Active Inductive Logic Programming for Code Search

Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, Miryung Kim
University of California, Los Angeles
{dcssiva, tianyi.zhang, guyvdb, miryung}@cs.ucla.edu

***Abstract*—Modern search techniques either cannot efficiently incorporate human feedback to refine search results or cannot express structural or semantic properties of desired code. The key insight of our interactive code search technique ALICE is that user feedback can be actively incorporated to allow users to easily express and refine search queries. We design a query language to model the structure and semantics of code as logic facts. Given a code example with user annotations, ALICE automatically extracts a logic query from code features that are tagged as important. Users can refine the search query by labeling one or more examples as desired (positive) or irrelevant (negative). ALICE then infers a new logic query that separates positive examples from negative examples via active inductive logic programming. Our comprehensive simulation experiment shows that ALICE removes a large number of false positives quickly by actively incorporating user feedback. Its search algorithm is also robust to user labeling mistakes. Our choice of leveraging both positive and negative examples and using nested program structure as an inductive bias is effective in refining search queries. Compared with an existing interactive code search technique, ALICE does not require a user to manually construct a search pattern and yet achieves comparable precision and recall with much fewer search iterations. A case study with real developers shows that ALICE is easy to use and helps express complex code patterns.**

***Index Terms*—Code Search, Active Learning, Inductive Logic Programming**

I. INTRODUCTION

Software developers and tools often search for code to perform bug fixes, optimization, refactoring, etc. For example, when fixing a bug in one code location, developers often search for other similar locations to fix the same bug [1]–[3]. Text-based search techniques allow users to express search intent using keywords or regular expressions. However, it is not easy to express program structures or semantic properties using text-based search, thus hindering its capability to accurately locate desired code locations. TXL [4] and Wang et al. [5] provide a domain-specific language (DSL) for describing structural and semantic properties of code. However, learning a new DSL for code search can be cumbersome and time consuming.

Several techniques infer an underlying code search pattern from a user-provided example [2, 6]–[8]. These techniques adopt fixed heuristics to generalize a concrete example to a search pattern, which may not capture various search intent or allow a user to refine the inferred pattern. Lase [9] and Refazer [10] use multiple examples instead of a single example to better infer the search intent of a user. However, this requirement poses a major usability limitation: a user must come up with multiple examples a priori. Critics allows a user to construct an AST-based search pattern from a single example through manual code selection, customization, and parameterization [11]. However, users of Critics report that, the internal representation of a search pattern is not easy to comprehend and that they could benefit from some hints to guide the customization process.

We propose an interactive code search technique, ALICE that infers a search pattern by efficiently incorporating user feedback via active learning. ALICE has three major components: (1) a novel user interface that allows a user to formulate an initial search pattern by tagging important features and iteratively refining it by labeling positive and negative examples, (2) a query language that models structural and semantic properties of a program as logic facts and expresses a search pattern as a logic query, and (3) an active learning algorithm that specializes a search pattern based on the positive and negative examples labeled by the user. In this interface, a user can start by selecting a code block in a method as a seed and by tagging important code features that must be included in the pattern. ALICE then automatically lifts a logic query from the tagged features and matches it against the logic facts extracted from the entire codebase. Our query language models properties including loops, method calls, exception handling, referenced types, containment structures, and sequential ordering in a program. Therefore, our query language can easily express a search pattern such as '*find all code examples that call the* `readNextLine` *method in a loop and handle an exception of type* `FileNotFoundException`,' which cannot be accurately expressed by text-based approaches.

Our active learning algorithm utilizes Inductive Logic Programming (ILP) [12, 13] to refine the search pattern. ILP provides an easy framework to express background knowledge as rules and is well-suited for learning from structured data. ILP does not assume a *flat* data representation and allows to capture structural properties of code not easily captured by other methods such as neural networks [14]. Additionally, ILP has been proven successful in program synthesis [15], specification mining [16, 17], and model checking [18]. Since users can only inspect a handful of search results due to limited time and attention [19], ALICE allows the user to provide *partial feedback* by only labeling a few examples and gradually refine the search pattern in multiple search iterations. ALICE uses an inductive bias

```
1  /**
2   * Return all leading comments of a given node.
3   * @param node
4   * @return an array of Comment or null if there's no
       leading comment
5   */
6  Comment[] getLeadingComments(ASTNode node) {
7    if (this.leadingPtr >= 0) { ①
8      int[] range = null;
9      for (int i=0; range==null &&
          i<=this.leadingPtr; i++) {
10       if (this.leadingNodes[i] == node)
11       range = this.leadingIndexes[i];
12     }
13     if (range != null) { ②
14       int length = range[1]-range[0]+1;
15       Comment[] leadComments = new Comment[length];
16       System.arraycopy(this.comments, range[0],
            leadComments, 0, length);
17       return leadComments;
18     }
19   }
20   return null;
21 }
```

Fig. 1: Bob tags `if` condition `range!=null` and `if` condition `this.leadingPtr>=0` as must-have code elements.

to specialize the previous search pattern to separate the labeled positive and negative examples in each iteration. This feedback loop iterates until the user is satisfied with the returned result.

We evaluate ALICE using two benchmarks from prior work [9, 20]. These benchmarks consist of 20 groups of similar code fragments in large-scale projects such as Eclipse JDT. ALICE learns an intended search pattern and identifies similar locations with 93% precision and 96% recall in three iterations on average, when the user initially tags only two features and labels three examples in each iteration.

A comprehensive simulation experiment shows that labeling both positive and negative examples are necessary to obtain the best search results, compared to labeling only positives or only negatives. It is because negative examples quickly reduce the search space, while positive examples are used to capture the search intent. We vary the number of labeled examples and find that labeling more examples in each iteration does not necessarily improve the final search accuracy. This indicates that a user can label only a few in each iteration to reach good accuracy eventually. ALICE thus alleviates the burden of labeling many examples at once, as its active learning can leverage partial feedback effectively. The comparison with an existing technique Critics [11] shows that ALICE achieves the same or better accuracy with fewer iterations.

This human-in-the-loop approach in ALICE opens a door for incorporating user feedback in a variety of software engineering techniques, e.g., bug detection, code optimization, refactoring, etc. For example, a code optimization framework, Casper, searches for Java programs with loops that sequentially iterate over data and translates the loops to semantically equivalent `map-reduce` programs [21]. However, the search patterns in Casper are hardcoded and cannot be easily expressed by a user. ALICE can aid in such scenarios by allowing a user to interactively construct a search pattern. Our paper makes the following contributions.

```
1  public int getExtendedStartPosition(ASTNode node) {
2    if (this.leadingPtr >= 0) {
3      int[] range = null;
4      for (int i=0; i<=this.leadingPtr; i++) {
5        if (this.leadingNodes[i] == node)
6        range = this.leadingIndexes[i];
7      }
8      if (range != null) {
9        return
10       this.comments[range[0]].getStartPosition() ;
11     }
12   }
13   return node.getStartPosition();
14 }
```

Fig. 2: Bob labels this example as positive. This example is syntactically similar to Figure 1, but has a different `loop` condition `i<=this.leadingPtr`.

