods, dunder methods, etc. Additionally, assessing the diversity of the global contexts also poses a challenge due to their complexity.

7.1.3 Practical applications of MNP techniques. Existing MNP studies primarily focus on improving context extraction and preprocessing, as well as designing advanced MNP models. There is less attention to the practical implementation and application of MNP techniques/models in production environments. Creating usable and user-friendly MNP tools or systems is a highly challenging

task. This challenge can equally be attributed to the difficulties in capturing context and achieving user-friendly interactions in a production environment.

Context Extraction in Production Environment. The actual software development environment is complex, and context extraction may involve (dynamic/static) program analysis techniques, such as control flow and data flow analysis, and function call analysis. Relying on developers to manually provide/extract complete context information is not user-friendly, especially for novice programmers or developers newly joining a project. How to automatically capture context information (including global context) in a production environment has not yet received sufficient attention and research.

Implementation of User-friendly Interaction. While numerous MNP models have been proposed, there is still a considerable distance to cover before these models can be practically implemented in engineering applications. The development of excellent programming assistance tools, including method name recommendation tools, is a complex, systematic engineering task. It not only requires developers to have a full stack of programming skills but also necessitates considerations for the actual user experience. How to implement efficient method name recommendation, and how to make it easier for developers to use have not yet received sufficient attention and research.

7.2 Opportunities

7.2.1 Addressing challenges mentioned in Section 7.1.

Typically, research challenges can also be viewed as opportunities from another perspective. Focusing on the challenges outlined in Section 7.1 and designing corresponding solutions provide many excellent research directions. Research results in these directions will significantly enhance the performance and practical usability of MNP.

7.2.2 Devising semantic-based performance metrics.

In Section 5.2, we have compiled commonly used performance metrics (e.g., *precision* and *ROUGE*) in existing MNP research. While these metrics, overall, contribute to the automated assessment of MNP techniques, they fundamentally calculate the textual similarity between generated method names and reference (or ground-truth) method names, and cannot evaluate their semantic similarity. It should be noted that, in real-world practice, developers with diverse programming/linguistic backgrounds are likely to employ distinct terminologies to convey identical meanings. In addition, variations in programming conventions and practices may result in the utilization of dissimilar abbreviation forms for the same set of terms. In short, developers can name the same method with totally different words. For instance, Feitelson et al. [28] find that the probability of two developers selecting the same name for the same method is only 6.9%. Furthermore, morphological problems (e.g., synonyms, abbreviations, and misspellings) in method names will affect the model train and test result. Even if the model recommends a better method name than the ground truth, existing metrics may also consider its prediction incorrect. In summary, we observe that there is still a lack of semantics-based performance metrics in this field. Attempting to propose and validate such metrics is undoubtedly a promising research opportunity/direction. The research outcomes in this direction would contribute to a more accurate evaluation of MNP performance, fostering further development in this field.

7.2.3 Adapting LLMs to MNP.

Recently, with the success of LLMs in NLP [25, 83], an increasing number of SE researchers have started integrating them into the resolution process of various SE tasks [18, 27, 42, 138], such as code generation [17, 113, 120], program repair [16, 48, 139], and code summarization [2, 102, 103]. To the best of our systematic review of existing research, there is currently no study that investigates

the performance of LLMs on MNP tasks. Certainly, investigating the performance of LLMs in MNP tasks and exploring the factors that affect their performance presents a valuable research opportunity. In addition, similar to LLMs for NLP (e.g., ChatGPT [74] and LLaMA [107]) tasks, there are many LLMs of code for SE tasks, e.g., Codex [75], StarCoder [55], CodeGen [70], and PolyCoder [125]. How to automatically adapt general-purpose LLMs or dedicated code LLMs to MNP tasks is also indeed a promising research opportunity.

8 THREATS TO VALIDITY

8.1 Publication Bias

Publication bias refers to the tendency in academic publishing for researchers to preferentially publish positive results while overlooking or suppressing negative ones. This can lead to a skewed presentation of research findings on a particular topic in the literature. Therefore, if there is publication bias in the primary literature, the arguments supported or refuted by the prior studies selected for this paper may also be biased. To mitigate this threat, we conduct extensive searches across multiple databases and literature repositories to ensure the comprehensive inclusion of all relevant studies and results. Additionally, we cross-validate the findings across different studies to enhance the reliability and validity of our analysis.

8.2 Search Terms

The selection and use of search terms are crucial for comprehensively retrieving all literature relevant to the research topic. We review titles, abstracts, and keywords from several known relevant papers to manually construct search strings and test their applicability. We also utilize popular search databases to retrieve all relevant papers and employ forward and backward snowballing methods. We also consult with authors of relevant papers to ensure any potentially missed articles are identified.

