Investigating and Recommending Co-Changed Entities for JavaScript Programs

Zijian Jiang

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Application

Na Meng, Chair
Ali Butt
Francisco J. Servant Cortes

November 30, 2020
Blacksburg, Virginia

Keywords: Multi-entity edit, change suggestion, machine learning, JavaScript

Investigating and Recommending Co-Changed Entities for JavaScript Programs

Zijian Jiang

(ABSTRACT)

JavaScript (JS) is one of the most popular programming languages due to its flexibility and versatility, but debugging JS code is tedious and error-prone. In our research, we conducted an empirical study to characterize the relationship between co-changed software entities (e.g., functions and variables), and built a machine learning (ML)-based approach to recommend additional entity to edit given developers' code changes. Specifically, we first crawled 14,747 commits in 10 open-source projects; for each commit, we created one or more change dependency graphs (CDGs) to model the referencer-referencee relationship between co-changed entities. Next, we extracted the common subgraphs between CDGs to locate recurring co-change patterns between entities. Finally, based on those patterns, we extracted code features from co-changed entities and trained an ML model that recommends entities-to-change given a program commit.

According to our empirical investigation, (1) 50% of the crawled commits involve **multi-entity edits** (i.e., edits that touch multiple entities simultaneously); (2) three recurring patterns commonly exist in all projects; and (3) 80–90% of co-changed function pairs either invoke the same function(s), access the same variable(s), or contain similar statement(s); and (4) our ML-based approach CoRec recommended entity changes with high accuracy. This research will improve programmer productivity and software quality.

Investigating and Recommending Co-Changed Entities for JavaScript Programs

Zijian Jiang

(GENERAL AUDIENCE ABSTRACT)

This thesis introduced a tool CoRec which can provide co-change suggestions when JavaScript programmers fix a bug. A comprehensive empirical study was carried out on 14,747 multi-entity bug fixes in ten open-source JavaScript programs. We characterized the relationship between co-changed entities (e.g., functions and variables), and extracted the most popular change patterns, based on which we built a machine learning (ML)-based approach to recommend additional entity to edit given developers' code changes. Our empirical study shows that: (1) 50% of the crawled commits involve multi-entity edits (i.e., edits that touch multiple entities simultaneously); (2) three change patterns commonly exist in all ten projects; (3) 80-90% of co-changed function pairs in the 3 patterns either invoke the same function(s), access the same variable(s), or contain similar statement(s); and (4) our ML-based approach CoRec recommended entity changes with high accuracy. Our research will improve programmer productivity and software quality.

Dedication

I dedicate this to my parents and brother.

Acknowledgments

The project presented in this dissertation was carried out from 2019 to 2020 and couldn't have been possible without the help from numerous people. First, I would like to thank the Department of Computer Science at Virginia Tech for admitting me into the program, since I was pursuing my P.hD degree in Physics simultaneously. I wish to thank Prof. Na Meng who was an exceptional advisor over the last two years, without her acceptance I wouldn't have been able to start my CS master's work. Prof. Na has unlimited energy in research, her problem solving skills and her never ending knowledge of all things are a great resource for me to learn from, and they encouraged me to build programming language co-change tools as good as possible. Next, I wish to thank my committee members, Prof. Francisco Servant and Prof. Ali R. Butt not only for serving as my committee, but also for their brilliant comments and suggestions to my thesis defense. I wish to thank all my lab colleagues, Ye Wang, Ying Zhang, Chengyuan Wen, and Bowen Shen, for the discussions we have shared. I want to specifically thank Ye Wang, who explained his research and shared his code with me at the very beginning. I wish to thank my collaborator Cam Tenny from R2C company, who provided the typed-ast-util tool as our main JavaScript parser.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

JavaScript (JS) has become one of the most popular programming languages because it is lightweight, flexible, and powerful [5]. Developers use JS to build web pages and games. However, debugging JS code is usually time-consuming and error-prone. Various tools were built to provide automatic support for JS programming. For instance, linting tools (e.g., JSLint [16] and JSHint [15]) identify bad coding styles (e.g., the usage of "`==`"); similar to JS compilers (e.g., WebStorm [26] and Closure [7]), linting tools can detect syntactic errors (e.g., type errors and undefined variables) but reveal no semantic error [51]. Bug detection tools leverage static and/or dynamic analysis to identify code matches for any predefined bug pattern, in order to reveal data races [58], performance issues [70], API misuse [54], or DOM-related errors [52]. Change suggestion tools recommend fixes for logic errors and performance bugs either based on predefined patch patterns [27, 63] or based on fixes extracted from codebases [50] or technical discussion forums (e.g., StackOverflow [21, 48]).

The above approaches detect bugs, after such bugs are already introduced in source files. It is desirable to assist JS programming in advance, so fewer bugs can be introduced. Towards this roadmap, researchers (e.g., [73]) have proposed various approaches to recommend code co-changes, and such recommendations show promising results of reducing potential bugs (see Section 8.3 for details). However, none of existing tools characterize the co-change patterns between JS software entities, or recommend missing changes based on those patterns to complement or correct developers' edits.