- We present a novel approach called ALICE that integrates active learning and inductive logic programming to incorporate partial user feedback and refine code search patterns. ALICE is instantiated as an Eclipse plug-in and the tool is available online.[1]
- We conduct a comprehensive simulation experiment that investigates the effectiveness of ALICE using different inductive biases, different numbers of labeled examples, and different numbers of annotated code features. An additional robustness experiment shows that our search algorithm is resilient to labeling mistakes by flagging contradictory examples labeled by a user.
- We conduct a case study with real developers, demonstrating that participants can easily interact with ALICE to search desired code by simply inspecting and labeling code examples. On average, each participant labels two examples in each search iteration and spends about 35 seconds on each example.

II. MOTIVATING EXAMPLE

This section describes the code search process using ALICE with a real-world example drawn from Eclipse Java Development Toolkit (JDT). Eclipse JDT provides basic tools and library APIs to implement and extend Eclipse plug-ins. As an Eclipse JDT developer, Bob wants to update the `getLeadingComments` method (Figure 1) to return an empty `Comment` array when there are no leading comments, instead of returning `null`, which may cause a `NullPointerException` in a caller method. Before modifying the code, Bob wants to check other similar locations.

Bob could use a text-based search tool to find other code fragments similar to the `getLeadingComments` method. For example, Bob could use `Comment` and `arrayCopy` as keywords to find code locations that reference the `Comment` type and call the `arrayCopy` method. Such text-based techniques are prone to return a large number of search results, many of which merely contain the same keywords but have significantly different code structures or irrelevant functionality. In the JDT codebase, searching with the

[1] Our tool and dataset are available at https://github.com/AishwaryaSivaraman/ALICE-ILP-for-Code-Search

```
1  public void checkComment() {
2    ...
3    if (lastComment >= 0) {
4      this.modifiersSourceStart =
           this.scanner.commentStarts[0];
5
6      while (lastComment >= 0 &&
           this.scanner.commentStops[lastComment] < 0)
           lastComment--;
7      if (lastComment >= 0 && this.javadocParser !=
           null) {
8        if (this.javadocParser.checkDeprecation(
9            this.scanner.commentStarts[lastComment],
10           this.scanner.commentStops[lastComment] - 1)) {
11         checkAndSetModifiers(AccDeprecated);
12       }
13       this.javadoc = this.javadocParser.docComment;
14     }
15   }
16 }
```

Fig. 3: Bob labels this example as negative. Though it has a similar `if` check `lastComment>=0`, its program structure is significantly different from Figure 1.

`arrayCopy` keyword returns 1854 method locations. It is prohibitively time-consuming to inspect all of them. Prior work has shown that developers only inspect a handful of search results and return to their own code due to limited time and attention [19, 22, 23].

To narrow down the search results, it would be useful to describe the structural properties of `getLeadingComments`. For example, a code snippet must contain an `if` condition that checks for `>=0` and contain a loop inside the `if` block. Instead of requiring a user to prescribe such a structural pattern manually, ALICE enables the user to tag important code features and label positive vs. negative examples instead.

Iteration 1. Select a Seed Example and Annotate Important Features. Bob first selects a code block of interest in the `getLeadingComments` method (lines 7-19 in Figure 1). He *annotates* an `if` condition, `range!=null` (② in Figure 1), as a code element that must be included in the search pattern, since he posits that similar locations would have a `null` check on the range object. Such must-have code elements are colored in green. ALICE then automatically constructs a search query based on the selected code and annotations. Since other developers may rename variables in different contexts, ALICE generalizes the variable name `range` in the tagged `if` condition to match a `null` check on any variable. Among all 12633 Java methods in the Eclipse JDT codebase, ALICE returns 1605 methods with a `null` check, which are too many to examine. Bob now has two choices. He can annotate more features or label a few examples to reduce the search results. Bob tags another `if` condition, `this.leadingPtr>=0`, as another must-have element (① in Figure 1). The field name `leadingPtr` is also generalized to match any variable name. ALICE refines the previous query and returns ten methods that both perform a `null` check and contain an `if` condition checking, if a field is no less than 0. The query is shown below and its syntax is detailed in Section III-D.

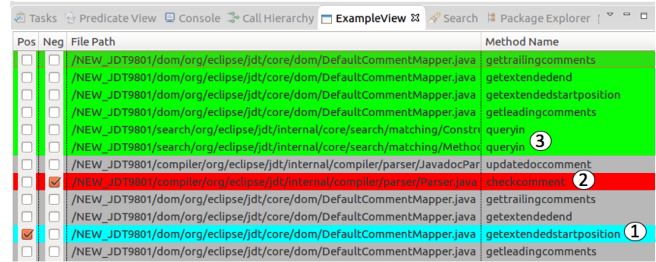

Fig. 4: Bob labels some examples as desired and some as irrelevant. Green ③ indicates newly returned examples, Red ② indicates a previously marked negative examples, and Cyan ① indicates a previously marked positive example.

```
query(X)  :-  methoddec(X),
  contains(X,IF0), iflike(IF0,"this.*>=0"),
  contains(X,IF2), iflike(IF2,".*!=null").
```

Iterations 2 to N. Active Learning by Labeling Positive and Negative Examples. After inspecting two examples returned from the previous iteration, Bob labels one example (Figure 2) as positive and another one (Figure 3) as negative. Bob labels Figure 3 as irrelevant, since this example has similar `if` conditions (line 3), but does not compute the range of a code comment or returns a range. By incorporating this feedback, ALICE learns a new query that includes a unique structural characteristic—*the null check is contained by the if check for an index field greater or equal to 0*—that appears in both the initially selected example and the positive example but not in the negative example, as shown below. As a result, ALICE narrows down to six examples (③ in Figure 4). A user can further refine the search query and continue to iterate by labeling examples. As ALICE uses a top-down search algorithm to *specialize* its search results in each iteration, the resulting set is always a subset of the previous results.

```
query(X)  :-  methoddec(X),
  contains(X,IF0), iflike(IF0,"this.*>=0"),
  contains(IF0,IF2), iflike(IF2,".*!=null").
```

III. APPROACH

A. Machine Learning Approach

Traditional code search techniques retrieve code examples by matching user-specified patterns. In contrast, ALICE's central thesis is that such patterns need not be fully specified by the user, and can instead be induced from specific user interactions. This is fundamentally a machine learning approach to code search: the intended pattern is a hypothesis that is learned from data. To achieve this goal, ALICE integrates several learning paradigms.