8.3 Study Selection Bias

Study selection bias arises from subjective or objective factors that may skew the selection process, potentially affecting the representativeness and comprehensiveness of the final chosen studies. To mitigate this validity threat, each selected paper undergoes scrutiny by at least two researchers. In cases where consensus cannot be reached between two researchers regarding a particular study, a third researcher is consulted for discussion. This selection process is iterated until researchers achieve a complete consensus on including relevant papers from all the retrieved literature.

8.4 Data Extraction

Data extraction is the process of retrieving relevant data and information from selected studies that align with the research objectives. Extracting data from studies requires manual effort and expertise, which carries the risk of human error. To standardize this process, we have established a clear procedure for identifying, recording, and summarizing data from the study. More than three researchers are involved in the data extraction process: each data item is initially extracted by one researcher from the relevant studies and then independently reviewed by two other researchers for accuracy and consistency.

8.5 Data Analysis

The validity threats in data analysis may arise during the process of handling, organizing, analyzing, and interpreting the collected data. In our survey, we included one of our studies, and while we are confident in the accuracy of our analysis, there is a possibility of misunderstanding technical

or methodological reports from other researchers, or misinterpreting details of their datasets and evaluation metrics. To mitigate these risks, we actively engaged with the authors of the papers we could contact, shared our research findings with them, and sought their feedback. We received confirmation and incorporated their suggestions to further refine our paper.

9 CONCLUSION

MNP tackle the critical challenge of automatically predicting suitable and accurate method names, significantly reducing the naming effort for developers and enhancing software development and maintenance. Recent advancements in learning-based MNP techniques have shown considerable promise, demonstrating the significant potential of ML/DL approaches in this area.

In this paper, we offer a comprehensive review of existing learning-based MNP studies, detailing the general workflow, including context extraction, context preprocessing, and context-based prediction. We summarize various strategies employed by existing techniques to optimize these key subprocesses, and we provide an overview of databases, metrics, datasets, and replication packages within the MNP research community.

Despite the progress made, several challenges remain. Accurately understanding and extracting context, preprocessing effectively, and encoding context remain significant hurdles. The construction of high-quality training data, assessing data quality and diversity, and the practical application of MNP techniques in real-world environments also pose ongoing challenges. Addressing these issues is crucial for advancing the field and improving the practical utility of MNP tools.

Future research should focus on overcoming these challenges by developing more robust methods for context extraction and preprocessing, improving data quality and diversity, and creating user-friendly applications. By addressing these areas, the field of learning-based MNP can continue to evolve and make meaningful contributions to SE.

10 ACKNOWLEDGMENTS

This work is supported partially by the National Natural Science Foundation of China (61932012, 62372228, 62141215), Science, Technology, and Innovation Commission of Shenzhen Municipality (CJGJZD20200617103001003).

REFERENCES

- [1] Ibtissam Abnane, Ali Idri, Imane Chlioui, and Alain Abran. 2023. Evaluating ensemble imputation in software effort estimation. *Empir. Softw. Eng.* 28, 2 (2023), 56.
- [2] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot Training LLMs for Project-specific Code-Summarization. In *Proceedings of the 37th International Conference on Automated Software Engineering*. ACM, Rochester, MI, USA, 177:1–177:5.
- [3] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual Training for Software Engineering. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh, PA, USA, 1443–1455.
- [4] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, Bergamo, Italy, 38–49.
- [5] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *Proceedings of the 41st International Conference on Programming Language Design and Implementation*. ACM, London, UK, 91–105.
- [6] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *Proceedings of the 6th International Conference on Learning Representations*. OpenReview.net, Vancouver, BC, Canada, 1–17.
- [7] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33rd International Conference on Machine Learning*. ICML, New York City, NY, USA, 2091–2100.
- [8] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33rd International Conference on Machine Learning*. JMLR.org, New York City, NY, USA, 2091–2100.

