Streamlining Java Programming: Uncovering Well-Formed Idioms with IdioMine

Yanming Yang
The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
yanmingyang@zju.edu.cn

Xing Hu*
The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
xinghu@zju.edu.cn

Xin Xia
Zhejiang University, China
xin.xia@acm.org

David Lo
Singapore Management University, Singapore
davidlo@smu.edu.sg

Xiaohu Yang
The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
yangxh@zju.edu.cn

ABSTRACT

Code idioms are commonly used patterns, techniques, or practices that aid in solving particular problems or specific tasks across multiple software projects. They can improve code quality, performance, and maintainability, and also promote program standardization and reuse across projects. However, identifying code idioms is significantly challenging, as existing studies have still suffered from three main limitations. First, it is difficult to recognize idioms that span non-contiguous code lines. Second, identifying idioms with intricate data flow and code structures can be challenging. Moreover, they only extract dataset-specific idioms, so common idioms or well-established code/design patterns that are rarely found in datasets cannot be identified.

To overcome these limitations, we propose a novel approach, named IdioMine, to automatically extract generic and specific idioms from both Java projects and libraries. We perform program analysis on Java functions to transform them into concise PDGs, for integrating the data flow and control flow of code fragments. We then develop a novel chain structure, Data-driven Control Chain (DCC), to extract sub-idioms that possess contiguous semantic meanings from PDGs. After that, we utilize GraphCodeBERT to generate code embeddings of these sub-idioms and perform density-based clustering to obtain frequent sub-idioms. We use heuristic rules to identify interrelated sub-idioms among the frequent ones. Finally, we employ ChatGPT to synthesize interrelated sub-idioms into potential code idioms and infer real idioms from them.

We conduct well-designed experiments and a user study to evaluate IdioMine's correctness and the practical value of the extracted idioms. Our experimental results show that IdioMine effectively extracts more idioms with better performance in most metrics. We compare our approach with *Haggis* and ChatGPT, IdioMine outperforms them by 22.8% and 35.5% in Idiom Set Precision (ISP) and by 9.7% and 22.9% in Idiom Coverage (IC) when extracting idioms from libraries. IdioMine also extracts almost twice the size of idioms than the baselines, exhibiting its ability to identify complete idioms. Our user study indicates that idioms extracted by IdioMine are well-formed and semantically clear. Moreover, we conduct a qualitative and quantitative analysis to investigate the primary functionalities of IdioMine's extracted idioms from various projects and libraries.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

Code Idiom Mining, Code Pattern, Large Language Model (LLM), Clustering

ACM Reference Format:
Yanming Yang, Xing Hu, Xin Xia, David Lo, and Xiaohu Yang. 2024. Streamlining Java Programming: Uncovering Well-Formed Idioms with IdioMine. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639135

1 INTRODUCTION

A code idiom is a programming pattern used to solve a specific problem efficiently and concisely [1, 64], and it is commonly encountered across diverse software projects [2]. According to Di Nucci et al. [40], if a piece of code is written in a natural and intuitive manner, it can be considered idiomatic. Well-established idioms can reduce ambiguity [55], optimize code performance and efficiency [58], and enhance code readability and understanding for programmers [54]. They also help establish coding standards and best practices in development activities [49] and facilitate their reuse in various contexts, such as data processing and resource management [40, 55]. In addition, developers widely recognize the significance of writing idiomatic code, evident from the abundance of relevant resources dedicated to this topic [36]. For example, A dedicated book on C++ idioms can be found on Wikibooks [3], and similar guides are also available for Java [4] and JavaScript [5, 6].

*Corresponding author

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04
https://doi.org/10.1145/3597503.3639135

An idiomatic JavaScript guide on GitHub [6] has gained considerable popularity, amassing over 22.8k stars and 3.2k forks. Therefore, mining code idioms is critical for enhancing the efficiency and effectiveness of software development and maintenance [44].

Several existing studies focus on code idiom extraction. Allamanis et al. [36] proposed a pioneering model, *Haggis*, a machine learning-based statistical method that mines code idioms based on code syntax. However, their approach relies heavily on manual annotations and is unable to identify code idioms with interleaved dataflow [64]. This indicates that *Haggis* is a labor-intensive approach, and less experienced programmers may find it difficult to generalize code patterns. To overcome this limitation, Sivaraman et al. [64] introduced dataflow information to improve the tree representation method used by Allamanis et al. [34, 36]. However, this approach only resulted in a marginal enhancement in *Haggis*'s performance, as it failed to merge the full semantic context of source code to identify code idioms.

Extracting code idioms with complete semantic information from software projects remains a challenging task due to three unresolved limitations in existing studies:

1) *Existing studies cannot extract idioms that are composed of non-contiguous lines of code from a code fragment.* Existing methods, as demonstrated by previous studies [36, 64], are unable to extract code idioms from non-contiguous lines of code. This restriction poses a challenge, as many idioms involve non-contiguous lines within a function. For instance, Listing 1 showcases an idiom extracted by IdioMine that involves non-contiguous lines of code. This particular idiom [7] is derived from code that utilizes the `com.rabbitmq` library, where it calculates the execution time of a code fragment by determining its start and end times. As the code in lines 1 and 3 are typically non-contiguous in most projects, current approaches are unable to detect such widely-used idioms.

Listing 1 An idiom with non-contiguous lines of code.

```
1 long start = System.currentTimeMillis();
2 # code snippet
3 long end= System.currentTimeMillis();
4 long time = end - start;
```

2) *Existing studies cannot effectively identify idioms that encompass intricate data flow and code structures.* A significant drawback of recent research on idiom mining [64] is that it captures only one-way information flow. This limitation may result in the loss of important semantic and data flow in code fragments, making it challenging to identify complex code idioms that involve intricate data and control flows. This difficulty persists irrespective of whether these idioms are constructed from contiguous or non-contiguous lines of code. As an example, Listing 2 exhibits an idiom, which is extracted from the source file named "`FileManager.java`" [8]. This idiom is a commonly used pattern for Git-based projects, e.g., JGit. It is used to push modifications to a specific Git repository based on its parent directory. Its intricate control and data flow are contained within a contiguous code fragment (specifically, Lines 3-7). This fragment encompasses an `if-else` statement and an `inner loop` structure featuring an additional `if` statement. Unfortunately, existing techniques fail to extract such idioms due to their intricate control and data flow as well as non-contiguous code lines. Note that the previous study [64] addressed the contiguity of code lines within idioms and the complexity of data and control flow as distinct issues in their Limitation Section. Hence, we also consider this limitation as a separate concern from the first one discussed.

Listing 2 An idiom with complex code structure.

```
1 if (!file.exists() || !file.isDirectory()){
2 # code snippet
3 } else{for (Repository r: reps){
4 if(r.getDirectory().getParentFile().getName().equals(projectName))
5 {   Git git = new Git(refreshed);
6     RevCommit commit = git.commit().setMessage(message)
7     .setAll(true).call();}}}
```

3) *Existing studies can only produce dataset-specific idioms, rather than generic idioms.* According to the public definition [2], *a code idiom is characterized as a code fragment that exhibits frequent recurrence across various software projects.* This implies that solely mining code idioms from specific datasets may overlook certain generic idioms that possess wider applicability but occur infrequently within that dataset. However, current techniques heavily rely on probability distributions of code fragments in the dataset to identify idioms. They only consider frequently used code fragments in the dataset as idioms, but some less common fragments in that dataset may still represent well-established and easy-to-reuse design/code patterns in other projects. This limitation may prevent existing techniques from recognizing certain real idioms. Listing 3 illustrates the use of an idiom [9] that encapsulates transactional behavior for structuring the project named *Stuctutr* [10]. This idiom incorporates the well-known "*Command Design Pattern*" [11]. Although the design pattern appears only twice in the dataset, it can be easily reused elsewhere because of its primary functionality, which involves encapsulating the information required to perform an action or trigger an event at a later time.

Listing 3 A design pattern with few occurrences in datasets.

```
1 Services.command(securityContext,
2     TransactionCommand.class).execute(new StructrTransaction(){
3 @Override public Object execute() {
4     if ((lockType == LockType.READ) &
5             (graphDb instanceof AbstractGraphDatabase)){
6         ((AbstractGraphDatabase)node.getGraphDatabase())
7             .getLockManager().getReadLock(node); }
8     return null; } };
```

To overcome the aforementioned constraints, our paper puts forth a novel approach, IdioMine, that can automatically recognize high-quality code idioms in an unsupervised manner. IdioMine comprises three distinct phases, i.e., idiom representation construction, frequent sub-idiom mining, and semantic idiom identification. Specifically, IdioMine analyzes Java functions to create concise program dependency graphs (PDGs), which seamlessly integrates dataflow and control information for a better understanding of the code's semantics. It then establishes a chain structure called Data-driven Control Chain (DCC) to extract code lines with contiguous semantic meanings as sub-idioms from PDGs. In phase two, we employ GraphCodeBERT to generate code embeddings for these sub-idioms and a density-based clustering technique to identify frequent sub-idioms. In phase three, we identify related sub-idioms among frequent ones by applying predefined rules. Afterward, with the aid of ChatGPT's capabilities, IdioMine synthesizes related sub-idioms as potential idioms and identifies real ones from them.