First, it requires labels that categorize code examples as either matching the intended hypothesis or not. Classical supervised learning would require the user to provide prohibitively many labels. ALICE, therefore, employs *active learning* [24], where the learner has the ability to present some unlabeled examples to the user during learning and ask to provide a label. In a well-designed active learning

setup, the user only needs to provide a minimal number of labels for the learner to reliably find the correct hypothesis.

Second, most machine learning approaches represent data as feature vectors, which cannot easily express the structure of source code. *Inductive logic programming* (ILP) [12, 25] is a form of relational learning that supports structured data encoded as logical facts. In particular, ALICE uses the logical predicates listed in the first column of Table I to represent each code example in a factbase. The next section describes this process in detail.

Given that the data is now in a suitable form, ILP aims to learn a hypothesis by constructing a logical (Prolog [26]) query that returns exactly those code IDs that the user is searching for. Queries in ALICE are represented as definite clauses. For example, in the queries shown in Section II, we are looking for values of the logical variable `X` for which the body of the query is true. The body is true if there exists a value for the remaining logical variables (uppercase), such that each atom in the body is also found in the factbase. The process of learning a query is centered around two operations: *generalization* and *specialization*. Generalization changes the query to return more results, e.g., removing atoms from the body, or replacing constants with variables. Specialization is the reverse operation, yielding a query with fewer results. An ILP learner uses these operations to search the space of all hypotheses for the right one.

Finally, even with active learning, ALICE operates in a regime where little labeled data is available. We know from learning theory that this is only possible when the learner has a strong *inductive bias*. That is, the learner already incorporates a lot of knowledge about the learning task, before even seeing the data. We address this in three ways. First, ILP affords the specification of declarative background knowledge, which helps to create a more meaningful inductive bias as it applies to code search. Table II shows ALICE's background knowledge, which enables ALICE to use additional predicates to construct richer queries. Second, we allow the user to annotate important code features in order to obtain a stronger inductive bias. Third, ALICE adopts a specialization procedure that is specifically designed to capture different program structures to strengthen the inductive bias. We implement our own realtime ILP system based on the high-performance YAP Prolog engine [27].

B. Logic Fact Extraction

ALICE creates a factbase from a given code repository using the predicates described in Table I. It parses program files to Abstract Syntax Trees (ASTs) and traverses the ASTs to extract logic facts from each method. Predicates `if` and `loop` capture the control flow within a method; `methodcall` represents the relationship between a method and its caller; `exception` captures the type of handled exceptions; `type` captures the type of defined and used variables; `contains` describes the relationship between each AST node and its parent recursively; `before` captures the sequential ordering of AST nodes. Specifically, before(id_1,id_2) is true, when node id_1 comes before node id_2 in preorder traversal while excluding any direct or transitive containment relationship. For Figure 1, `loop(loop₁, "range==null && i<=this.leadingPtr")` comes before `if(if₂, "range!=null")` creating `before(loop₁, if₂)`. Ground facts are then loaded into a YAP Prolog engine [27] for querying.

TABLE I: Predicates in Logical Representation

Fact Predicates	Rule Predicates
if(ID, COND)	iflike(ID, REGEX)
loop(ID, COND)	looplike(ID, REGEX)
parent(ID, ID)	contains(ID, ID)
next(ID, ID)	before(ID, ID)
methodcall(ID, CALL)	
type(ID, NAME)	
exception(ID, NAME)	
methoddec(ID)	

TABLE II: Background Knowledge

Prolog Rules		
iflike(ID, REGEX)	:-	if(ID, COND), regex_match(COND, REGEX).
looplike(ID, REGEX)	:-	loop(ID, COND), regex_match(COND, REGEX).
contains(ID_1, ID_2)	:-	parent(ID_1, ID_2).
contains(ID_1, ID_3)	:-	parent(ID_1, ID_2), contains(ID_2, ID_3).
before(ID_1, ID_2)	:-	next(ID_1, ID_2).
before(ID_1, ID_3)	:-	next(ID_1, ID_2), before(ID_2, ID_3).

C. Generalization with Feature Annotation (Iteration 1)

When a user selects a code block of interest, ALICE generates a specific query which is a conjunction of atoms constructed using the predicates in Table I. Each atom in the query is grounded with constants for its location `ID` and value representing the AST node content. ALICE replaces all `ID` constants in the query with logical variables to generate an initial candidate hypothesis h_0. For example, one ground atom in the query is of the form `if(if,"range!=null")` and its variablized atom is of the form `if(IF,"range!=null")`. To find more examples, ALICE generalizes the hypothesis by dropping atoms in h_0 other than the user annotated ones, producing h_1.

Regex Conversion. ALICE further abstracts variable names in h_1 to identify other locations that are similar but have different variable names. ALICE converts predicates `if` and `loop` to `iflike` and `looplike` respectively. As defined in Table II, `iflike` and `looplike` are background rules that match a regular expression (`REGEX`) with ground conditions (`COND`) in the factbase. For instance, each variable name in `loop(ID,"range==null && i<= this.leadingPtr")` is converted from a string constant to a Kleene closure expression, generating `looplike(ID,".*==null && .*<=this.*")`. The output of this phase is a generalized query h_2 and a set of code examples that satisfy this query. This example set is displayed in the *Example View* in Figure 4.

D. Specialization via Active Learning (Iterations 2 to N)

In each subsequent search iteration, given the set of positive examples (P) and the set of negative examples (N) labeled by a user, the previous hypothesis h_{i-1}, which is a conjunction of atoms, is *specialized* to h_i by adding another atom to exclude all negatives while maximally covering positives. The specialization function is defined below.

$$\texttt{Specialize}(h_{i-1}, P, N) = \underset{h_i}{\operatorname{argmax}} \sum_{p \in P} [p \models h_i]$$
$$\text{such that } h_i \models h_{i-1} \text{ and } \forall n \in N, n \not\models h_i$$

Suppose all positive examples call `foo` and all negative examples call `bar` instead of `foo`. We add a new atom `calls(m,"foo")` to specialize h_{i-1}, which distinguishes the positives from the negatives. As a design choice, our active ILP algorithm is *consistent* (i.e., not covering any negative example) but is not *complete* (i.e., maximally satisfying positive examples). Our learning algorithm is monotonic in that it keeps adding a new conjunctive atom in each search iteration. This specialization procedure always returns a subset of the previous search results obtained by h_{i-1}. This feedback loop continues to the n-th iteration until the user is satisfied with the search results.

Given the large number of candidate atoms, *inductive bias* is required to guide the specialization process of picking a discriminatory atom. ALICE implements three inductive biases, which are described below. The effectiveness of each bias is empirically evaluated in Section IV.