- [9] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR, San Francisco, CA, USA, 207–216.
- [10] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *Proceedings of the 7th International Conference on Learning Representations*. OpenReview.net, New Orleans, LA, USA, 1–13.
- [11] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-based Representation for Predicting Program Properties. In *Proceedings of the 39th Conference on Programming Language Design and Implementation*. ACM, Philadelphia, PA, USA, 404–419.
- [12] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages* 3, 40 (2019), 40:1–40:29.
- [13] Francesco Bertolotti and Walter Cazzola. 2023. Fold2Vec: Towards a Statement-Based Representation of Code for Code Comprehension. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 6:1–6:31.
- [14] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees. In *Proceedings of 43rd International Conference on Software Engineering*. IEEE, Madrid, Spain, 1186–1197.
- [15] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. TreeCaps: Tree-Based Capsule Networks for Source Code Processing. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 1 (2021), 30–38.
- [16] Jialun Cao, Meiziniu Li, Ming Wen, and Shing chi Cheung. 2023. A Study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair. *CoRR abs/2304.08191*, 1 (2023), 1–12.
- [17] Hailin Chen, Amrita Saha, Steven Chu-Hong Hoi, and Shafiq Joty. 2023. Personalized Distillation: Empowering Open-Sourced LLMs with Adaptive Learning for Code Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Singapore, 6737–6749.
- [18] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR abs/2107.03374*, 1 (2021), 1–19.
- [19] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.
- [20] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. 2013. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In *Proceedings of the 29th International Conference on Software Maintenance*. IEEE Computer Society, Eindhoven, The Netherlands, 516–519.
- [21] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. 2020. Embedding Java Classes with code2vec: Improvements from Variable Obfuscation. In *Proceedings of the 17th International Conference on Mining Software Repositories*. ACM, Seoul, Republic of Korea, 243–253.
- [22] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*. Springer, Aarhus, Denmark, 77–101.
- [23] Prem Devanbu, Matthew B. Dwyer, Sebastian G. Elbaum, Michael Lowry, Kevin Moran, Denys Poshyvanyk, Baishakhi Ray, Rishabh Singh, and Xiangyu Zhang. 2020. Deep Learning & Software Engineering: State of Research and Future Directions. *CoRR abs/2009.08525*, 1 (2020), 1–37.
- [24] Junwei Du, Xinshuang Ren, Haojie Li, Feng Jiang, and Xu Yu. 2023. Prediction of bug-fixing time based on distinguishable sequences fusion in open source software. *J. Softw. Evol. Process.* 35, 11 (2023), 1–20.
- [25] Mengnan Du, Fengxiang He, Na Zou, Dacheng Tao, and Xia Hu. 2022. Shortcut Learning of Large Language Models in Natural Language Understanding: A Survey. *CoRR abs/2208.11857*, 1 (2022), 1–10.
- [26] Tore Dybå and Torgeir Dingsøyr. 2008. Empirical studies of agile software development: A systematic review. *Information & Software Technology*. 50, 9-10 (2008), 833–859.
- [27] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. *CoRR abs/2310.03533*, 1 (2023), 1–23.
- [28] Dror G. Feitelson, Ayelet Mizrahi, Nofar Noy, Aviad Ben Shabat, Or Eliyahu, and Roy Sheffer. 2022. How Developers Choose Names. *IEEE Transactions on Software Engineering* 48, 2 (2022), 37–52.