We evaluate IdioMine's effectiveness through extensive experiments on project-level and library-level datasets, and perform a user

study to explore the practical value of extracted idioms. In addition, we perform a quantitative and qualitative analysis of IdioMine's extracted idioms, thereby discovering novel insights. After comparing the results of IdioMine with baselines, we observe that IdioMine outperforms them in many evaluation metrics. Specifically, IdioMine exhibits exceptional performance in identifying idioms, surpassing *Haggis* and ChatGPT by 22.8% and 35.5%, respectively, in terms of Idiom Set Precision (ISP). Additionally, our approach excels in Idiom Coverage (IC), outperforming baselines by 9.7% and 22.9% when extracting idioms from libraries. IdioMine's performance is impressive: the size of the extracted idioms is nearly twice that of the baselines, indicating the ability of our approach to identify both specific and generic idioms in a more complete manner. Moreover, IdioMine's extracted idioms outperform the baselines in terms of code completeness and semantic clarity, with an average improvement of 0.22 and 0.19, respectively. In conclusion, our study makes the following primary contributions:

(1) We introduce a novel approach, called IdioMine, for automatically extracting both dataset-specific and generic idioms from Java code. The replication package is available at [12].
(2) We devise DCC, a chain structure, that enables us to extract sub-idioms with contiguous semantic meanings from PDGs.
(3) We conduct well-designed experiments to evaluate the effectiveness of IdioMine and design a user study to validate that IdioMine's extracted idioms exhibit superior quality in comparison to the baselines.
(4) Notably, we examine the primary functionalities of idioms extracted from every Java project and library using IdioMine through both quantitative and qualitative analysis.

2 PROBLEM DEFINITION

Definition of Code Idioms. A *code idiom* is a code fragment that demonstrates two key characteristics. Firstly, it exhibits recurring patterns across multiple software projects [2]. Secondly, it serves a singular semantic purpose [36]. Code idioms thus can encompass non-contiguous lines of code, exemplified by Listing 1, and involve intricate data and control flow, as demonstrated in Listing 2. Considering the formalization of code idioms in the prior study [36], which primarily emphasizes their syntactic structures while overlooking the completeness of semantic aspects, we propose an extension to enhance the original version. Specifically, we define idioms as sets of interconnected nodes, $\mathcal{I} = \{V, E\}$, by data or control flow, $E = \{E_d, E_c\}$, in PDGs, representing the completed semantic and syntactic structure of idioms.

What Idioms are Not. 1) *Simple APIs or method invocations are not code idioms.* The previous study [36] illustrated the difference between code idiom mining and API mining and clearly stated that simple APIs or method invocations (e.g., `parcel.writeDouble($Var);`) are not code idioms due to limited syntactic structure. However, their tool, *Haggis*, identified some common API usages and method invocation patterns with richer syntactic structures in the dataset as code idioms, such as "`Location.distanceBetween($(Location).getLatitude(), $...);`" and "`Toast.makeText(this, $stringLit, Toast.LENGTH_SHORT).show();`". Actually, certain API-based usages can be regarded as code idioms. Because when developers consistently employ a particular API in a specific manner to achieve a desired functionality or implement a common design pattern, it can be recognized as an API-based code idiom. Hence, our study abides by this rule to not regard single APIs or method invocations as idioms.

2) *While one-line statements can be classified as code idioms, simple return statements and certain types of assignment statements with limited semantic significance are excluded from this categorization.* The previous study [36] acknowledged one-line statements as code idioms, exemplified by the top idiom "`Elements $name=$(Element).select($StringLit);`" identified by *Haggis*. However, to enhance the mining of more meaningful idioms, our study chooses not to classify simple return statements (i.e., `return $var;`) and two specific types of assignment statements (i.e., `this.$Var = Var;` and `super.$Var = Var;`) as code idioms.

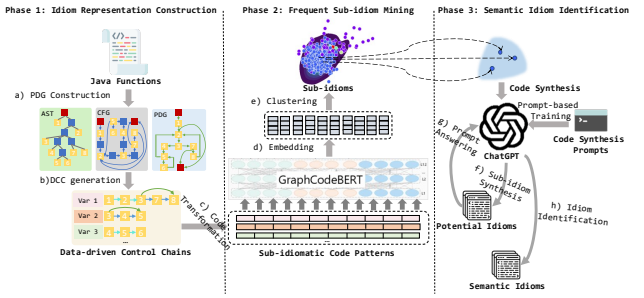

Figure 1: The overall framework of IdioMine.

3 IDIOMINE

In this section, we introduce the technical framework of our proposed approach, IdioMine. Specifically, we provide an overview of the framework and then explain each phase.

3.1 Overall Framework

Our approach, IdioMine, endeavors to extract code idioms within software projects. The framework of IdioMine, shown in Fig. 1, has three phases: 1) idiom representation construction, 2) frequent sub-idiom mining, and 3) semantic idiom identification. In phase one, we conduct program analysis to construct a PDG for each Java function, which comprehensively integrates the data flow and control flow of Java functions. Additionally, we develop a chain structure, DCC, which extracts code fragments possessing contiguous semantic meanings from PDGs as sub-idioms. In phase two, we encode sub-idioms using GraphCodeBERT and utilize density-based clustering to identify frequently occurring sub-idioms. Finally, we identify interrelated sub-idioms based on specific rules and employ ChatGPT to synthesize them as potential code idioms. Using ChatGPT's powerful capability [37], we can accurately identify real idioms from the pool of candidates. In the following sections, we will present a step-by-step explanation to facilitate readers' understanding.

3.2 Phase 1: Idiom Representation Construction

Existing methods cannot handle complex idioms made up of ❶ non-contiguous lines of code and ❷ intricate data and control flow. This is because they place a strong emphasis on code syntax during program analysis, disregarding the semantic connections and data flow between lines of code. To overcome these limitations, we employ

data flow analysis to extract semantically related code fragments from Java functions. We integrate the variables' data flow into the CFG to create the PDG, which comprehensively combines control flow and data flow, facilitating us to process complex code with ease. Next, we propose a chain structure, Data-driven Control Chains (DCCs), to obtain code lines with semantic relationships and regard them as sub-idiom. Fig. 2 provides an example that illustrates the complete process of constructing the PDG and generating DCCs. We will delve into the specifics below.

3.2.1 PDG Construction. Our intuition suggests that a code fragment's semantics is shaped by data exchanges of variables within a function. To capture this, we extract data flows of variables as semantic information of a function. A variable's data flow can be characterized by two types of usage: writing and reading. Following the Def-Use Chain definition[32], our study uses `Def` to represent writes and `Use` for reads. We merge these variables' data flow (i.e., `Defs` and `Uses`) into the CFG to build the PDG, preserving the control information from the CFG.

Specifically, we begin by generating ASTs and CFGs from Java functions. We use Javalang [13] to parse the source code and extract the AST. As code idioms typically are composed of sequential statements (yellow nodes) and control statements (brown nodes), we only retain necessary nodes at the expression level and beyond, including nodes at the expression, statement, and block levels to ensure the conciseness of ASTs. Since the PDG is based on the CFG, our next step is to extract the control flows from the AST and use them to build the corresponding CFG. The control structure, such as conditional branches and loops, enables the classification of execution paths between AST nodes into two distinct categories: `MustExe` (blue dotted edges) and `MayExe` (red dotted edges). `MustExe` edges represent paths that will definitely be executed, while `MayExe` edges represent paths that will be executed under specific conditions. We depict the code example's control flow with AST's `MustExe` and `MayExe`, and the resulting CFG is located adjacent to AST in Fig. 2.

To construct a concise PDG, it is necessary to simplify the corresponding CFG by excluding nodes (hollow nodes) that do not provide specific code of Java functions. This is because these nodes do not convey any semantic information about program variables. Next, the specific code represented by each node in the CFG is examined to capture the `Defs` and `Uses` of variables used in the node. We formalize the usage (i.e., `Def` and `Use`) of these variables into `<Def/Use->VarName: VarType>` and serve these usages as the semantic information required for constructing the concise PDG. For example, the first CFG node in Fig. 2 declares a variable named `file`. The usage of this variable can be denoted as `<Def->file: File>`, where `Def` represents the creation (write) operation of the variable `file`, and `File` denotes the variable's type. Therefore, the data flow in the first CFG node can be effectively symbolized by the usage of the variable `file`. In a similar fashion, the `For Loop` node utilizes the `file` variable in its loop condition, thus representing the semantic information of `file` as `<Use->file: File>`. We afterward integrate semantic information of variables into the concise version of the CFG to build the PDG.