- **Feature Vector.** This bias considers each code block to have a *flat* structure. The feature vector bias does not consider the nested structure or sequential code order. It specializes by adding a random atom that reflects the existence of loops, if checks, method calls, types, or exceptions in the code block. It is used as the baseline bias in the evaluation since it does not utilize any structural information such as containment and ordering.
- **Nested Structure.** This bias utilizes the containment structure of the seed code example to add atoms. In addition to adding an atom corresponding to the AST node, the bias adds a `contains` predicate to connect the newly added atom to one that already appears in the query. Consider an AST with root node A, whose children are B and C; B has children D and E, and C has child F. Suppose that h includes an atom referring to B. Then based on the containment relationships, ALICE selects one of D or E to specialize the query for the next iteration, not F. If there are no available children, or if this query fails to separate positives from negatives, it falls back to the parent of B or its further ancestors to construct the next query. We choose this nested structure bias as default since it empirically achieves the best performance (detailed in Section IV-A).
- **Sequential Code Order.** This bias uses sequential ordering of code in the seed example to determine which atom to add next. Consider an example AST with root node A and children B and C; C itself has children D and E. Atoms `before(`B`,`D`)`, `before(`B`,`C`)`, and `before(`B`,`E`)` are generated according to the rules in Table II. Given a query that contains atoms referring to B, ALICE now chooses one of C, D, or E, to connect to B using the `before` predicate, and adds this node to the query. If there are no available atoms to add, or if this query fails to separate positives from negatives, it falls back to the original feature vector bias.

An alternative approach is to carry out logical generalization where we generate a query by generalizing positive examples and taking the conjunction with the negation of negative example generalization. As a design choice, we do not allow negations in our query for realtime performance, since supporting negations would significantly increase the search space and execution time.

IV. Simulation Experiment

We systematically evaluate the accuracy and effectiveness of ALICE by assessing different search strategies and by simulating various user behaviors.

Dataset. We use two complementary sets of similar code fragments as the ground truth for evaluation, as shown in Table III. The first dataset is drawn from the evaluation data set of LASE [9]. This dataset consists of groups of syntactically similar code locations in Eclipse JDT and SWT, where developers need to apply similar bug fixes. We select groups with more than two similar locations, resulting in 14 groups in total. Each group contains an average of five similar code fragments and each fragment contains a median of 24 lines of code, ranging from 3 to 648. ALICE extracts an average of 670K logic facts from each repository. The second data set is from the evaluation dataset of Casper [20], an automated code optimization technique. This dataset consists of groups of similar code fragments that follow the same data access patterns (e.g., a sequential loop over lists) and can be systematically optimized by Casper. By evaluating ALICE with both datasets, we demonstrate that ALICE is capable of accurately and effectively searching code in two different usage scenarios—bug fixing and code optimization. Because the second data set includes only the relevant files not the entire codebase, we cannot reliably assess the rate of false positives. Hence we exclude this second dataset when assessing the impact of individual biases, annotations, and labeling effort in Sections IV-A, IV-B, IV-C, and IV-D.

Experiment environment. All experiments are conducted on a single machine with an Intel Core i7-7500U CPU (2.7GHz, 2 cores/4 threads, x64 4.13.0-31-generic), 16GB RAM, and Ubuntu 16.04 LTS. We use YAP Prolog (version 6.3.3), a high-performance Prolog engine [27] to evaluate search queries.

We write a simulation script to randomly select a code fragment in each group as the seed example. In the first iteration, the script randomly tags k important features that represent control structures, method calls, and types in the seed example. In each subsequent iteration, it simulates the user behavior by randomly labeling n examples returned in the previous iteration. If a code example appears in the

TABLE III: Simulation Experiment Dataset and Results Summary

Groups								Repo Revision	#Methods	Factbase Size
ID	LOC	Snippets	Precision	Recall	F1	#Iterations	Query Length			
1	14	4	1.0	0.75	0.86	3	12	JDT 9801	12633	541360
2	44	9	1.0	0.91	0.95	3	11	JDT 10610	13165	570049
3	648	5	1.0	0.92	0.96	2	11			
4	8	6	1.0	1.0	1.0	4	18	SWT 16739	35863	918818
5	47	3	1.0	1.0	1.0	4	13			
6	11	7	1.0	1.0	1.0	2	12			
7	19	3	1.0	1.0	1.0	3	11	SWT 3213515	20899	663980
8	3	6	1.0	1.0	1.0	5	18			
9	18	5	1.0	1.0	1.0	3	11			
10	28	10	1.0	1.0	1.0	4	13			
11	112	4	1.0	1.0	1.0	3	17			
12	30	3	1.0	1.0	1.0	5	11			
13	34	3	1.0	1.0	1.0	3	15			
14	18	7	1.0	1.0	1.0	3	12			
Average	74	5	1.0	0.97	0.98	3.4	13		20640	673551
15	10	11	1.0	1.0	1.0	1	5	Arith	31	316
16	16	6	1.0	1.0	1.0	1	5	Bigλ	15	498
17	9	6	1.0	0.66	0.8	1	3	Phoenix	30	783
18	37	2	1.0	1.0	1.0	2	5			
19	6	3	0.3	1.0	0.46	1	3	Stats	60	1343
20	9	2	0.2	1.0	0.33	1	3			
Average	15	5	0.75	0.94	0.76	1.1	4		34	735
Total Average	56	5	0.93	0.96	0.92	2.7	11		10337	337143

ground truth, it is labeled as positive. Otherwise, it is labeled as negative. The script terminates if no more examples can be labeled in a new search iteration. To mitigate the impact of random choices, we repeat the simulation ten times and report the average numbers for each group.

Result summary. Table III summarizes the precision, recall, and F1 score of ALICE in the final search iteration. When setting k and n to two and three respectively, ALICE empirically achieves the best result, which we detail in Sections IV-C and IV-D. On average, ALICE successfully identifies similar code locations with 93% precision and 96% recall in 2.7 search iterations. ALICE achieves 100% precision and 97% recall in the first dataset, while it achieves 75% precision and 94% recall in the second dataset. The reason is that the second dataset contains code fragments that loop over a double array with no write to output operations, which is a semantic constraint imposed by Casper [20] for loop optimization. However, ALICE does not learn predicates that differentiate read and write operations on an array and therefore returns a large number of code fragments that write double arrays in a loop, which leads to low precision.

In the first search iteration, 138 methods are returned by ALICE on average (median: 23 and maximum: 2352). This set is large, motivating our approach to leverage partial feedback (i.e., only three labeled examples at a time) as opposed to labeling all returned results at once. ALICE effectively incorporates partial feedback to reduce the number of returned examples by 93% in the second iteration. Since ALICE adopts a top-down search process and always starts with a general query, the initial search result tends to include all code locations in the ground truth. Therefore, the first search iteration often starts with 100% recall and relatively low precision. The precision then improves gradually, as the query is specialized to eliminate false positives in the search results. In Table III, column query length represents the number of atoms in the final query. On average the query length inferred by ALICE contains 11 atoms.