- [29] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured neural summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, New Orleans, LA, USA, 1–12.
- [30] Shogo Fujita, Hidetaka Kamigaito, Hiroya Takamura, and Manabu Okumura. 2020. Pointing to Subwords for Generating Function Names in Source Code. In *Proceedings of the 28th International Conference on Computational Linguistics*. COLING, Barcelona, Spain (Online), 316–327.
- [31] Han Gao, Shaoyin Cheng, Yinxing Xue, and Weiming Zhang. 2021. A Lightweight Framework for Function Name Reassignment Based on Large-Scale Stripped Binaries. In *Proceedings of the 30th International Symposium on Software Testing and Analysis*. ACM, Virtual Event, Denmark, 607–619.
- [32] Sa Gao, Chunyang Chen, Zhenchang Xing, Yukun Ma, Wen Song, and Shang-Wei Lin. 2019. A Neural Model for Method Name Generation from Functional Description. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Hangzhou, China, 411–421.
- [33] Fan Ge and Li Kuang. 2021. Keywords Guided Method Name Generation. In *Proceedings of the 29th International Conference on Program Comprehension*. IEEE, Madrid, Spain, 196–206.
- [34] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg, Sweden, 933–944.
- [35] Zhenting Guo, Meng Yan, Hongyan Li, Zhezhe Chen, and Weifeng Sun. 2023. Just-In-Time Method Name Updating With Heuristics and Neural Model. In *Proceedings of the 23rd International Conference on Software Quality, Reliability, and Security*. IEEE, Chiang Mai, Thailand, 707–718.
- [36] Mayy Habayeb, Syed Shariyar Murtaza, Andriy V. Miranskyy, and Ayse Basar Bener. 2018. On the Use of Hidden Markov Model to Predict the Time to Fix Bugs. *IEEE Trans. Software Eng.* 44, 12 (2018), 1224–1244.
- [37] William L. Hamilton, Zhitaoying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 30th Conference on Neural Information Processing Systems*. MIT Press, Long Beach, CA, USA, 1024–1034.
- [38] Ruidong Han, Siqi Ma, Juanru Li, Surya Nepal, David Lo, Zhuo Ma, and Jianfeng Ma. 2024. Range Specification Bug Detection in Flight Control System Through Fuzzing. *IEEE Trans. Software Eng.* 50, 3 (2024), 461–473.
- [39] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Computer Society, Zurich, Switzerland, 837–847.
- [40] Roya Hosseini and Peter Brusilovsky. 2013. JavaParser; A Fine-Grain Concept Indexing Tool for Java Problems. In *Proceedings of the Workshops at the 16th International Conference on Artificial Intelligence in Education*. CEUR-WS.org, Memphis, USA, 60–63.
- [41] Einar W. Høst and Bjarte M. Østvold. 2009. Debugging Method Names. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*. Springer, Genoa, Italy, 294–317.
- [42] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John C. Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. *CoRR abs/2308.10620*, 1 (2023), 1–62.
- [43] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation. In *Proceedings of the 26th International Conference on Program Comprehension*. ACM, Gothenburg, Sweden, 200–210.
- [44] Jianjun Huang, Jianglei Nie, Yuanjun Gong, Wei You, Bin Liang, and Pan Bian. 2024. Raisin: Identifying Rare Sensitive Functions for Bug Detection. In *Proceedings of the 46th International Conference on Software Engineering*. ACM, Lisbon, Portugal, 175:1–175:12.
- [45] Michael Hucka. 2018. Spiral: Splitters for Identifiers in Source Code Files. *Journal of Open Source Software* 3, 24 (2018), 653.
- [46] Falleri Jean-Rémy, Morandat Floréal, Blanc Xavier, Martinez Matias, and Monperrus Martin. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th International Conference on Automated Software Engineering*. ACM, Vasteras, Sweden, 313–324.
- [47] Lin Jiang, Hui Liu, and He Jiang. 2019. Machine Learning Based Recommendation of Method Names: How Far are We. In *Proceedings of the 34th International Conference on Automated Software Engineering*. IEEE, San Diego, CA, USA, 602–614.
- [48] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs over Retrieval-Augmented Prompts. *CoRR abs/2303.07263*, 1 (2023), 1–11.
- [49] Thomas N. Kipf and Max Welling. 2016. Variational Graph Auto-Encoders. *CoRR abs/1611.07308*, 1 (2016), 1–3.
- [50] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*. OpenReview.net, Toulon, France, 1–14.

- [51] Barbara Kitchenham. 2004. Procedures for performing systematic reviews. *Keele, UK, Keele University* 33, 2004 (2004), 1–26.
- [52] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners’ Expectations on Automated Fault Localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, Saarbrücken, Germany, 165–176.
- [53] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization*. IEEE, San Jose, CA, USA, 75–88.
- [54] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE / ACM, Montreal, QC, Canada, 795–806.
- [55] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *CoRR abs/2305.06161*, 1 (2023), 1–44.
- [56] Yue Li, Zhong Ren, Zhiqi Wang, Lanxin Yang, Liming Dong, Chenxing Zhong, and He Zhang. 2024. Fine-SE: Integrating Semantic Features and Expert Features for Software Effort Estimation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. ACM, Lisbon, Portugal, 27:1–27:12.
- [57] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. A Context-based Automated Approach for Method Name Consistency Checking and Suggestion. In *Proceedings of the 43rd International Conference on Software Engineering*. IEEE, Madrid, Spain, 574–586.
- [58] Yanli Li, Chongbin Ye, Huaming Chen, Shiping Chen, Minhui Xue, and Jun Shen. 2023. Towards Better ML-Based Software Services: An Investigation of Source Code Engineering Impact. In *Proceedings of the 2023 International Conference on Software Services Engineering*. IEEE, Chicago, IL, USA, 1–10.
- [59] Fang Liu, Ge Li, Zhiyi Fu, Shuai Lu, Yiyang Hao, and Zhi Jin. 2022. Learning to Recommend Method Names with Global Context. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh, PA, USA, 1294–1306.
- [60] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st International Conference on Software Engineering*. ICSE, Montreal, QC, Canada, 1–12.
- [61] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon. 2021. Mining Fix Patterns for FindBugs Violations. *IEEE Trans. Software Eng.* 47, 1 (2021), 165–188.
- [62] Linfeng Liu, Hoan Nguyen, George Karypis, and Srinivasan Sengamedu. 2021. Universal Representation for Code. In *Proceedings of the 25th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*. Springer, Virtual Event, 16–28.
- [63] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural Machine Translation-based Commit Message Generation: How Far Are We?. In *Proceedings of the 33rd International Conference on Automated Software Engineering*. ACM, Montpellier, France, 373–384.
- [64] Wei Ma, Mengjie Zhao, Ezekiel O. Soremekun, Qiang Hu, Jie M. Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. 2022. GraphCode2Vec: Generic Code Embedding via Lexical and Program Dependence Analyses. In *Proceedings of the 19th International Conference on Mining Software Repositories*. ACM, Pittsburgh, PA, USA, 524–536.
- [65] Yi-Fan Ma, Yali Du, and Ming Li. 2023. Capturing the Long-Distance Dependency in the Control Flow Graph via Structural-Guided Attention for Bug Localization. In *Proceedings of the 32nd International Joint Conference on Artificial Intelligence*. ijcai.org, Macao, SAR, China, 2242–2250.
- [66] Junayed Mahmud, Nadeeshan De Silva, Safwat Ali Khan, Seyed Hooman Mostafavi, S. M. Hasan Mansur, Oscar Chaparro, Andrian Marcus, and Kevin Moran. 2024. On Using GUI Interaction Data to Improve Text Retrieval-based Bug Localization. In *Proceedings of the 46th International Conference on Software Engineering*. ACM, Lisbon, Portugal, 40:1–40:13.