3.2.2 DCC Generation. We develop a novel chain structure, called Data-driven Control Chain (DCC), to capture semantic relationships between code lines from the PDG. Each DCC captures the semantic information of one variable by connecting a variable's `Defs` and `Uses` along one possible execution path of that variable. Hence, the nodes in a DCC can represent non-contiguous lines of code within a function. Since functions typically involve data exchanges between multiple variables, and the same variable may produce several data flows due to differences in execution paths, it is possible to extract multiple DCCs of one variable from a single PDG and produce numerous short code fragments involving data interactions with a particular variable. These short code fragments have the potential to become idioms or essential building blocks of a code idiom. Hence, we consider these DCCs as sub-idioms.

Fig. 2 explains the DCC extraction process for `file`. Node 1 in the PDG defines the variable `file`, which is subsequently utilized in Nodes 2, 3, 4, and 8. Hence, the initial DCC for `file` comprises {*Node* 1, *Node* 2, *Node* 3, *Node* 4, *Node* 8}, signifying the variable's `Defs` and `Uses`. Among these, Nodes 2 and 4 depict control statements and offer the necessary control information for DCCs of `file`, whereas the other nodes represent sequential statements that use `file`. However, merely connecting the nodes that define and use the same variable in sequence is insufficient to obtain a complete DCC for that variable, as it could lead to a loss of control information or semantic meanings. To build a comprehensive DCC for a variable, it is crucial to integrate certain essential nodes into the initial DCC. These necessary nodes typically represent the requisite conditions or code preceding the execution of `Def` and `Use` nodes within a single execution path of said variable. For instance, the data flow for the initial DCC from Node 4 to Node 8 passes through Node 6, a control structure that must be executed before Node 8. To ensure a seamless data flow within this sequence, Node 6 should be included in the initial DCC for the variable `file`. Therefore, as depicted in Fig. 2, one of the final DCCs for the variable `file` is {*Node* 1, *Node* 2, *Node* 3, *Node* 4, *Node* 6, *Node* 8}. There is another possible execution path for the variable `file` that does not activate the catch block, this results in Nodes 6 and 8 being skipped. As a result, another DCC for `file` is {*Node* 1, *Node* 2, *Node* 3, *Node* 4}. Fig. 2 also presents the DCCs for the `br` and `line` variables, which contain the corresponding data and control flows.

3.3 Phase 2: Frequent Sub-idiom Mining

The code fragments, which are represented by a DCC and encompass the control flow and data flow of a single variable, have the potential to be a code idiom or a component of one. Hence, those code fragments could be regarded as a sub-idiom. To ensure that the idioms extracted by our approach adhere to the definition of code idioms – code patterns that frequently occur within functions across software projects [2] – we qualify frequent sub-idioms that occur in datasets multiple times. Given that a single sub-idiom can be part of different complete idioms, we consider sub-idioms occurring at least twice in our datasets as frequent ones to maximize IdioMine's ability to extract both generic and real idioms. We achieve this by converting DCCs into specific code fragments, embedding and clustering them to discover frequent sub-idioms.

3.3.1 Code Transformation. Our initial step towards obtaining frequent sub-idioms is to generate the code fragments represented by DCCs. We do this by mapping each DCC node to its corresponding node in the CFG and extracting the relevant code indicated by that

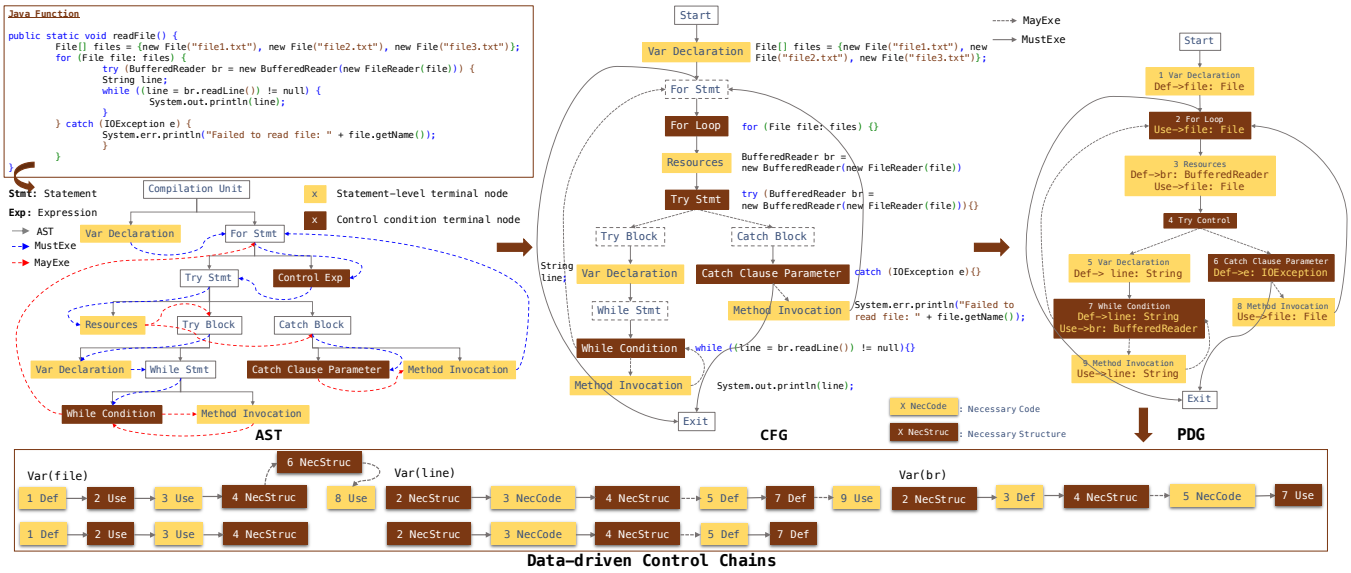

Figure 2: An example of constructing a concise PDG from a Java function and generating DCCs from it.

CFG node. These code lines are then assembled to form a sub-idiom. To enhance comprehension of this step, Fig. 3 furnishes an example to illustrate the formation of the sub-idiom for the first DCC of the variable `file`. All nodes, with the exception of Node 6, utilize the variable `file`. Despite this, the control structure of Node 6 should be incorporated into the sub-idiom as it must be executed prior to the final output statement at Node 8. Moreover, if there are no relevant statements in a particular code block, such as the try block in the example given, it is acceptable for that block to be empty within sub-idioms.

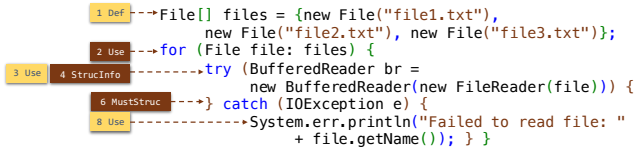

Figure 3: One of the DCCs for the variable `file`.

3.3.2 Sub-idiom Embedding. This step serves as a preparation for mining frequent sub-idioms, with the goal of transforming them into code embeddings. Due to the difficulty in constructing labeled idioms for code representation, employing a pre-training model to embed sub-idioms in an unsupervised manner is a suitable solution for this challenge. Therefore, we take into consideration a high-regard pre-training model, i.e., GraphCodeBERT [46], that is used for code representation and incorporates the structural information and data flow of code. This feature aligns with IdioMine's overall design, enhancing its suitability for sub-idiom representation.

3.3.3 Sub-idiom Clustering. To identify frequent sub-idioms, we use a density-based clustering algorithm, DBSCAN [42], which automatically identifies the number of clusters without prior specification. We employ the *Parameter Tuning* technique [14] to optimize clustering performance. This optimal technique involves iteratively adjusting parameter values while we achieve an optimal Silhouette Score metric [15]. Based on the results, we identify the sub-idioms represented by the cluster centroids as frequent ones since they encompass the fundamental features of their respective clusters.

3.4 Phase 3: Semantic Idiom Identification

A frequent sub-idiom may be deemed a complete code idiom on occasion, but most idioms entail the data flow of multiple variables, necessitating the merging of several frequent sub-idioms. To spot these intricate idioms, two vital steps are required: a) fusing multiple related sub-idioms into a potential code idiom (PCI) and b) distinguishing real idioms from synthesized potential idioms. Both of these steps present significant challenges. For instance, synthesizing sub-idioms can be problematic as it is difficult to ❶ determine which sub-idioms and ❷ decide how many sub-idioms should be fused and ❸ ensure each statement or code block is correctly incorporated in its designated location. Furthermore, identifying real idioms from a vast of synthesized potential idioms is a crucial problem in the second step. Addressing these challenges would necessitate human intelligence to manually synthesize sub-idioms and identify real idioms from them, which could lead to a significant increase in operational costs.