A. *Impact of Inductive Bias*

During query specialization, an inductive bias is used to effectively navigate the space of all hypotheses. We evaluate the effectiveness of each of the three implemented inductive biases discussed in Section III-D: (1) feature vector, (2) nested structure, and (3) sequential order.

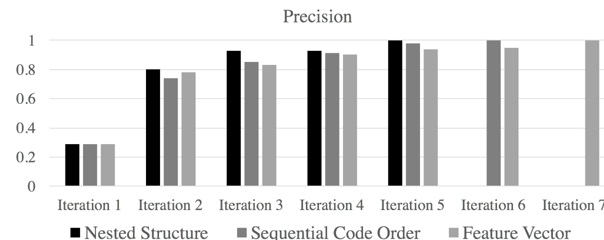

Fig. 5: Precision of ALICE using different inductive biases

Figures 5 and 6 show the effectiveness of each bias in terms of precision and recall, averaged across ten runs. Overall, the nested structure bias converges fast by taking fewer iterations to reach the highest F1 score. The sequential order bias performs better than the feature vector bias, converging in six iterations as opposed to seven. Although

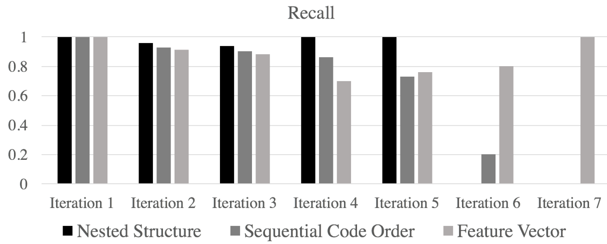

Fig. 6: Recall of ALICE using different inductive biases

both the sequential order bias and the nested structure bias perform well in terms of precision, encountering the same sequential order between statements or methods is not common in the data set. Therefore, the sequential order bias has a lower recall. The nested structure bias adds atoms based on containment relationships and hence it filters out false positives early and converges faster.

```
1 checkWidget ();
2 if (!parent.checkData (this, true)) error
      (SWT.ERROR_WIDGET_DISPOSED);
3 return font != null ? font : parent.getFont ();
```

Fig. 7: A code snippet from Group 8 in Table III

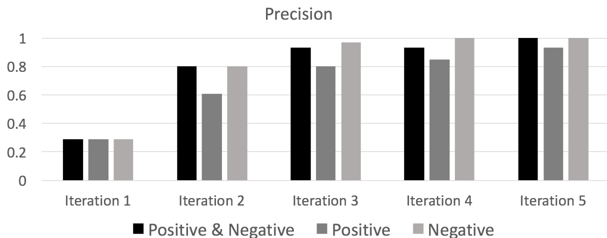

Fig. 8: Precision with different types of labeled examples

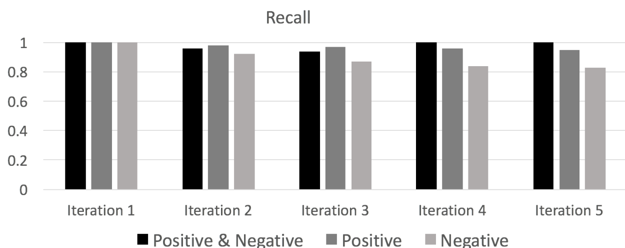

Fig. 9: Recall with different types of labeled examples

Consider Figure 7, taken from group 8 in Table III. The first iteration returns 90 examples when annotating `checkData` and `error`. The feature vector bias takes six iterations; the sequential order bias takes five; and the nested structure bias takes four. This is because the feature vector bias learns a query that describes calls to `checkData`, `checkWidget`, and `error` regardless of their structure and order. In contrast, the sequential bias finds code that calls `checkWidget` before `error` and `getFont`. The nested structure bias finds code that calls `checkData` within an `if` statement, which respects the structure of the selected block. We find that a *flat* representation of code as a feature vector is not powerful enough to effectively capture the search intent in a few iterations. We need to incorporate the rich structural information already available in the code to achieve desired performance.

TABLE IV: Varying # of tagged features in the first iteration

	1 Feature	2 Features	3 Features	4 Features
Precision	0.16	0.47	0.68	0.80
Recall	0.91	0.86	0.80	0.78

B. Labeling Positives and Negatives

To quantify the effect of using different types of labeled code examples, we assess ALICE in three different conditions during query specialization—(1) incorporating both positive and negative examples, (2) considering negative examples only, and (3) considering positive examples only.

Figures 8 and 9 compare the precision and recall in different conditions. When considering negative examples only, ALICE achieves a high precision while converging in five iterations, whereas considering positives only takes more iterations to converge with lower precision. This is because ALICE often starts with a general query in the first search iteration, resulting in a large number of search results that contain many false positives. Hence, it is essential to label several negatives to eliminate those false positives quickly. However, if we do not label any positives, we are likely to remove some true positives as well. This is reflected in the recall, where giving negative examples only has the least recall. Therefore, an optimal choice would be to use both positive and negative examples while iteratively specializing the query, which justifies the design of ALICE.

C. Varying the Number of Annotated Features

Table IV summarizes the average precision and recall right after feature annotation, when varying the number of annotations from one to five. These features are randomly selected from the control structures, method calls and types in the seed example, and the results are averaged over ten runs. The result suggests that annotating more code elements in the seed example can more precisely express the search intent and thus increase the precision. However, the recall is negatively impacted by increasing the number of annotations. When the simulation experiment chooses more features, it is likely to choose features that are not shared across expected code fragments in the ground truth. Thus, the initial search query becomes too specific and therefore misses some expected code fragments. Let us consider the two similar but not identical code examples in Section II (Figures 1 and 2). If a user tags `range==null&&i<=this.leadingPtr` (line 9 in Figure 1) as important, the recall decreases since the expected code example in Figure 2 does not contain this feature. Though a real user can annotate any number of code elements, our experiment shows that tagging two code elements leads to the optimal precision and recall.