- [67] Andriy Mnih and Geoffrey E. Hinton. 2007. Three New Graphical Models for Statistical Language Modelling. In *Proceedings of the 24th International Conference on Machine Learning*. ACM, Corvallis, Oregon, USA, 641–648.
- [68] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the 30th Conference on Artificial Intelligence*. AAAI Press, Phoenix, Arizona, USA, 1287–1293.
- [69] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting Natural Method Names to Check Name Consistencies. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM, Seoul, South Korea, 1372–1384.
- [70] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *Proceedings of the 11th International Conference on Learning Representations*. OpenReview.net, Kigali, Rwanda, 1–13.
- [71] Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence*. IJCAI, Vienna, Austria, 5546–5555.
- [72] Nan Niu, Juha Savolainen, Tanmay Bhowmik, Anas Mahmoud, and Sandeep Reddivari. 2012. A Framework for Examining Topical Locality in Object-oriented Software. In *Proceedings of the 36th Annual Computer Software and Applications Conference*. IEEE Computer Society, Izmir, Turkey, 219–224.
- [73] Safa Omri and Carsten Sinz. 2020. Deep Learning for Software Defect Prediction: A Survey. In *Proceedings of the 42nd International Conference on Software Engineering, Workshops*. ACM, Seoul, Republic of Korea, 209–214.
- [74] OpenAI. 2022. ChatGPT. site: <https://openai.com/blog/chatgpt>. Accessed December, 2023.
- [75] OpenAI. 2023. Codex. site: <https://openai.com/blog/openai-codex>. Accessed December, 2023.
- [76] Saeed Parsa, Morteza Zakeri Nasrabadi, Masoud Ekhtiarzadeh, and Mohammad Ramezani. 2023. Method Name Recommendation Based on Source Code Metrics. *Journal of Computer Languages* 74, 1 (2023), 101177.
- [77] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. SPOON: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46, 9 (2016), 1155–1179.
- [78] Han Peng, Ge Li, Wenhan Wang, Yunfei Zhao, and Zhi Jin. 2021. Integrating Tree Path in Transformer for Code Representation. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*. NeurIPS, virtual, 9343–9354.
- [79] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *Proceedings of the 19th Conference on Empirical Methods in Natural Language Processing*. ACL, Doha, Qatar, 1532–1543.
- [80] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for Conducting Systematic Mapping Studies in Software Engineering: An Update. *Information and Software Technology* 64, 1 (2015), 1–18.
- [81] Maxim Petukhov, Evelina Gudauskayte, Arman Kaliyev, Mikhail Oskin, Dmitry Ivanov, and Qianxiang Wang. 2022. Method Name Prediction for Automatically Generated Unit Tests. In *Proceedings of the 2nd International Conference on Code Quality*. IEEE, Innopolis, Russian, 29–38.
- [82] Hanwei Qian, Wei Liu, Ziqi Ding, Weisong Sun, and Chunrong Fang. 2023. Abstract Syntax Tree for Method Name Prediction: How Far Are We?. In *Proceedings of the 23rd International Conference on Software Quality, Reliability, and Security*. IEEE, Chiang Mai, Thailand, 464–475.
- [83] Chengwei Qin, Aston Zhang, Zhuosheng Zhang, Jiaao Chen, Michihiro Yasunaga, and Diyi Yang. 2023. Is ChatGPT a General-Purpose Natural Language Processing Task Solver?. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Singapore, 1339–1384.
- [84] Zhiheng Qu, Yi Hu, Jianhui Zeng, Bo Cai, and Shun Yang. 2022. Method Name Generation Based on Code Structure Guidance. In *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Honolulu, HI, USA, 1101–1110.
- [85] Md. Rafiqul Islam Rabin, Nghi D. Q. Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Inf. Softw. Technol.* 135 (2021), 106552.
- [86] Md. Rafiqul Islam Rabin, Arjun Mukherjee, Omprakash Gnawali, and Mohammad Amin Alipour. 2020. Towards Demystifying Dimensions of Source Code Embeddings. *CoRR* abs/2008.13064, 1 (2020), 1–10.
- [87] Juan Ramos. 2003. Using TF-IDF to determine word relevance in document queries. In *Proceedings of the First Instructional Conference on Machine Learning*, Vol. 242. Citeseer, 133–142.
- [88] Veselin Raychev, Pavol Bielik, and Martin T. Vechev. 2016. Probabilistic Model for Code with Decision Trees. In *Proceedings of the 31st International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, Amsterdam, The Netherlands, 731–747.