Fortunately, the recent unveiling of ChatGPT [16] has substantially augmented the cognitive prowess of large language models (LLMs), which showcases their potential to match and even surpass human intelligence [17]. Hence, we would like to use ChatGPT to: 1) synthesize interrelated frequent sub-idioms as PCIs and 2) distinguish real idioms from PCIs. We utilize ChatGPT through its open-source API [18], which is based on the GPT-3.5 architecture.

3.4.1 Frequent Sub-idiom Synthesis. We first design a rule-based sub-idiom selection strategy that facilitates us to choose the most suitable ones for synthesis. This strategy consists of two rules: a) The related sub-idioms to be synthesized must involve the shared variable, ensuring the consistency of the data flow in the synthesized code. b) The sub-idioms selected for synthesis must belong to the same Java function in the same source file, ensuring the correctness and realism of the data flow in the synthesized code. We introduce a parameter k to indicate the number of selected sub-idioms that adhere to the aforementioned rules for code synthesis.

Once we select the appropriate sub-idioms for synthesis, we then use ChatGPT for sub-idiom synthesis. We design a suitable prompt to ensure the accurate synthesis of sub-idioms. The initial step in crafting an effective prompt is to establish the task objective. Our objective is to input k sub-idioms ($k \geq 1$), denoted as $\{\texttt{SI}_1, \texttt{SI}_2, \cdots, \texttt{SI}_k\}$, and require a function f that can accurately synthesize multiple sub-idioms into a potential code idiom, PCI, with both correct syntax and accurate semantic meaning. With this definition, we establish the format for the input and output of this task: the input comprises k separate code segments, and the output is a synthesized code segment. Due to a fixed input and output format, we opt to employ the *discrete prompt* (i.e., *hard prompt*) [52] as the template prompt for our task. Thus, our prompt is as follows: *"The following code fragments, [$\texttt{SI}_1$], [$\texttt{SI}_2$], $\cdots$, [$\texttt{SI}_k$] can be integrated into the reasonable code [PCI]".*

To optimize the performance, we create specific typical examples by leveraging our template prompt as a foundation to form a demonstration context and utilize them to instruct ChatGPT to generate proper outputs [41, 52]. This serves two primary purposes: a) It enables the used LLM to comprehend our template prompt. b) It facilitates the used LLM's acquisition of potential rules for sub-idiom synthesis from specific prompts, which can enhance the model's performance, particularly in few-shot scenarios. To generate effective prompts for training, we conduct an analysis and categorization of potential actions that may arise during sub-idiom synthesis. These actions fall into two distinct categories: a) the concatenation action (SCA), which involves incorporating the respective sequential statements of sub-idioms in sequence, following the order of the shared variable's data flow among the sub-idioms. b) The insertion action (IA), which involves inserting one or more sequential statements from one sub-idiom into the appropriate position of a certain control structure of synthesized code, based on the data flow of the shared variable. We thus generate specific prompts that encompass both SCA, IA, and a combination of both. Finally, we utilize the fine-tuned ChatGPT to synthesize selected sub-idioms to obtain potential code idioms.

3.4.2 Code Idiom Identification. We employ the technique called *prompt ensembling* [52] to enhance the ability of ChatGPT to identify code idioms. This involves utilizing a series of questions as prompts to guide the used LLM to generate reliable results [52]. Based on the definition of code idioms, which refers to patterns with both high semantic clarity and commonly used in source code, we develop three distinct questions as prompts. These three prompts and their workflow are presented in Fig. 4. To be precise, our approach involves inputting a potential code idiom (PCI) into the model and guiding it to assess the PCI's generality and semantic clarity using the first two distinct prompts, PR 1 and PR 2. Based on the responses to both prompts, we input the final prompt into ChatGPT to determine whether the input PCI qualifies as a code idiom or not. The final output will be either "Yes" or "No" to ensure accurate identification of genuine idioms.

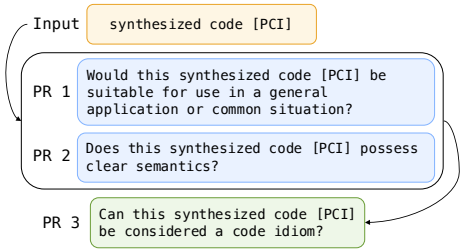

Figure 4: Designed prompts used for idiom identification.

Table 1: Details for two evaluation datasets.

	Projects	Stars	Description
Project	arduino	13.5k	Electronics Prototyping
	atmosphere	3.6k	WebSocket Framework
	bigbluebutton	7.9k	Web Conferencing
	elasticsearch	62.3k	REST Search Engine
	grails-core	2.7k	Web App Framework
	hadoop	13.4k	Map-Reduce Framework
	hibernate	5.3k	ORM Framework
	libgdx	21.3k	Game Dev Framework
	netty	31k	Net App Framework
	storm	6.4k	Distributed Computation
	vert.x	13.5k	Application Platform
	voldemort	2.6k	NoSQL Database
	wildfly	2.8k	Application Server
Library	android.location		Android Location API
	android.net.wifi		Android WiFi API
	com.rabbitmq		Messaging System
	com.spatial4j		Geospatial Library
	io.netty		Network App Framework
	opennlp		NLP Tool
	org.apache.hadoop		Map-Reduce Framework
	org.apache.lucene		Search Server
	org.elasticsearch		REST Search Engine
	org.eclipse.jgit		Git Implementation
	org.hibernate		Persistence Framework
	org.jsoup		HTML Parser
	org.mozilla.javascript		JavaScript implementation
	org.neo4j		Graph Database
	twitter4j		Twitter API

4 EVALUATION

This section outlines the techniques and datasets used to evaluate the effectiveness of IdioMine. It also includes details on our experimental settings.

4.1 Baselines

After analyzing existing techniques for mining idioms, we select a widely used, well-recognized, and reproducible approach called *Haggis* as our baseline. *Haggis* is the first model proposed in [36] for extracting Java idioms. This framework serves as a foundation for other tools, such as the latest work [64], and is a popular choice for research. The latest work experimented on Hack [31], a niche PL developed by Facebook, and its applicability to other languages is unknown due to the unavailability of the tool for replication. We thus select *Haggis* as a baseline due to no other good options available. Similarly, the latest work selected only one idiom extractor, i.e., *Haggis*, as the baseline for the same reason. To address this issue, we also conduct a comparison experiment between IdioMine and ChatGPT in Section 6.

4.2 Dataset

We capitalize on the widespread use of idioms in source code to assess the performance of IdioMine. In our study, we utilize the same set of Java projects and libraries used in *Haggis* [36] as our two evaluation datasets. The information of the two datasets is explicated in Table 1. The Project dataset serves the purpose of identifying project-specific idioms, whereas the Library dataset is utilized for extracting library-specific idioms. Idioms extracted from the Library dataset can be regarded as cross-project code

idioms due to their versatility in various projects. The projects in the Project dataset are widely recognized in the community and have earned significant popularity owing to their high-quality code and utility. The same as the method used in [36], we build the Library dataset by extracting source files from the Java GitHub Corpus [35] that import and use the 15 popular libraries but do not implement them. Following the data segmentation method used in [36], both datasets are split into a 70% training set and a 30% testing set.

4.3 Experimental Settings

We implemented IdioMine using Pytorch[19]. Our experiments were conducted on Ubuntu v20.04.1 64-bit OS with an RTX3090-24GB GPU, ensuring reliable and replicable results. The replication package has extra model settings info accessible.

5 RESULTS

To better understand IdioMine's performance and the usefulness of extracted code idioms, we analyze our evaluation results by addressing four research questions (RQs):

(1) RQ1: How effective is IdioMine?
(2) RQ2: How effective is IdioMine in code synthesis?
(3) RQ3: Are extracted idioms of high quality?
(4) RQ4: What are the characteristics of extracted idioms?

5.1 RQ1: How effective is IdioMine?

5.1.1 Motivation. In this RQ, we conduct experiments to evaluate IdioMine's effectiveness in idiom extraction and compare its performance to the baseline.

5.1.2 Method. We utilize both IdioMine and *Haggis* to extract code idioms from two evaluation datasets and evaluate their performance, respectively. During experiments with IdioMine, we vary the parameter k, which determines the maximum number of sub-idioms that can be synthesized, to 1, 2, and 3. This is based on our observation that idioms typically consist of three or fewer sub-idioms. Additionally, we prioritize regarding a code fragment as a complete function rather than an idiom with a single purpose when increasing k results in excessive code complexity. To ensure a fair comparison with *Haggis*, we set two parameters of *Haggis*, n_{min} and C_{min}, to the same values used in their study [36].