TABLE V: Varying # of labeled examples in each iteration

	2 Labels	3 Labels	4 Labels	5 Labels
Precision	1.0	1.0	1.0	1.0
Recall	1.0	0.88	0.81	0.75
# Iterations	7	6	5	5
# Total Labels	14	18	20	25

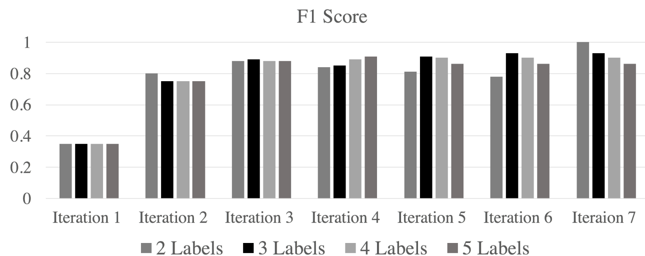

Fig. 10: F1 score for different # labels set at each iteration

D. Varying the Number of Labeled Examples

The type and number of examples that a user labels in each iteration is an important aspect of active-learning based tool design. To quantify the benefit of iterative feedback over simultaneous labeling, we vary the number of labeled examples n from two to five. Setting n to four or five converges the fastest, with five iterations. When n is three, ALICE takes six iterations to converge. When n is two, it takes seven. Table V summarizes the average precision and recall for different number of labels in the final iteration. Overall, increasing the number of labeled examples maintains precision but does not necessarily increase recall. This is due to overfitting of labeled examples. From Table V and Figure 10, $n = 3$ gives a good trade-off between F1 score and the required amount of user feedback, and is more robust than $n = 2$. Figure 10 shows how user feedback helps, and why it is better to spread labeling effort over multiple iterations rather than providing all labeled examples at once.

ALICE takes an average of 20 seconds to parse a Java repository of about 20K methods and extract logic facts. This fact extraction is done only once in the beginning. ALICE takes 6.8 seconds on average (median: 5 seconds) to infer a query and match the learned query to the entire codebase of about 670K facts. This realtime performance is appreciated by study participants, described in Section V.

V. A Case Study with Users

To investigate how a user perceives the paradigm of annotating features and labeling examples during code search, we undertake a case study with three participants. The goal of this study is to investigate how ALICE may fit or augment current code search practices, rather than assessing how much benefit ALICE provides in comparison to alternative code search approaches. Therefore, we use a case study research method [28] to understand the in-depth context of how a user may interact with ALICE to search desired code. In this study, a user tagged features, examined search results, and labeled them as positive and negative examples. All participants were graduate students with multiple years of Java development experience. Participants reported that they use `grep` or a regular expression search on a regular basis or have experience of using search features in IDEs. In terms of the study setup, we gave a tutorial about how to use ALICE. This included reading a written tutorial followed by a step-by-step demonstration of tool features. The participants were given a seed example of `getSelectionText` from Eclipse SWT, which is part of Group 5 in Table III. Since the participants were not familiar with the codebase, we provided two other similar snippets as a reference. Each session took about 30 minutes to 1 hour. Participants were encouraged to follow a think-aloud protocol to verbalize their thought process.

Do you think marking features in a snippet helps search?

> *"I think marking features helps greatly when trying to find similar code snippets—as a human, my intuition is to identify key sections of code that I expect to be similar and ignore other sections that I expect to differ."*
> *"It is obvious that when we are searching for a piece of code and we are expecting to have certain features in it, so marking features in a code snippet helps."*

Participants can easily identify important features in the seed example. Two of three participants annotated three features and one started with one feature. Participants found ALICE useful for refactoring and reorganizing *fragmented yet similar* code snippets. They said that ALICE would be more useful if integrated into code sharing websites such as GitHub and BitBucket by enabling advanced code search in large-scale repositories, because novice programmers would often prefer to look up functions with similar implementations for optimization opportunities.

How do you think that labeling positive and negative examples fit in the search process?

> *"As a developer, I will mentally do a quick evaluation of search results to determine if something is definitely negative (i.e., I can skip it) or potentially positive (i.e., it warrants more investigation). It's normally not an explicit process, but it makes sense to explicitly request feedback when interacting with tools."*
> *"I really liked the iterative refinement of examples and the almost-realtime performance of the tool. One aspect that may be improved is, assigning a score or priority to the examples (based on how many features are satisfied) so that the user can prioritize which examples to inspect."*

How do you think that an interactive and iterative approach such as ALICE compares to other search techniques that you used in the past?

> *"ALICE seems useful for finding similar code snippets that span code blocks. It provides a functionality separate from other search techniques I've used such as IDE-based functionalities (e.g., call declaration/ method invocation locations) and standard `grep` which is limited to single line searches. I think ALICE is generally less applicable, but also significantly more powerful. It also helps promote better understanding—for example, IDEs can show where

TABLE VI: Time taken to label examples in Iteration 1

	#Examples	#Positives	#Negatives	Time Taken(s)
User 1	8	1	1	20
User 2	437	0	2	55
User 3	8	1	0	25

a method is called from, but ALICE can easily show the context in which that method invocation appears.”

Participants said ALICE was more powerful than standard search tools like `grep` and built-in search features in IDEs.

What do you like or not like about Alice? Overall, participants like the interactive feature, as it allows for refinement and builds on developer understanding. Some participants find the color scheme in ALICE confusing due to the conflict with existing syntax highlighting in Eclipse.

We observe that participants were able to easily recognize the important features in the code example and tag them in the first search iteration. Though participants had little experience with the codebase, they could still distinguish positive and negative examples without much effort. Table VI summarizes the number of examples returned in the first search iteration and the time taken for each user to refine the search. In particular, a user took an average of 35 seconds to inspect each example and categorize it as positive or negative. This indicates that the tool does not require much effort for a user to inspect and label examples.

VI. COMPARISON

We compare ALICE with an existing interactive code search technique called Critics. We choose Critics as a comparison baseline, since Critics also performs interactive search query refinement [11]. ALICE differs from Critics in two ways. First, in Critics, a user has to manually select code blocks such as an if-statement or a while loop and parameterizes contents, for example, by replacing `Foo f=new Foo()` to `$T $v=new $T()`. Such interaction is time-consuming. To reduce the user burden, ALICE infers a syntactic pattern such as “`an if-statement with a condition .*!=null inside a while loop`” from positive and negative methods. Second, Critics identifies similar code via tree matching, while ALICE abstracts source code to logic facts and identifies similar code via ILP.

We run ALICE on the public data set obtained from Critics's website.[2] Table VII summarizes the results of Critics vs. ALICE. In six out of seven cases, ALICE achieves the same or better precision and recall with fewer iterations to converge, compared to Critics. In ID 4, ALICE has low precision because the expected code contains a `switch` statement, which is currently not extracted by ALICE as a logic fact. Extending current logic predicates to support more syntactic constructs remain as future work.

VII. DISCUSSION

Noisy Oracle. The simulation in Section IV assumes that a user makes no mistakes when labeling examples. However, it is possible that a real user may label a positive example as negative, or even provide an inconsistent set of labels. We investigate how resilient ALICE is to labeling mistakes and how quickly ALICE can inform the user of such inconsistencies. We mutate our automated oracle with an error rate of 10%, 20%, and 40%. Each of the 14 groups from the first dataset is run five times (70 trials) with different annotations, labels, and errors. In many cases, ALICE reports inconsistencies in user labeling and provides immediate feedback on the infeasibility of specializing queries (33% to 60%). When ALICE does not find any inconsistencies, ALICE behaves robustly with respect to errors, eventually reaching 100% precision. Table VIII summarizes the results.