- [89] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI, Edinburgh, United Kingdom, 419–428.
- [90] Felice Salviulo and Giuseppe Scanniello. 2014. Dealing with Identifiers and Comments in Source Code Comprehension and Maintenance: Results from an Ethnographically-informed Study with Students and Professionals. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, London, England, United Kingdom, 48:1–48:10.
- [91] Hazem Peter Samoa, Firas Bayram, Pasquale Salza, and Philipp Leitner. 2022. A systematic mapping study of source code representation for deep learning in software engineering. *IET Softw.* 16, 4 (2022), 351–385.
- [92] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *Proceedings of the 15th International Conference on The Semantic Web*. Springer, Heraklion, Crete, Greece, 593–607.
- [93] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*. The Association for Computer Linguistics, Berlin, Germany, 1715–1725.
- [94] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI, Philadelphia, PA, USA, 693–706.
- [95] Nathan Sobo. 2018. Tree-sitter. site: <https://github.com/tree-sitter/tree-sitter>. Accessed: December 31, 2023.
- [96] Egor Spirin, Egor Bogomolov, Vladimir Kovalenko, and Timofey Bryksin. 2021. PSIMiner: A Tool for Mining Rich Abstract Syntax Trees from Code. In *Proceedings of the 18th International Conference on Mining Software Repositories*. IEEE, Madrid, Spain, 13–17.
- [97] Keele Staffs. 2007. Guidelines for Performing Systematic Literature Reviews in Software Engineering. site: http://robertfeldt.net/advice/kitchenham_2007_systematic_reviews_report_updated.pdf. Accessed: December 31, 2023.
- [98] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: value-flow-based precise code embedding. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 233:1–233:27.
- [99] Weisong Sun, Chunrong Fang, Yuchen Chen, Guan hong Tao, Tingxu Han, and Quan jun Zhang. 2022. Code Search based on Context-aware Code Translation. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh, PA, USA, 388–400.
- [100] Weisong Sun, Chunrong Fang, Yuchen Chen, Quan jun Zhang, Guan hong Tao, Yudu You, Tingxu Han, Yifei Ge, Yuling Hu, Bin Luo, and Zhenyu Chen. 2023. An Extractive-and-Abstractive Framework for Source Code Summarization. *ACM Transactions on Software Engineering and Methodology* Just Accepted, 1 (2023), 1–39.
- [101] Weisong Sun, Chunrong Fang, Yifei Ge, Yuling Hu, Yuchen Chen, Quan jun Zhang, Xiuting Ge, Yang Liu, and Zhenyu Chen. 2024. A Survey of Source Code Search: A 3-Dimensional Perspective. *ACM Trans. Softw. Eng. Methodol.* 33, 6 (2024), 166.
- [102] Weisong Sun, Chunrong Fang, Yudu You, Yuchen Chen, Yi Liu, Chong Wang, Jian Zhang, Quan jun Zhang, Hanwei Qian, Wei Zhao, Yang Liu, and Zhenyu Chen. 2023. A Prompt Learning Framework for Source Code Summarization. *CoRR abs/2312.16066*, 1 (2023), 1–23.
- [103] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quan jun Zhang, Hanwei Qian, Yang Liu, and Zhenyu Chen. 2023. Automatic Code Summarization via ChatGPT: How Far Are We? *CoRR abs/2305.12865* (2023), 1–13.
- [104] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*. MIT Press, Montreal, Quebec, Canada, 3104–3112.
- [105] Takayuki Suzuki, Kazunori Sakamoto, Fuyuki Ishikawa, and Shinichi Honiden. 2014. An approach for evaluating and suggesting method names using n-gram models. In *22nd International Conference on Program Comprehension*. ICPC, Hyderabad, India, 271–274.
- [106] Chris Thunes. 2013. Javalang: Python Library for Working with Java Source Code. <https://github.com/c2nes/javalang>. Accessed: December 31, 2023.
- [107] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR abs/2302.13971*, 1 (2023), 1–16.
- [108] Ilya Utkin, Egor Spirin, Egor Bogomolov, and Timofey Bryksin. 2022. Evaluating the Impact of Source Code Parsers on ML4SE Models. *CoRR abs/2206.08713*, 1 (2022), 1–12.
- [109] Dheeraj Vagavolu, Karthik Chandra Swarna, and Sridhar Chimalakonda. 2021. A Mocktail of Source Code Representations. In *Proceedings of the 36th International Conference on Automated Software Engineering*. IEEE, Melbourne, Australia, 1296–1300.