5.1.3 Evaluation Metrics. We adhere to evaluation metrics proposed in the prior research [36], i.e, *idiom coverage (IC)* and *idiom set precision (ISP)*, for comparison experiments. *IC* measures the percentage of AST nodes in a Java function that correspond to any of the identified code idioms, while *ISP* measures the proportion of idioms in the training corpus that are also found in the test corpus. *IC* assesses how well an idiom mining technique considers the full semantic meaning of a function when generating code idioms. It's a real number between 0 and 1. *ISP* assesses the frequency of identified code idioms in projects. Both metrics are tailored for code idiom mining and are similar to precision and recall in information retrieval, but adjusted for the specific domain of code idioms. We also use an additional metric, the average size of the extracted idioms (*Avg Size*), which calculates the average number of AST nodes contained within extracted idioms.

Table 2: The performance of *Haggis* and IdioMine.

	Approach	IC(%)		ISP(%)		Avg Size(#Nodes)	
Project	$Haggis_1$	30.3	± 12.5	14.4	± 9.4	15.5	± 3.1
	$Haggis_2$	3.1	± 2.6	29.9	± 19.4	25.3	± 3.5
	$Haggis_{avg}$	*16.7*	–	*22.2*	–	*20.4*	–
	$IdioMine_1$	29.2	± 11.2	42.4	± 21.3	39.5	± 24.9
	$IdioMine_2$	32.1	± 12.4	46.3	± 23.5	45.7	± 27.8
	$IdioMine_3$	34.2	± 13.5	50.8	± 25.6	51.5	± 31.7
	$IdioMine_{avg}$	*31.8*	–	*46.5*	–	*45.6*	–
Library	$Haggis_1$	23.5	± 13.2	8.5	± 3.2	15.0	± 2.1
	$Haggis_2$	2.8	± 3.0	16.9	± 10.1	27.9	± 8.6
	$Haggis_{avg}$	*13.2*	–	*12.7*	–	*21.5*	–
	$IdioMine_1$	17.7	± 5.6	31.8	± 24.0	44.7	± 27.5
	$IdioMine_2$	23.3	± 9.9	34.7	± 26.1	53.4	± 30.3
	$IdioMine_3$	27.6	± 13.4	39.9	± 27.9	58.6	± 34.0
	$IdioMine_{avg}$	*22.9*	–	*35.5*	–	*52.2*	–

* n_{min} and c_{min} for $Haggis_1$ and $Haggis_2$ are 5/2 and 20/25, respectively.

5.1.4 Results. Table 2 allows us to draw the following conclusions:

a) **Compared to *Haggis*, IdioMine extracts code idioms with more complete semantic meanings.** Specifically, as evidenced by Table 2, IdioMine exhibits a higher *IC* value than baseline, with an average of 15.1% and 9.7% on the Project and Library, respectively. Furthermore, on average, the number of nodes in IdioMine's extracted idioms on the Project dataset is over double that of the baseline, and almost triple on the Library dataset. This indicates that IdioMine's extracted idioms can cover a broader range of code within a function and offer more semantic information than *Haggis*.

b) IdioMine **can not only extract commonly used idioms but also identify infrequently occurring idioms in datasets but with a broad generality.** As demonstrated in Table 2, our approach outperforms the baseline in terms of ISP values for all k values. This clearly demonstrates that the idioms mined by IdioMine adhere to the definition of code idioms, which is characterized by being frequently used in source code. However, upon manual inspection of IdioMine's extracted idioms, we observe a subset of them with generic semantics despite having fewer occurrences in the dataset. As illustrated in the third motivating example of Section 1, an idiom extracted from the library "*org.neo4j*" by IdioMine appears in only two out of the 2,357 Java functions across 1,294 source files. Despite its infrequency, the idiom represents a well-known and generic code pattern called the "*Command Pattern*". This pattern can occur in various projects underscores its versatility, which verifies the capability of IdioMine in identifying infrequently occurring in our datasets, but generic idioms.

c) **Compared to the baseline,** IdioMine **demonstrates a superior capability in identifying idioms with intricate syntax.** The *Avg Size* metric in Table 2 indicates that IdioMine's extracted idioms contain a greater number of AST nodes compared to those of the baseline. This illustrates that the idioms extracted by IdioMine are more complex, possibly owing to their distinctive capability to identify non-continuous and semantically linked code lines.

5.2 RQ2: How effective is IdioMine in code synthesis?

5.2.1 Motivation. IdioMine leverages ChatGPT to generate potential code idioms by synthesizing interrelated sub-idioms. However, despite ChatGPT's proficiency in various domains, it may still generate erroneous potential code idioms, which can impact its overall effectiveness in identifying code idioms to some extent.

Table 3: The performance of IdioMine in code synthesis.

P(%)	R(%)	F1(%)
93.6	97.9	95.7

Hence, we conduct experiments to evaluate IdioMine's performance during the sub-idiom synthesis phase.

5.2.2 Method. We select the *hibernate* project as the dataset since it is the largest project in the Project, comprising the highest number of source files. We do not select any libraries from the Library for sub-idiom synthesis evaluation, as most sub-idioms utilizing the same library typically originate from different functions, source files, or even projects, rendering them ineligible to meet the rules for code synthesis.

To ensure the accuracy of our evaluation, we engage two experienced Java developers with more than seven years of Java development expertise to create a ground-truth dataset. First, they independently identify sub-idioms to be synthesized based on the proposed rules, and then manually synthesize them to generate sensible code fragments. Subsequently, they discuss their results together to resolve any disagreements. We assess their inter-rater agreement using Cohen's Kappa [56], a widely used metric [68, 69] and obtain a robust value of 0.83, indicating a strong level of agreement. In total, they cost two weeks to synthesize 48 potential code idioms from the 776 sub-idioms in the *hibernate* project.

5.2.3 Evaluation Metrics. We use three commonly used metrics, *precision (P)*, *recall (R)*, and *F1-score (F1)*, to evaluate IdioMine's performance in sub-idiom synthesis. *Precision (P)* refers to the percentage of correctly identified idioms, while *recall (R)* indicates the percentage of identified idioms out of those manually identified. We only evaluate IdioMine's performance in this phase, as *Haggis* does not involve sub-idiom synthesis.

5.2.4 Results. Conclusions can be drawn from Table 3:

a) IdioMine **performs well in sub-idiom synthesis, which is essential for accurate code idiom identification.** As demonstrated in Table 3, with the help of ChatGPT, IdioMine can synthesize code fragments with a high degree of accuracy and correct syntax. It achieves impressive Recall and F1 scores of 97.9% and 95.7%, which lays a solid foundation for IdioMine's outstanding performance in idiom extraction. Based on our analysis, we have observed that the strong performance of ChatGPT in this phase can be attributed to our utilization of its exceptional code understanding capabilities, coupled with our strategic approach to mitigate its highly powerful generation capability. By constraining ChatGPT to synthesize code fragments based on our provided input, we effectively leverage its strengths while preventing the generation of potentially incorrect code that may appear plausible.

Failure analysis. During the code synthesis phase, there are still three errors and one omission that persists. Upon manual inspection, we notice that IdioMine struggles with synthesizing sub-idioms that feature complex syntax, such as ❶ the sub-idiom to be synthesized involving multiple control structures or ❷ sub-idioms whose statement order is challenging to determine. Furthermore, we observe one sub-idiom pair which IdioMine fails to synthesize since they do not possess any shared variables. However, our developers are confident that these sub-idioms should be synthesized as they are located within the same exception-handling block. Overall, IdioMine has proven to be effective in identifying code idioms during the sub-idiom synthesis phase.

Table 4: The quality performance of the extracted idioms.

Approach	Code Completeness	Semantic Clarity	Generality
Haggis	3.65 ± **0.80**	3.68 ± 0.85	3.23 ± 0.92
IdioMine	**3.95** ± **0.80**	**3.91** ± **0.82**	**3.41** ± **0.82**

5.3 RQ3: Are extracted idioms of high quality?

5.3.1 Motivation. This RQ aims to ascertain the quality of IdioMine's extracted idioms through human analysis.

5.3.2 Method. According to the definition of the idiom, we evaluate the quality of extracted code idioms by considering three aspects, i.e., code completeness, semantic clarity, and generality. To achieve this, we design a user study with the aim of investigating the preferences in those three aspects of experienced Java practitioners towards the top idioms extracted by the baseline and IdioMine.

We investigate the top 18 most common code idioms in our user study extracted by different approaches. Those top idioms are the ones with the highest *ISP* extracted from the Library dataset. *This is for two reasons:* 1) The idioms extracted from the Library dataset are cross-project in nature and have a greater degree of universality compared to those found in the Project dataset. 2) The top 18 idioms extracted from the Library dataset by *Haggis* are available in [36], ensuring consistency of idioms used in our study as well as in others. This enhances the accuracy and rigor of our study and provides a standardized reference for future studies.

11 highly experienced Java developers with over seven years of experience are invited to our user study. They independently evaluate each of the top idioms in terms of code completeness, semantic clarity, and generality on a scale of 1 to 5. A score of 1 indicates poor performance while a score of 5 indicates excellent performance. We will analyze the results to evaluate the quality and practicality of the top idioms extracted by *Haggis* and IdioMine. This study is approved by the Institutional Review Board (IRB).