[2]https://github.com/tianyi-zhang/Critics

TABLE VII: Comparison Against Critics

	Alice			Critics		
Critics ID	Precision	Recall	Iterations	Precision	Recall	Iterations
1	1.0	1.0	2	1.0	1.0	4
2	1.0	1.0	2	1.0	0.9	6
3	1.0	1.0	1	1.0	0.88	6
4	0.0	1.0	1	1.0	1.0	0
5	1.0	1.0	3	1.0	1.0	7
6	1.0	1.0	3	1.0	1.0	4
7	1.0	1.0	3	1.0	0.33	3
Average	0.86	1.0	2.1	1.0	0.87	4.3

TABLE VIII: Sensitivity of ALICE to labeling errors.

	Error Rates		
	10%	20%	40%
Precision	1.0	1.0	1.0
Recall	0.95	0.90	0.93
% Inconsistent Cases	33%	60%	54%

Threats to Validity. Regarding internal validity, the effectiveness of different inductive biases may depend on the extent and nature of code cloning in a codebase. For example, when there are many code clones with similar nested code structures (`while` and `if` statements), the nested structure may perform better than other inductive biases. The current simulation experiment is run on ALICE by choosing one seed example from each group, by randomly selecting annotations from the selected seed, and by labeling a randomly chosen subset of returned results. To mitigate the impact of random selection, we repeat ten runs and report the average numbers. In terms of external validity, we assume that any user could easily annotate features and label examples. However, it is likely that a novice programmer might find it hard to identify important features. To mitigate this threat to validity, as future work, we will investigate the impact of different expertise levels.

Limitations and Future Work. Currently, we generate facts based on structural and intra-procedural control flow properties. Other types of analysis such as dataflow analysis or aliasing analysis could be used in identifying similar snippets. In addition, the query language itself can be extended to make it easier to capture the properties of desired code. For example, by introducing negations in the query language,

a user can specify atoms that should not be included. There could be specializations that strictly require negations. However, in our experiments, empirically, we are always able to find a pattern without negations. As mentioned in Section III-D, our learning process is monotonic and to learn a different query, a user may need to start over. To overcome this, we may need backtracking and investigate new search algorithms that generalize and specialize the query in a different way.

VIII. Related Work

Code Search and Clone Detection. Different code search techniques and tools have been proposed to retrieve code examples from a large corpus of data [29]–[34]. The most popular search approaches are based on text, regular expressions, constraints [35], and natural language [31, 36]–[38]. Exemplar [39] takes a natural language query as input and uses information retrieval and program analysis techniques to identify relevant code. Wang et al. propose a dependence-based code search technique that matches a given pattern against system dependence graphs [5]. XSnippet [30] allows a user to search based on object instantiation using type hierarchy information from a given example. ALICE differs from these search techniques in two ways. First, ALICE allows a user to tag important features to construct an initial query. Second, ALICE uses active learning to iteratively refine a query by leveraging positive vs. negative labels.

ALICE is fundamentally different from clone detectors [40]–[44] in two ways. First, while clone detectors use a given internal representation such as a token string and a given similarity threshold to search for similar code, ALICE infers the commonality between positive examples, encodes them as a search template, and uses negative examples to decide what not to include in the template. Second, ALICE presents the search template as a logic query to a user, while clone detectors do not infer nor show a template to a user.

Logic-Programming-Based Techniques. JQuery is a code browsing tool [45] based on a logic query language. Users can interactively search by either typing a query in the UI or selecting a predefined template query. Hajiyev et al. present an efficient and scalable code querying tool [46] that allows programmers to explore the relation between different parts of a codebase. A number of techniques use logic programming as an abstraction for detecting code smells [47]–[49]. Many program analysis techniques abstract programs as logic facts and use Datalog rules, including pointer and call-graph analyses [50, 51], concurrency analyses [52, 53], datarace detection [54], security analyses [55], etc. Apposcopy [56] is a semantics-based Android malware detector, where a user provides a malware signature in Datalog. While ALICE and this line of research both use logic programs as an underlying representation, ALICE does not expect a user to know how to write a logic query nor requires having a set of pre-defined query templates. Instead, ALICE allows the user to interactively and incrementally build a search query using active ILP.

Interactive Synthesis. Some program synthesis techniques use input-output examples to infer a program and interactively refine its output [57, 58]. For instance, CodeHint [58] is a dynamic code synthesis tool that uses runtime traces, a partial program sketch specification, and a probabilistic model to generate candidate expressions. Interactive disambiguation interfaces [59, 60] aim to improve the accuracy of programming-by-example systems. ALICE is similar to these in leveraging *interactivity*, but these do not target code search, do not use ILP, and do not assess the impact of iterative labeling and annotations.

Machine Learning. Active learning is often used when unlabeled data may be abundant or easy to come by, but training labels are difficult, time-consuming, or expensive to obtain [24, 61]. An active learner may pose questions, usually in the form of unlabeled data instances to be labeled by an "oracle" (e.g., a human annotator). LOGAN-H is an ILP-based active learning approach [62]. It learns clauses by either asking the oracle to label examples (membership queries) or to answer an equivalence query. Such oracles were first proposed by Angluin in the query-based learning formalism [63]. Other approaches to inductive logic programming and relational learning are surveyed in De Raedt [13]. Alrajeh et al. integrate model checking and inductive learning to infer requirement specifications [18]. Other applications of ILP to software engineering include the work of Cohen [16, 64], to learn logical specifications from concrete program behavior. Because ultimately our approach is not concerned with finding the right hypothesis, and only with retrieving the right code examples, it can also be thought of as a transductive learning problem [65, 66].

IX. Conclusion

ALICE is the first approach that embodies the paradigm of active learning in the context of code search. Its algorithm is designed to leverage partial incremental feedback through tagging and labelling. ALICE demonstrates realtime performance in constructing a new search query. Study participants resonate with ALICE's *interactive* approach and find it easy to describe a desired code pattern without much effort. Extensive simulation shows that leveraging both positive and negative labels together can help achieve high precision and recall. Tagging features is also necessary for minimizing the size of initial search space. Our experimental results justify the design choice of ALICE, indicating that *interactivity* pays off—labeling a few in a spread out fashion is more effective than labeling many at a time.

Acknowledgment

Thanks to anonymous participants from the University of California, Los Angeles for their participation in the user study and to anonymous reviewers for their valuable feedback. This work is supported by NSF grants CCF-1764077, CCF-1527923, CCF-1460325, CCF-1837129, IIS-1633857, ONR grant N00014-18-1-2037, DARPA grant N66001-17-2-4032 and an Intel CAPA grant.

References

[1] S. Kim, K. Pan, and E. Whitehead Jr, "Memories of bug fixes," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2006, pp. 35–45.