- [110] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot-a Java Bytecode Optimization Framework. In *Proceedings of the 9th Conference of The Centre for Advanced Studies on Collaborative Research*. IBM, Mississauga, Ontario, Canada, 214–224.
- [111] Zane Varner, Çerağ Oğuztüzün, and Feng Long. 2022. Neural Model for Generating Method Names from Combined Contexts. In *Proceedings of the 29th Annual Software Technology Conference*. IEEE, Gaithersburg, MD, USA, 1–6.
- [112] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *Proceedings of the 6th International Conference on Learning Representations*. OpenReview.net, Vancouver, BC, Canada, 1–12.
- [113] Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. 2024. Teaching Code LLMs to Use Autocompletion Tools in Repository-Level Code Generation. *CoRR* abs/2401.06391, 1 (2024), 1–13.
- [114] Daniel C. Wang, Andrew W. Appel, Jeffrey L. Korn, and Christopher S. Serra. 1997. The Zephyr Abstract Syntax Description Language. In *Proceedings of the 1st Conference on Domain-Specific Languages*. USENIX, Santa Barbara, California, USA, 213–228.
- [115] Ke Wang and Zhendong Su. 2020. Blended, Precise Semantic Program Embeddings. In *Proceedings of the 41st International Conference on Programming Language Design and Implementation*. ACM, London, UK, 121–134.
- [116] Simin Wang, Liguang Huang, Amiao Gao, Jidong Ge, Tengfei Zhang, Haitao Feng, Ishna Satyarth, Ming Li, He Zhang, and Vincent Ng. 2023. Machine/Deep Learning for Software Engineering: A Systematic Literature Review. *IEEE Trans. Software Eng.* 49, 3 (2023), 1188–1231.
- [117] Shangwen Wang, Ming Wen, Bo Lin, Yepang Liu, Tegawendé F. Bissyandé, and Xiaoguang Mao. 2023. Pre-implementation Method Name Prediction for Object-oriented Programming. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 157:1–157:35.
- [118] Shangwen Wang, Ming Wen, Bo Lin, and Xiaoguang Mao. 2021. Lightweight Global and Local Contexts Guided Method Name Recommendation with Prior Knowledge. In *Proceedings of the 29th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Athens, Greece, 741–753.
- [119] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning Semantic Program Embeddings with Graph Interval Neural Network. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 137:1–137:27.
- [120] Zejun Wang, Jia Li, Ge Li, and Zhi Jin. 2023. ChatCoder: Chat-based Refine Requirement Improves LLMs’ Code Generation. *CoRR* abs/2311.00272, 1 (2023), 1–11.
- [121] Cody Watson, Nathan Cooper, David Nader-Palacio, Kevin Moran, and Denys Poshyvanyk. 2022. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. *ACM Trans. Softw. Eng. Methodol.* 31, 2 (2022), 32:1–32:58.
- [122] Da Xiao, Dengji Hang, Lu Ai, Shengping Li, and Hongliang Liang. 2022. Path context augmented statement and network for learning programs. *Empir. Softw. Eng.* 27, 2 (2022), 37.
- [123] Rui Xie, Wei Ye, Jinan Sun, and Shikun Zhang. 2021. Exploiting Method Names to Improve Code Summarization: A Deliberation Multi-Task Learning Approach. In *29th International Conference on Program Comprehension*. ICPC, Madrid, Spain, 138–148.
- [124] Yutao Xie, Jiayi Lin, Hande Dong, Lei Zhang, and Zhonghai Wu. 2024. Survey of Code Search Based on Deep Learning. *ACM Trans. Softw. Eng. Methodol.* 33, 2 (2024), 54:1–54:42.
- [125] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th International Symposium on Machine Programming*. ACM, San Diego, CA, USA, 1–10.
- [126] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *Proceedings of the 7th International Conference on Learning Representations*. OpenReview.net, New Orleans, LA, USA, 1–17.
- [127] Sihan Xu, Sen Zhang, Weijing Wang, Xinya Cao, Chenkai Guo, and Jing Xu. 2019. Method Name Suggestion with Hierarchical Attention Networks. In *Proceedings of the 2019 Workshop on Partial Evaluation and Program Manipulation*. ACM, Cascais, Portugal, 10–21.
- [128] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2020. Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts. In *Proceedings of the 35th International Conference on Automated Software Engineering*. IEEE, Melbourne, Australia, 1029–1040.
- [129] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 35th Symposium on Security and Privacy*. IEEE Computer Society, Berkeley, CA, USA, 590–604.
- [130] Yanming Yang, Xin Xia, David Lo, and John C. Grundy. 2022. A Survey on Deep Learning for Software Engineering. *ACM Comput. Surv.* 54, 10s (2022), 206:1–206:73.
- [131] Yanping Yang, Ling Xu, Meng Yan, Zhou Xu, and Zhongyang Deng. 2022. A Naming Pattern Based Approach for Method Name Recommendation. In *Proceedings of the 33rd International Symposium on Software Reliability Engineering*.

