When developing a program to solve a task, developers usually be-
a search problem, where the goal is to explore the set of programs to find the one corresponding to the provided specification. Approaches based on deep learning extract knowledge from large code bases, for instance to generate domain specific programs based on input/output examples [15, 47]. Other approaches rely on neural machine translation to translate functionality descriptions in natural language into executable code in domain specific languages [55], general purpose languages for API usage sequences [13], or more generalized purpose [32, 54]. Automatic program repair is a sub-field of program synthesis defined as “the transformation of an unacceptable behavior of a program into an acceptable one according to a specification” [36]. The goal of automatic program repair is to recommend appropriate patches to modify the part of the program responsible for an identified defect. Some approaches use repair templates to address specific classes of defects [4, 24, 33], while others rely on genetic programming and mutations to find appropriate patches [3, 52]. Code Phage [48] is a system transferring correct code from programs passing a set of test cases to programs in which defects were detected. Program synthesis and program repair approaches rely on specifications that characterize the missing behavior, such as test cases and textual descriptions. Code recommendation is then akin to suggesting code fragments implementing an outlined behavior. In other words, it helps developers determine how to code for a particular scenario. On the contrary, code sophistication aims to suggest missing conditional paths corresponding to unspecified behaviors, relying only on the program under development. Note that some approaches in automatic program repair generate patches without relying on specifications [4, 17, 51]; however, they only target specific classes of defects.

When facing unfamiliar programming tasks, developers often seek code snippet examples; code search aims to suggest such relevant snippets. Most code search approaches are based on information retrieval techniques to find relevant code snippets depending on a query either formulated directly by the developer or inferred from the code [1]. Some approaches focus on recommending specific types of snippets, such as framework usages [21], Java methods [1], exception handling examples [43], auxiliary functionalities [29] or API usages [42], while others are general purpose [22, 23, 25, 35, 45, 49]. Approaches inferring a query from the code can be based on tokens and/or statements similarity [1, 49], structural code features [21, 25, 35, 43], or lexical ones [43]. Code search approaches which are the closest to code sophistication are the ones in which no query is formulated directly and which rely only on the code under development to suggest code fragments. These approaches search for similar fragments and use the provided program as a specification of the intended behavior. In contrast, code sophistication focuses on completing the logic of a program with behaviors that were missed. These behaviors are by definition assumed to be absent from the code. Code search approaches are thus not applicable for sophistication.

3 CODE SOPHISTICATION

In this section, we first lay down and motivate the problem of code sophistication. We then give directions for addressing it.

3.1 Definitions and Motivations

The set of behaviors of a program, sometimes referred as the program’s logic, can be outlined through execution paths, pseudocode, or use cases. Execution paths offer an interesting perspective in our case, because we can think of each path as one behavior, depending on the conditions that delimit in which scenario it is executed [26]. In other words, the choice of an execution path is determined by the program’s inputs and the conditions they satisfy. Those inputs may be explicit (e.g., parameters, attributes) or implicit (e.g., time). Each conditional path thus defines how to behave in a scenario circumscribed by the values of the inputs. Common scenarios correspond to combinations of input values that are typically observed, whereas atypical scenarios correspond to uncommon combinations which happen rarely.

Recommending alternative behaviors to handle omitted scenarios can thus be defined as suggesting missing conditional paths targeting specific combinations of input’s values.

In software testing, missing conditional paths are known as a particular class of defects which do not reside in the written code, but in the absence of a particular fragment [37]. Several studies have analyzed bug fixes of large projects and report that missing paths are a particularly abundant class of defects [7, 11, 31, 41], known to be hard to predict and detect. In The Art of Software Testing [37], Myers et al. report that exhaustive path testing, a common approach to test the logic of a program, might not uncover errors caused by missing paths. Hemmati showed that indeed the largest category of errors that remain undetected by code coverage criteria are “faults of omission” related to missing conditional paths [19]. Chen et al. studied dormant bugs, i.e., bugs introduced in a version of a system and not reported before after the release of the next version [9]. They found that 52% of dormant bugs are related to corner cases and control flow, while only 11% of the non-dormant bugs concern these categories, suggesting they take longer to be exposed. The reasons for this are that scenarios which are not present in the requirements are difficult to detect due to a lack of “local clues about the omission” [8] or because they necessitate particular conditions to be triggered [41]. These suggest that missing conditional paths indeed correspond to omitted scenarios.