[2] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1. IEEE, 2010, pp. 315–324.

[3] J. Park, M. Kim, B. Ray, and D.-H. Bae, "An empirical study of supplementary bug fixes," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press, 2012, pp. 40–49.

[4] J. R. Cordy, "The txl source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[5] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu, "Matching dependence-related queries in the system dependence graph," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 457–466.

[6] J. Andersen and J. L. Lawall, "Generic patch inference," *Automated software engineering*, vol. 17, no. 2, pp. 119–148, 2010.

[7] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 447–456.

[8] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: generating program transformations from an example," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 329–342, 2011.

[9] ——, "Lase: locating and applying systematic edits by learning from examples," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 502–511.

[10] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 404–415.

[11] T. Zhang, M. Song, J. Pinedo, and M. Kim, "Interactive code review for systematic changes," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 111–122.

[12] S. Muggleton and L. De Raedt, "Inductive logic programming: Theory and methods," *The Journal of Logic Programming*, vol. 19, pp. 629–679, 1994.

[13] L. D. Raedt, *Logical and relational learning*, ser. Cognitive Technologies. Springer, 2008. [Online]. Available: https://doi.org/10.1007/978-3-540-68856-3

[14] S. Haykin, *Neural Networks: A Comprehensive Foundation (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2007.

[15] S. Muggleton, C. Feng *et al.*, *Efficient induction of logic programs*. Citeseer, 1990.

[16] W. W. Cohen, "Recovering software specifications with inductive logic programming," in *AAAI*, vol. 94, 1994, pp. 1–4.

[17] I. Bratko and M. Grobelnik, "Inductive learning applied to program construction and verification," 1993.

[18] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel, "Elaborating requirements using model checking and inductive learning," *IEEE Transactions on Software Engineering*, vol. 39, no. 3, pp. 361–383, 2013.

[19] J. Starke, C. Luce, and J. Sillito, "Working with search results," in *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE'09. ICSE Workshop on*. IEEE, 2009, pp. 53–56.

[20] M. B. S. Ahmad and A. Cheung, "Automatically leveraging mapreduce frameworks for data-intensive applications," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 1205–1220.

[21] ——, "Leveraging parallel data processing frameworks with verified lifting," *arXiv preprint arXiv:1611.07623*, 2016.

[22] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 1589–1598.

[23] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar apis: An exploratory study," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 266–276.

[24] B. Settles, "Active learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 6, no. 1, pp. 1–114, 2012.

[25] L. De Raedt, *Logical and relational learning*. Springer Science & Business Media, 2008.

[26] P. A. Flach, "Simply logical intelligent reasoning by example," 1994.

[27] V. S. Costa, R. Rocha, and L. Damas, "The yap prolog system," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 5–34, 2012.

[28] R. K. Yin, *Case Study Research: Design and Methods (Applied Social Research Methods)*, fourth edition. ed. Sage Publications, 2008.

[29] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*. New York, NY, USA: ACM Press, 2005, pp. 117–125.

[30] N. Sahavechaphan and K. Claypool, "Xsnippet: Mining for sample code," *ACM Sigplan Notices*, vol. 41, no. 10, pp. 413–430, 2006.

[31] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 111–120.

[32] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 204–213.

[33] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 513–522.

[34] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 102–111.

[35] K. T. Stolee, S. Elbaum, and D. Dobos, "Solving the search for source code," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 3, p. 26, 2014.

[36] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006, pp. 681–682.

[37] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 87–98.

[38] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 933–944.

[39] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1069–1087, 2012.

[40] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code." in *OSdi*, vol. 4, no. 19, 2004, pp. 289–302.

[41] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[42] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 1157–1168.

[43] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.30

[44] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE, 2008, pp. 172–181.

[45] D. Janzen and K. De Volder, "Navigating and querying code without getting lost," in *Proceedings of the 2nd international conference on Aspect-oriented software development*. ACM, 2003, pp. 178–187.
[46] E. Hajiyev, M. Verbaere, and O. De Moor, "Codequest: Scalable source code queries with datalog," in *European Conference on Object-oriented Programming*. Springer, 2006, pp. 2–27.
[47] R. Wuyts, "Declarative reasoning about the structure of object-oriented systems," in *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*. IEEE, 1998, pp. 112–124.
[48] T. Tourwé and T. Mens, "Identifying refactoring opportunities using logic meta programming," in *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*. IEEE, 2003, pp. 91–100.
[49] Y.-G. Guéhéneuc and H. Albin-Amiot, "Using design patterns and constraints to automate the detection and correction of inter-class design defects," in *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on*. IEEE, 2001, pp. 296–305.
[50] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," *ACM SIGPLAN Notices*, vol. 44, no. 10, pp. 243–262, 2009.
[51] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: understanding object-sensitivity," in *ACM SIGPLAN Notices*, vol. 46, no. 1. ACM, 2011, pp. 17–30.
[52] M. Naik, A. Aiken, and J. Whaley, *Effective static race detection for Java*. ACM, 2006, vol. 41, no. 6.
[53] M. Naik, C.-S. Park, K. Sen, and D. Gay, "Effective static deadlock detection," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 386–396.
[54] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 462–473.
[55] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using pql: a program query language," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 365–383, 2005.
[56] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 576–587.
[57] C. Wang, A. Cheung, and R. Bodik, "Synthesizing highly expressive sql queries from input-output examples," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM, 2017, pp. 452–466.
[58] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen, "Codehint: Dynamic and interactive synthesis of code snippets," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 653–663.
[59] F. Li and H. V. Jagadish, "Nalir: an interactive natural language interface for querying relational databases," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 709–712.
[60] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. Zorn, and S. Gulwani, "User interaction models for disambiguation in programming by example," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 2015, pp. 291–301.
[61] S. Tong and E. Chang, "Support vector machine active learning for image retrieval," in *Proceedings of the ninth ACM international conference on Multimedia*. ACM, 2001, pp. 107–118.
[62] M. Arias, R. Khardon, and J. Maloberti, "Learning horn expressions with logan-h," *Journal of Machine Learning Research*, vol. 8, no. Mar, pp. 549–587, 2007.
[63] D. Angluin, "Queries and concept learning," *Machine learning*, vol. 2, no. 4, pp. 319–342, 1988.
[64] W. W. Cohen, "Inductive specification recovery: Understanding software by learning from example behaviors," *Automated Software Engineering*, vol. 2, no. 2, pp. 107–129, 1995.
[65] A. Gammerman, V. Vovk, and V. Vapnik, "Learning by transduction," in *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1998, pp. 148–155.
[66] O. Chapelle, B. Scholkopf, and A. Zien, "Semi-supervised learning," *IEEE Transactions on Neural Networks*, vol. 20, no. 3, pp. 542–542, 2009.