- IEEE, Charlotte, NC, USA, 344–354.
- [132] Hiroshi Yonai, Yasuhiro Hayase, and Hiroyuki Kitagawa. 2019. Mercem: Method Name Recommendation Based on Call Graph Embedding. In *Proceedings of the 26th Asia-Pacific Software Engineering Conference*. IEEE, Putrajaya, Malaysia, 134–141.
 - [133] Yijun Yu. 2019. fAST: Flattening Abstract Syntax Trees for Efficiency. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. IEEE / ACM, Montreal, QC, Canada, 278–279.
 - [134] Oleksandr Zaitsev, Stéphane Ducasse, Alexandre Bergel, and Mathieu Eveillard. 2020. Suggesting Descriptive Method Names: An Exploratory Study of Two Machine Learning Approaches. In *Proceedings of the 13th International Conference on the Quality of Information and Communications Technology*. QUATIC, Faro, Portugal, 93–106.
 - [135] Fengyi Zhang, Bihuan Chen, Rongfan Li, and Xin Peng. 2021. A Hybrid Code Representation Learning Approach for Predicting Method Names. *Journal of Systems and Software* 180, 1 (2021), 111011.
 - [136] Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. 2022. Learning to Represent Programs with Heterogeneous Graphs. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. ACM, Virtual Event, 378–389.
 - [137] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2024. A Survey of Learning-based Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 2 (2024), 55:1–55:69.
 - [138] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023. A Survey on Large Language Models for Software Engineering. *CoRR* abs/2312.15223, 1 (2023), 1–57.
 - [139] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. A Critical Review of Large Language Model on Software Engineering: An Example from ChatGPT and Automated Program Repair. *CoRR* abs/2310.08879, 1 (2023), 1–12.
 - [140] Xufan Zhang, Yilin Yang, Yang Feng, and Zhenyu Chen. 2019. Software Engineering Practice in the Development of Deep Learning Applications. *CoRR* abs/1910.03156, 1 (2019), 1–11.
 - [141] Li Zhong, Chengcheng Xiang, Haochen Huang, Bingyu Shen, Eric Mugnier, and Yuanyuan Zhou. 2024. Effective Bug Detection with Unused Definitions. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22–25, 2024*. ACM, Athens, Greece, 720–735.
 - [142] Jie Zhu, Lingwei Li, Li Yang, Xiaoxiao Ma, and Chun Zuo. 2023. Automating Method Naming with Context-Aware Prompt-Tuning. In *Proceedings of the 31st International Conference on Program Comprehension*. IEEE, Melbourne, Australia, 203–214.
 - [143] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-Agnostic Representation Learning of Source Code from Structure and Context. In *Proceedings of the 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, 2021*. OpenReview.net, Virtual Event, Austria.