5.3.3 Results. Conclusions can be inferred from Table 4:

a) **Compared to *Haggis*, IdioMine's extracted idioms perform well in code completeness, semantic clarity, and generality.** Table 4 shows that IdioMine's extracted idioms score 3.95, 3.91, and 3.41, respectively, outperforming the baseline by 0.30, 0.23, and 0.17 in code completeness, semantic clarity, and generality. This is likely due to IdioMine's ability to extract idioms from non-contiguous code lines that possess semantic relationships and the ability to recognize generic idioms. Therefore, the results suggest that IdioMine's extracted idioms are complete, easy to understand, and generality, offering greater practical value than the baseline.

b) IdioMine**'s extracted idioms achieve lower standard deviations among all three metrics.** In Table 4, the standard deviations of IdioMine's idioms exhibit a decrease of 3% and 10% in semantic clarity and generality, respectively, as compared to those of *Haggis*. Besides, it shows an equivalent standard deviation in code completeness as that of the baseline. These demonstrate that IdioMine's idioms consistently outperform the baseline in terms of quality performance and display lower volatility across all metrics.

Table 5: Analysis results on idioms extracted from the Project and Library datasets by IdioMine.

Project/Library	#Total	#API	#CS	#WP	Code Fragment		Top Three Functionalities (#Num)		
					#Generic	#Specific			
arduino	34	1	10	1	18	4	GUI Programming (16)	Input Text Processing (5)	Preference Setting (3)
atmosphere	39	10	10	-	15	4	Data processing(6)	WebSocket Communication (5)	Configuration Management (3)
bigbluebutton	46	6	10	2	20	8	Web Event Programing (9)	Message Processing (7)	RedisConnection Operation (5)
elasticsearch	11	1	3	-	4	3	Configuration Management (5)	Information Retrieval (4)	Search Result Processing (3)
grails-core	32	5	3	-	18	6	Network Cmomunication (4)	Configuration Management (4)	URL Processing (3)
hadoop	29	5	8	-	15	1	Web Service (4)	Resource Management (3)	Hadoop YARNApplication (3)
hibernate	31	3	3	1	21	3	I/O Operation (10)	Configuration Management (8)	Event-driven Programing (5)
libgdx	45	4	15	-	17	9	GUI programming (8)	Data Processing (4)	Network Communication (3)
netty	26	7	5	-	11	3	I/O Operation (12)	Net Communication (4)	Memory Operation (4)
storm	75	6	24	1	34	10	Real-time Operation (9)	Authentication&Authorization (4)	Network Communication (3)
vert.x	57	9	6	-	36	6	Network Communication (11)	Data Processing (8)	Command-line Input (5)
voldemort	32	7	12	-	10	3	Network Communication (5)	Input Validation (5)	Data Processing (3)
wildfly	33	19	2	-	11	1	XML Configuration (5)	Resource Registering (4)	DeploymentUnit Operation (3)
Project	490	83	111	5	230	61			
android.location	39	15	5	-	14	5	Location-based Operation (11)	UI Element Processing (10)	Preference Setting (3)
android.net.wifi	33	8	5	-	14	6	WiFi Network Management (11)	Information Retrival (8)	Preference Setting (2)
com.rabbitmq	98	2	17	2	54	23	Message Processing (14)	Network Communication (13)	Timestamp-related Operation (5)
com.spatial4j	54	10	8	-	21	15	Data Processing (13)	Graphics-related Operation (13)	Distance Calculation (3)
io.netty	155	21	33	1	48	52	Network Communication (8)	Configuration Management (4)	Web Service (4)
opennlp	52	10	13	-	26	3	ML Implementation (8)	Modle Parameter Processing (4)	Natual Language Processing (4)
org.apache.hadoop	48	17	4	-	26	1	(large-scale) data processing (19)	Configuration Management (12)	File Cache (3)
org.apache.lucene	51	9	5	-	32	5	Lucene Search Processing (23)	Search Result Processing (5)	Data Tokenizing (2)
org.elasticsearch	35	8	3	-	21	3	Elasticsearch Search Processing (8)	Search Result Processing (4)	Data Tokenizing (2)
org.eclipse.jgit	69	9	10	1	37	12	Git Command Excution (42)	Git Connection (8)	Information Retrieval (6)
org.hibernate	177	15	27	-	83	52	SQL Operation (50)	Data Processing (27)	Configuration Management (14)
org.jsoup	28	6	6	-	10	6	HTML Scraping&Parsing (8)	Network Connection (3)	Data Processing (3)
org.mozilla.javascript	35	10	2	-	18	5	Javascript Operation (11)	Configuration Management (4)	Atom Feeds&Publishing (2)
org.neo4j	36	11	3	1	15	6	Graph Dataset Operation (18)	I/O operation (6)	Configuration Management (3)
twitter4j	100	4	12	4	54	26	Twitter API-elated (16)	UI Element Processing (13)	Drool-based Application (3)
Library	1,010	155	153	9	473	220			

5.4 RQ4: What are the characteristics of the extracted idioms?

5.4.1 Motivation. This RQ aims to systematically investigate IdioMine's extracted idioms to explore their classification and primary functionalities.

5.4.2 Method. To accomplish our objective, we perform a comprehensive analysis of the idioms extracted from two evaluation datasets utilizing IdioMine. Our analysis comprises two phases. In the first phase, we classify idioms into five categories based on their formations, including: a) API – an API usage recognized as a code idiom, b) Code Skeleton (CS) – a code structure considered as a code idiom, c) Well-known Pattern (WP) – an idiom formed by a well-known design/code pattern, d) Generic idiom – an idiom that can reoccur and be reused in multiple projects, and e) Specific idiom – an idiom that requires significant modifications when reoccurring in other usages. After categorizing idioms, we quantitatively analyze the distribution of idioms in each type.

In the second phase, we undertake a qualitative analysis to investigate the main functionalities of idioms extracted from each project and library. To ensure the correctness of the conclusions drawn from this RQ, two experienced Java developers with over seven years of experience are involved in the classification and semantic analysis of code idioms. First, they independently label the semantic of these idioms based on their experience and ChatGPT's answers. Next, they independently derive the top three functionalities by generalizing idioms extracted from each project and library. They collaborate to reach a consensus in case of discrepancies and integrate their results into a final version that serves as the ground truth. Similar to RQ1, Cohen's Kappa [56] is used to measure the agreement between them, achieving values of 0.88 and 0.80 in the two phases and indicating high concordance. Note that we focus on key features of idioms and exclude less important and too generic functions, such as exception handling (empty code structure) and collection operations (e.g., element iteration).

5.4.3 Results. Drawing on the experimental results presented in Table 5, we arrive at the following conclusions:

a) **Idioms are more commonly extracted from the Library dataset compared to the Project dataset.** Table 5 reveals that 1,010 idioms are identified in the Library dataset, whereas the Project dataset yields only 490 idioms, which is less than half the number extracted from the Library dataset. Our insight into this observation is that the Library dataset primarily consists of source files that both import and use 15 different Java libraries. Consequently, the code in these source files tends to be more similar and idiomatic compared to that in the Project dataset, resulting in a higher number of extracted idioms from the Library dataset.

b) **Idioms from Library embody the primary functionalities of their datasets better than those from Project.** Table 5 highlights the top three functionalities of both projects and libraries. It is worth noting that some of these functionalities are generic and do not necessarily represent the main purpose of the projects or libraries. These generic functionalities encompass *Configuration Management, I/O Operation, Data Processing, Input Validation, Memory Operation, Input Text Processing, Command-line Input, and File Cache*. In the Project dataset, 85 out of 215 top idioms, which account for 39.5%, fall under these generic functionalities. On the other hand, only 108 out of 453 top idioms, which account for 23.8%, in the Library dataset are classified as generic functionalities. This difference can be attributed to the fact that the idioms

Table 6: The performance of ChatGPT in RQ1.

	Approach	IC(%)	ISP(%)	Avg Size(#Nodes)
Project	ChatGPT	31.7 ± 29.8	40.8 ± 49.6	16.4 ± 11.3
Library	ChatGPT	–	–	–

Table 7: The performance of ChatGPT in RQ3.