To sum up, the literature provides evidence that missing conditional paths corresponding to omitted scenarios are one of the most occurring class of defects found in software projects, and are particularly difficult to detect by both manual and automated software testing approaches. Detecting and patching missing behaviors is thus an important and challenging issue. In addition, we showed in Section 2 that existing code recommendation approaches, while providing valuable avenues to investigate, are not sufficient to recommend missing behaviors in the general case. In what follows, we provide directions for addressing this problem.

3.2 Toward Logic Recommendation

We hypothesize that knowledge about programs’ logic can be derived from available code in large project repositories. More specifically, we aim at learning sophistication patterns from recurring code changes across project histories. We consider that commits adding conditional paths, as illustrated in the Python method in Fig. 1 (a), are good candidates to learn sophistication patterns. Such patterns
should characterize alternative behaviors (i.e., what should be added) and their context (i.e., where and when it should be added).

To achieve generalizability of sophistication patterns across different projects and domains, it is necessary to work with high level characterization of behaviors, independent from implementation concerns. For example, consider two applications, one for the reserving travel tickets, one for banking. The reservation application tests if the date of departure is before the date of return; the banking application tests if the amount of money to be withdrawn exceeds the bank balance. Despite having domain specific implementations, these behaviors are logically similar: they both raise an error if one input value is superior to another. Their characterization must be at a sufficient level of abstraction to reflect this similarity.

Because a program’s logic pertains to its structure, leveraging structural information is crucial to depict the context of alternative behaviors. We are currently investigating the use of structural representations of the code, notably the control flow graph (CFG). A CFG is a graph representing the flow (directed edges) between the statements (nodes) of a program, portraying its different execution paths, as shown in Fig. 1 (b) for the same Python method. The nodes and edges in green represent the path added in the commit. Predicate nodes have two outgoing edges and divide their ingoing paths into two paths which depend on a condition. Adding a new path in a program entails adding a predicate node in its corresponding CFG: thus, we consider that each edge of a CFG is a potential extension point, on which a predicate node can be added to divide the current path and add an alternative path. In Fig 1 (b), we can see that the added predicate node (represented by an hexagon) splits the edge coming from the first node to insert a new path: this edge is an extension point. Also, because the executed behaviors depend on the inputs of the program, we hypothesize that conditional paths may depend on the usage of inputs through the program, and we take these usages into account when describing the context in sophistication patterns.

Based on these assumptions, we outline the problem of recommending missing behaviors as three levels of increasing complexity. (Level 1) Suggest possible extension points in a program. Extension points show where the current control flow should be diverted when a certain condition is met to introduce a new behavior. (Level 2) Given an extension point in the program, suggest a high level description characterizing the potentially missing behavior. Instead of suggesting code fragments representing specific implementation of a behavior, recommended solutions must rather characterize this behavior with a sufficient level of abstraction to be independent of domain specific implementation concerns. (Level 3) Given an extension point and a characterization of the missing behavior, suggest code templates of proposed implementations of the missing behavior.

4 IMPLEMENTATION AND VALIDATION

We propose a preliminary implementation of a code sophistication, based on Graph Convolutional Networks. We present early results for Levels 1 and 2: detecting potential extension points and characterizing missing behaviors. Figure 1 presents an overview of the proposed approach.

4.1 Collecting and Encoding Data

The change history of a software project is an important indicator of the defects which were uncovered and fixed by developers. To gather information about patterns of program logic completion, we focused on commits adding conditional statements, as they can show where a developer added a conditional path that was missing [41]. We analyzed all commits in the history of 250 Github projects written in Python, for a total of 1.2M commits, and extract methods in which an if statement block was added, as illustrated in Fig. 1 (a). To leverage structural information in the extracted methods, we construct their control flow graph (CFG), as shown in Fig.1 (b). In each CFG, we removed the nodes corresponding to the added path (Fig 1 (c)): the pruned CFG thus corresponds to the CFG of the method before the commit. We keep track of the

---

**Figure 1:** Processing a commit’s method to detect extension points (level 1) and characterize missing behaviors (level 2).
edge corresponding to the extension point (represented in orange in Fig 1(c)), as well as the statements present in the path added by the commit (here an Assign and a Call). Finally, we annotate each node of the CFG with the usage of the inputs, as illustrated in Fig 1(d). We considered two types of inputs: the method’s parameters and the attributes used in its body. In our example, the method has two inputs: the parameter point and the attribute self.points. We characterize the usage of inputs by the type of statements in which they appear. In the middle node of Fig. 1(c), self.points is used in Assign and Subscript statements, and point in an Assign statement. The Return statement of the bottom node does not use any input. For each method, we thus have a CFG annotated with the usage of inputs, where one of its edges is identified as an extension point, associated with the statements from the removed path.