Approach	Code Completeness	Semantic Clarity	Generality
ChatGPT	3.81 ± **0.77**	3.77 ± **0.64**	**3.41** ± **0.63**

in the Library dataset summarize the common patterns in different projects' source files that import and use the selected libraries. Hence, these idioms better represent the libraries' main role and usages compared to those in the Project dataset.

c) IdioMine **can produce both dataset-specific and generic idioms.** As previously demonstrated, *Haggis* uses an ML-based statistical method to extract idioms from projects and libraries. However, these idioms are not easily transferable to other Java projects. Our approach, IdioMine, which not only performs program analysis on specific code but also integrates ChatGPT's capability can produce both specific and generic idioms. For example, Table 5 demonstrates that IdioMine generates 230 and 473 generic idioms, as well as 61 and 220 dataset-specific idioms from the Project and Library datasets, respectively.

d) IdioMine **extracts well-recognized design patterns from datasets, exhibiting its practicality and effectiveness.** Our approach successfully identifies five and nine widely-used design patterns from Project and Library datasets, respectively. These patterns are the *Dispose Pattern* [20], *Observer Pattern* [21], *Consumer-Producer Pattern* [22], *Builder Pattern* [23], and *Lazy Initialization* [24] for the Project dataset. Meanwhile, the Library dataset contains the *Observer Pattern* [21], *Wait-retry Pattern* [25], *Double-Checked Locking pattern* [26], *Lazy Initialization* [24], *Command Pattern* [11], *Factory Method Pattern* [27], *ViewHolder Pattern* [28], *Dynamic Dispatch* [29], and *Random Access* [30]. These patterns have a different scope of application. For example, some of them such as *Lazy Initialization* and *Factory Method Pattern* are not language-specific, while the *ViewHolder Pattern* is tailored to Android projects, used for providing efficient processing of collection elements without the need for lookup. Overall, the recognition of these well-established patterns highlights IdioMine's practicality and effectiveness.

6 DISCUSSION

Considering only one baseline, in this section, we select ChatGPT as an additional baseline and compare the performance of IdioMine and ChatGPT in addressing RQ1 and RQ3. Moreover, since IdioMine incorporates ChatGPT in Phase Three, this study can also be regarded as an *ablation study* aimed at *validating the necessity of program analysis in the first two phases.*

6.1 ChatGPT in RQ1

6.1.1 Method. We employ two prompts to facilitate ChatGPT in generating idioms from two evaluation datasets. PR 1 is "Generate as many code idioms as possible from the Java project named [`project name`]". PR 2 is "Generate as many code idioms as possible from the code that imports and uses the Java library [`library name`]".

6.1.2 Results. Conclusions can be drawn from Table 6:

a) **Without the program analysis phase, ChatGPT cannot perform well in idiom extraction.** ChatGPT performs poorly in code idiom extraction compared with IdioMine. In Table 6, ChatGPT's extracted idioms from Project are too brief and general, averaging only 16.4 AST nodes with the longest being 52 nodes. The idioms extracted from Library are even of low quality, describing only general structures without contextual meanings. For instance, ChatGPT identifies "*Using the try-catch statement to handle exceptions*", "*Creating and using objects from a class*", and "*Utilizing for loops for iterating over arrays and collections*" as code idioms. These code fragments are not idioms due to their lack of usefulness. Thus, program analysis for building DvCFGs and frequent sub-idioms is essential for idiom extraction.

b) **After upgrading** IdioMine **and ChatGPT, updated** IdioMine **can still achieve better performance than updated ChatGPT.** Although ChatGPT based on GPT-3.5 is unable to generate useful idioms from the Library dataset, we discover that manual input on the updated GPT-4 version on its website can produce satisfactory results. We thus compare our updated IdioMine, which is also upgraded to GPT-4, and find that while the updated ChatGPT can generate a limited number of generic idioms from the Library dataset, the quality decreases as the quantity increases. On the other hand, our updated IdioMine can generate a larger number of idioms, including both general and specific ones, outperforming both the previous IdioMine version and the updated ChatGPT.

6.2 ChatGPT in RQ3

6.2.1 Method. We follow RQ3's experimental method to assess the quality of ChatGPT's extracted idioms in code completeness, semantic clarity, and generality by a user study. Once ChatGPT generates fewer than 18 high-quality idioms from Library, we will supplement them with idioms having the highest *ISP* which ChatGPT generates from Project.

6.2.2 Results. Conclusions observed from Table 7 are:

a) **ChatGPT has the lowest standard deviation among the three metrics than** IdioMine **and *Haggis*.** ChatGPT demonstrates exceptional consistency with the lowest standard deviation across code completeness, semantic clarity, and generality, measuring at 0.77, 0.64, and 0.63. This is due to its built-in algorithm and stereotyped templates to generate idioms, resulting in minimal variations in the three metrics.

b) **ChatGPT's generated idioms show similar generality to** IdioMine**'s performance.** ChatGPT excels at producing idiomatic expressions with high generality, thanks to its impressive cognitive abilities and general intelligence that allow it to analyze source code in a great number of projects comprehensively. However, these idioms are often too broad to represent the main purpose and role of the generated dataset. Furthermore, ChatGPT's ability to generate idioms is limited, as it can only generate 49 idioms from the Project, representing merely 8.7% of the idioms extracted by IdioMine, and produces unqualified idioms from the Library.

7 THREATS TO VALIDITY

We identify the following threats to the validity of our study:

Internal validity. Our approach uses the parameter value k to determine the scale of a potential code idiom, indicating the number of sub-idioms that can be synthesized. Initially, in RQ1, we experimented with a range of 1 to 3 for k but later increased it to 5 to allow for more flexibility in parameter selection. Our experiments have shown that idioms synthesized from more than three sub-idioms are rare. Therefore, we recommend using the optimal range of 1 to 3 for k. To ensure accuracy, we tested IdioMine, but some errors may have occurred during its implementation. Furthermore, we have made an interesting observation during our study: ChatGPT frequently generates additional, unnecessary code content based on our provided inputs, ensuring the synthesis of complete code. A typical example of this occurrence is when the code fragments to be synthesized contain a try block; ChatGPT autonomously generates a void catch block to complement the try-catch structure. Indeed, these cases have no impact on the performance of our study. Besides, ChatGPT sometimes overlooks consolidating duplicated control blocks, such as try blocks, from different sub-idioms into a unified form during code synthesis. We have improved the corresponding prompt to impose restrictions on ChatGPT, thereby minimizing the generation of additional and duplicated code.

External validity. Although idioms extracted from the source code are often specific to the projects and libraries in the dataset and the selected PL, our approach, IdioMine, is better than existing methods to generate both specific and generic idioms using a small number of projects and libraries. To improve the practicality of IdioMine, we study Java due to its prevalence on GitHub, with a source code volume of 117 million. This surpasses other commonly used industry programming languages like C#, Python, Scala, and JavaScript. As ChatGPT's response varies with each of the N times it is called, we input the same prompt to ChatGPT five times and record all relevant answers after filtering duplicated ones.

Construct validity. The validity of the evaluation can be affected by human errors. We minimize such threats by electing well-experienced researchers and developers. While assessing the performance of IdioMine in terms of *IC* and *ISP*, we identify code fragments that share identical or similar structures with certain code idioms, particularly structural idioms (i.e., code skeletons), as idiomatic expressions. Note that during the development of IdioMine, GPT-3.5 API served as the most recent version available to us since GPT-4 API had not yet been released. As the version and ability of GPT improve, the extracted idioms may show some differences. In addition, access to the automatic API of ChatGPT is restricted to sponsors or paying customers of OpenAI.

8 RELATED WORK

We discussed idiom mining research by Allamanis et al.[34, 36] as the first work in the field, and the latest work by Sivaraman et al.[64] throughout the paper. As noted in Section 1, IdioMine overcomes three key limitations in their approaches. Merchante et al. [57] developed a web tool to mine Python idioms from GitHub by searching related tokens in Python code. Yang et al. [67] developed a method for identifying fix patterns in Python by extracting fine-grained bug-fixing code changes and clustering similar bug-fixing code changes using abstract syntax tree edit distance. Orlov et al. [60] introduced an idiom extraction algorithm based on subtree counting and information metrics to extract Python idioms. Existing idiom extraction techniques [40, 50, 61, 62] only identify commonly used idioms in their datasets but do not capture generic ones. Also, many ignored code fragment semantics to extract idioms.

Several studies [38, 43, 45, 48, 51, 65, 66] examined idiom impact on code quality and applied them to SE tasks. Ajami et al.[33] empirically demonstrated that disregarding common idioms may increase code complexity. Long et al. [53] noticed that idioms improve code changes in remixes. Da et al.[39] found that high-confidence code recommendations perform better, and recommendation performance declines as source code evolves due to outdated patterns. Haase et al.[47] provided practical guidance on using and choosing idioms, and Nielebock et al. [59] used code patterns for API misuse detection. Finally, Shin et al.[63] introduced a system that mines idioms for code generation by training a neural synthesizer.

9 CONCLUSIONS AND FUTURE WORK

In this paper, we introduce IdioMine, an innovative unsupervised approach for extracting code idioms by three phases, including idiom representation construction, frequent sub-idiom mining, and semantic idiom identification. We assess the effectiveness of IdioMine and the quality of extracted idioms using well-designed experiments and a user study. Experimental results reveal that IdioMine accurately extracts both generic and specific idioms from Java projects and libraries. Specifically, IdioMine outperforms *Haggis* and ChatGPT by 22.8% and 35.5% in Idiom Set Precision (ISP) and by 9.7% and 22.9% in Idiom Coverage (IC) when extracting idioms from Library. IdioMine extracts almost twice the size of idioms than baselines. The user study results validate our extracted idioms outperform baselines in terms of code completeness, semantic clarity, and generality. Furthermore, we thoroughly analyze the primary functionalities of our idioms extracted from each Java project and library. In the future, we plan to expand IdioMine to extract idioms from various programming languages and conduct a thorough analysis of them.

ACKNOWLEDGMENTS

This research/project is supported by the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), National Natural Science Foundation of China (No. 62141222), and the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] 2023. https://en.wikipedia.org/wiki/Programming_idiom.
[2] 2023. https://en.wikipedia.org/wiki/Programming_idiom.
[3] 2023. https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms.
[4] 2023. http://c2.com/ppr/wiki/JavaIdioms/JavaIdioms.html.
[5] 2023. http://shichuan.github.io/javascript-patterns/.
[6] 2023. https://github.com/rwaldron/idiomatic.js/.
[7] 2023. https://github.com/jeffkit/rabbitmq-benchmark/blob/Producer.java#L53.
[8] 2023. http://www.java2s.com/Open-Source/Java_Free_Code/Server/Download_Apposite_Repository_Server_Free_Java_Code.htm.

[9] 2023. https://github.com/structr/structr/blob/0.4.9/structr/structr-core/src/main/java/org/structr/core/entity/RelationClass.java#L330.
[10] 2023. https://structr.com/#start.
[11] 2023. https://en.wikipedia.org/wiki/Command_pattern.
[12] 2023. https://github.com/Yanming-Yang/idioMine.
[13] 2023. https://github.com/c2nes/javalang.
[14] 2023. https://en.wikipedia.org/wiki/Hyperparameter_optimization).
[15] 2023. https://en.wikipedia.org/wiki/Silhouette_(clustering).
[16] 2023. https://openai.com/blog/chatgpt.
[17] 2023. https://openai.com/.
[18] 2023. https://openai.com/blog/introducing-chatgpt-and-whisper-apis.
[19] 2023. https://pytorch.org/.
[20] 2023. https://en.wikipedia.org/wiki/Dispose_pattern.
[21] 2023. https://en.wikipedia.org/wiki/Observer_pattern.
[22] 2023. https://java-design-patterns.com/patterns/producer-consumer/.
[23] 2023. https://en.wikipedia.org/wiki/Builder_pattern.
[24] 2023. https://en.wikipedia.org/wiki/Lazy_initialization.
[25] 2023. https://learn.microsoft.com/en-us/power-query/wait-retry.
[26] 2023. https://en.wikipedia.org/wiki/Double-checked_locking.
[27] 2023. https://en.wikipedia.org/wiki/Factory_method_pattern.
[28] 2023. https://www.javacodegeeks.com/2013/09/android-viewholder-pattern-example.html.
[29] 2023. https://en.wikipedia.org/wiki/Dynamic_dispatch.
[30] 2023. https://en.wikipedia.org/wiki/Random_access.
[31] 2023. Hack is a programming language for the HipHop Virtual Machine, created by Facebook as a dialect of PHP. https://hacklang.org/.
[32] 2023. *Use-define chain.* https://en.wikipedia.org/wiki/Use-define_chain.
[33] Shulamyt Ajami, Yonatan Woodbridge, and Dror G Feitelson. 2019. Syntax, predicates, idioms—what really affects code complexity? *Empirical Software Engineering* 24 (2019), 287–328.
[34] Miltiadis Allamanis, Earl T Barr, Christian Bird, Premkumar Devanbu, Mark Marron, and Charles Sutton. 2018. Mining semantic loop idioms. *IEEE Transactions on Software Engineering* 44, 7 (2018), 651–668.
[35] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *2013 10th working conference on mining software repositories (MSR)*. IEEE, 207–216.
[36] Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*. 472–483.
[37] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).
[38] Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu. 1996. Automatic code generation from design patterns. *IBM systems Journal* 35, 2 (1996), 151–171.
[39] Luiz Laerte Nunes da Silva Junior, Troy Costa Kohwalter, Alexandre Plastino, and Leonardo Gresta Paulino Murta. 2021. Sequential coding patterns: How to use them effectively in code recommendation. *Information and Software Technology* 140 (2021), 106690.
[40] Dario Di Nucci, Hoang-Son Pham, Johan Fabry, Coen De Roover, Kim Mens, Tim Molderez, Siegfried Nijssen, and Vadim Zaytsev. 2019. A Language-Parametric Modular Framework for Mining Idiomatic Code Patterns.. In *SATToSE*.
[41] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey for in-context learning. *arXiv preprint arXiv:2301.00234* (2022).
[42] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, Vol. 96. 226–231.
[43] Joseph Gil and Itay Maman. 2005. Micro patterns in Java code. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 97–116.
[44] Boryana Goncharenko and Vadim Zaytsev. 2016. Language design and implementation for the domain of coding conventions. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. 90–104.
[45] Eduardo Guerra, Menanes Cardoso, Jefferson Silva, and Clovis Fernandes. 2010. Idioms for code annotations in the java language. In *Proceedings of the 8th Latin American Conference on Pattern Languages of Programs*. 1–14.
[46] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).
[47] Arno Haase. 2001. Java Idioms: Code Blocks and Control Flow.. In *EuroPLoP*. 227–250.
[48] Bogumiła Hnatkowska and Anna Jaszczak. 2014. Impact of selected java idioms on source code maintainability–empirical study. In *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30–July 4, 2014, Brunów, Poland*. Springer, 243–254.
[49] Andrew Hunt. 1900. *The pragmatic programmer*. Pearson Education India.
[50] Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. 2019. Learning programmatic idioms for scalable semantic parsing. *arXiv preprint arXiv:1904.09086* (2019).
[51] Christos Kartsaklis, Oscar Hernandez, Chung-Hsing Hsu, Thomas Ilsche, Wayne Joubert, and Richard L Graham. 2012. HERCULES: A pattern driven code transformation system. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 574–583.
[52] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.
[53] Xingyu Long, Peeratham Techapalokul, and Eli Tilevich. 2021. The common coder's scratch programming idioms and their impact on project remixing. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on SPLASH-E*. 1–12.
[54] Robert C Martin. 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.
[55] Steve McConnell. 2004. *Code complete*. Pearson Education.
[56] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.
[57] José J Merchante and Gregorio Robles. 2017. From Python to Pythonic: Searching for Python idioms in GitHub. In *Seminar Series on Advanced Techniques and Tools for Software Evolution (SATToSE)*.
[58] Scott Meyers. 2005. *Effective C++: 55 specific ways to improve your programs and designs*. Pearson Education.
[59] Sebastian Nielebock, Robert Heumüller, Kevin Michael Schott, and Frank Ortmeier. 2021. Guided pattern mining for API misuse detection by change-based code analysis. *Automated Software Engineering* 28, 2 (2021), 15.
[60] Dmitry Orlov. 2020. Finding idioms in source code using subtree counting techniques. In *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part II 9*. Springer, 44–54.
[61] Purit Phan-Udom, Naruedon Wattanakul, Tattiya Sakulniwat, Chaiyong Ragkhitwetsagul, Thanwadee Sunetnanta, Morakot Choetkiertikul, and Raula Gaikovina Kula. 2020. Teddy: automatic recommendation of pythonic idiom usage for pull-based software projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 806–809.
[62] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: synthesizing what I mean: code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering*. 357–367.
[63] Eui Chul Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. 2019. Program synthesis and semantic parsing with learned code idioms. *Advances in Neural Information Processing Systems* 32 (2019).
[64] Aishwarya Sivaraman, Rui Abreu, Andrew Scott, Tobi Akomolede, and Satish Chandra. 2022. Mining idioms in the wild. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 187–196.
[65] Lawrence Snyder. 1982. Recognition and selection of idioms for code optimization. *Acta Informatica* 17, 3 (1982), 327–348.
[66] Miguel Terra-Neves, João Nadkarni, Miguel Ventura, Pedro Resende, Hugo Veiga, and António Alegria. 2021. Duplicated code pattern mining in visual programming languages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1348–1359.
[67] Yilin Yang, Tianxing He, Yang Feng, Shaoying Liu, and Baowen Xu. 2022. Mining Python fix patterns via analyzing fine-grained source code changes. *Empirical Software Engineering* 27, 2 (2022), 48.
[68] Yanming Yang, Xin Xia, David Lo, Tingting Bi, John Grundy, and Xiaohu Yang. 2022. Predictive models in software engineering: Challenges and opportunities. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–72.
[69] Yanming Yang, Xin Xia, David Lo, and John Grundy. 2022. A survey on deep learning for software engineering. *ACM Computing Surveys (CSUR)* 54, 10s (2022), 1–73.