4.2 Implementation of Learning Models

For the two experiments we built classification models taking, as input, a CFG annotated with input usages. We used the StellarGraph library [10] to build the graph classification models based on the graph convolutional layers from Kipf and Welling [27]. Because we aim to represent an edge of the graph, we divide the CFG in two parts (before and after the tagged edge, representing the structured inputs’ usage before and after the extension point, as shown in Fig 1(d) and feed the models with these two sub-graphs.

To detect extension points (level 1), we trained a binary classifier to detect if an edge of a given CFG is an extension point. For positive examples, we used the extracted graphs for which we know the edge on which a path was added. For negative examples, we mined methods for which commits added lines that do not correspond to new paths. We created the same annotated CFGs, but with edges which did not correspond to extension points. We evaluate the model with the following metrics: accuracy, precision, recall, F1 measure and area under the curve (AUC). The AUC score measures the probability that our classifier will rank an edge corresponding to an extension point higher than an edge which does not.

To characterize the missing behavior (level 2), we trained a multi-label classifier to identify the types of statements that should be used in the behavior to be added at a given extension point. We selected 8 recurring types of statements as defined in the Python AST module [2] as our classes – Return, Assign, AugAssign, Raise, If, Call, Subscript and BinOp – and labeled each method with the statements appearing in the commit added lines. We compared the results with two state-of-the-art models for code completion in Python, GraphCodeBERT [16] and CodeGPT [34]. Because these two models do not suggest types of statements, we generate for each method the same number of tokens added in the commit. We then extracted the types of statements from the models’ suggestions to obtain a baseline for comparison. We evaluate the three models with the following metrics: AUC, macro and micro F1, and Hamming loss. AUC scores, in the multilabel case, represent an average of the AUC of each class. Macro F1 is the average of F1 measure for each class, while micro F1 is the F1 measure computed on all examples and for all classes. The Hamming loss represents how many times a pair edge-label is misclassified: the lower the loss the better.

### Table 1: Results for the detection of extension points.

| Metrics   | CC(CodeGPT) | CC(GraphCodeBERT) | CS   |
|-----------|-------------|--------------------|------|
| AUC       | .564        | .561               | .750 |
| Macro F1  | .309        | .072               | .358 |
| Micro F1  | .333        | .905               | .397 |
| Hamming loss | .308      | .264               | .237 |

### Table 2: Results for the characterization of behaviors.

4.3 Results

For the two experiments, we performed a 5-fold cross validation: the presented results are the averages for the 5 folds. Table 1 presents the results for the model detecting extension points. A classifier is often considered suitable if its AUC score is above 0.7 [46]. Table 2 compares the results obtained with the two state-of-the-art models for code completion (denoted CC) and our multilabel classifier for code sophistication (denoted CS). We can see that our model provides better results than the two baselines.

In both cases, we have encouraging results suggesting that a) knowledge about programs’ logic can be learned from code repositories and b) structural code information and input’s usage are adequate to identify extension points and characterize missing behaviors, without relying on specifications.

5 CONCLUSION AND FUTURE WORK

We defined and motivate the problem of code sophistication, i.e., completing programs with missing behaviors that were neither specified nor predicted. We discussed how existing code recommendation approaches are not suitable for this problem and proposed an approach to recommending appropriate behaviors that relies on knowledge learned from large code repositories. We presented early results demonstrating we could detect extension points and characterize missing behaviors by learning from the structure and the usage of inputs from commits adding conditional paths.

Our next steps are to study in detail the added lines in the commits of our dataset to provide a characterization of the missing alternative behaviors, with descriptions independent from implementation and domain concerns. This knowledge is crucial to better understand what parts of program logic we should aim to infer for code sophistication. We are currently investigating the approaches used for related tasks, especially code completion and code search, with the goal of adapting and extending the more relevant ones to recommend alternative behaviors and code templates (Level 3). This will help us further determine which code properties are decisive to detect and infer missing behaviors. In the long term, we aim to study how code sophistication could be efficiently carried out by recommendation systems to successfully assist developers into handling omitted scenarios.

REFERENCES

[1] Lei Ai, Zhiqiu Huang, Weiwei Li, Yu Zhou, and Yaoshen Yu. 2019. Sensory: Leveraging code statement sequence information for code snippets recommendation. In 43rd Ann. Computer Software and Applications Conference, Vol. 1. IEEE, 27–36.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. ACM Computing