As JS programs become complicated and software maintenance tasks become challenging, we believe that it is necessary to characterize and recommend co-changed entities. As the first step of proposing a JS co-change recommendation system, we need to analyze how JS source files are co-changed. Thus, in this paper, we conducted a study on 14,747 program commits from 10 open-source projects to investigate (1) what software entities are usually edited together, and (2) how those simultaneously edited entities are related. Specifically, given a program commit, we adopted ESprima [9] and typed-ast-util [23] to create ASTs for both the old and new versions of each changed JS file. We then extended a tree differencing tool—GumTree [30]—to compare ASTs and identify all edited entities (e.g., Deleted Classes (DC), Changed Functions (CF), and Added Variables (AV)). Next, we created one or more change dependency graphs (CDGs) for each commit by treating edited entities as nodes and linking entities that have referencer-referencee relations. Afterwards, we applied a subgraph isomorphism algorithm VF2 [28] to extract common subgraphs between CDGs and regarded those common subgraphs as recurring change patterns.

Our study shows that 7,144 explored commits (50%) edit multiple entities simultaneously. Among the multi-entity commits, our study reveals three most popular change patterns: ***CF$\xrightarrow{f}$CF** (i.e., one or more caller functions are changed together with one changed function that they commonly invoke), ***CF$\xrightarrow{f}$AF** (i.e., one or more functions are changed together to commonly invoke an added function), and ***CF$\xrightarrow{v}$AV** (i.e., one or more functions are changed together to commonly access an added variable). All these patterns imply that developers usually modify multiple JS functions simultaneously (i.e., *CF) to fulfill one maintenance task. In reality, as the co-changed functions contain different program contexts and experience divergent changes, it is possible that developers may forget to change all relevant functions at once and consequently introduce new bugs when editing code.

Based on the above-mentioned observations, we built a machine learning-based approach—CoRec—to recommend functions for co-change given the applied changes in a program commit. One insight obtained from our study is that co-changed functions (*CF) usually share certain commonality by containing similar code, accessing the same variable(s), or calling the same function(s). With the insight, we designed CoRec to extract 10 program features to characterize co-changed function pairs, and relied on those features to train a model. Afterwards, given a new program change, the model predicts whether any unchanged function should be changed as well and recommends changes if necessary. Our evaluation shows that CoRec can recommend co-change functions with 73–78% accuracy; it significantly outperformed a baseline technique that suggests co-changes purely based on software evolution.

To sum up, we made following research contributions in this paper:

- We conducted an empirical study to characterize the frequency and composition of multi-entity edits in JS programs, and to identify the syntactic and semantic relevance between frequently co-changed entities.
- We developed CoRec, a change recommendation approach that suggests co-changes based on (1) observed recurring change patterns and (2) identified commonality between co-changed functions (*CF).
- We compared CoRec with a baseline technique that recommends co-changes by mining software version history. Our evaluation shows that CoRec recommended changes in more scenarios and its recommendations achieved much higher accuracy.
- We investigated how sensitive CoRec is to the ML algorithm it uses. Compared with four other algorithms (i.e., J48 [57], Random Forest [45], Adaboost (default) [33], and Naïve Bayes [44]), Adaboost (with Random Forest as the "weak learner") [33] enables CoRec to work most effectively.

We envision CoRec to be used in the integrated development environments (IDE) for JS, code review systems, and version control systems. In this way, after developers make code changes or before they commit edits to software repositories, CoRec can help detect and fix incorrectly applied multi-entity edits.

In the sections below, we will first describe a motivating example (Section 2), and then introduce the background knowledge of our research (Section 3). Next, we will present the empirical study to characterize co-changes in JS programs (Section 4). Afterwards, we will explain our change recommendation approach CoRec (Section 5) and expound on the evaluation results (Section 6).

Chapter 2

A motivating example

In this section, we use a program revision on Node.js [19] to illustrate how CoRec predicts co-changes. Node.js is a server-side JS runtime environment, and Figure 2.1 shows the simplified program revision. In this revision, developers added a function `maybeCallback(...)` to check whether the pass-in parameter `cb` is a function, and modified seven functions in distinct ways to invoke the added function(e.g., changing `fs.write(...)` on line 10 and line 14).

We developed an ML-based tool CoRec that can recommend co-changes. For this example, given the added function `maybeCallback(...)` and a changed function `fs.write(...)`, CoRec extracts commonality between the changed function and any unchanged one, and relies on its ML model to predict whether the function pair should be changed together. Because `fs.write(...)` and `fs.read(...)`

- commonly access one variable `binding`,
- commonly invoke two functions: `makeCallback(...)` and `wrapper(...)`,
- declare the same parameters in sequence,
- have token-level similarity as 41%, and
- have statement-level similarity as 42%,

The pre-trained ML model inside CoRec considers the two methods to share sufficient commonality and thus recommends developers to also change `fs.read(...)` to invoke `maybeCallback(...)`. In this way, CoRec can suggest entities for change, which edits developers may otherwise miss.

As co-changes are complicated, developers may incompletely apply multi-entity edits, but CoRec can help programmers avoid incomplete edits. In this example, developers forgot to simultaneously change function `fs.read(...)` to also invoke `maybeCallback(...)`. Consequently, the multi-entity edit is incomplete and introduced a software bug. This inadvertently "*missed change*" stayed in the program for more than two years, until programmers fixed the bug by inserting a statement `callback = maybeCallback(callback);` to `fs.read(...)` [4]. Examining such completeness of above-mentioned multi-entity edit is challenging. This is because when developers forgot to change all methods for a new function invocation, there is no

```
+ function maybeCallback(cb) {
+   return typeof cb === 'function' ? cb : rethrow();
+ }

  fs.write = function(fd, buffer, offset, length, position, callback) {
-   callback = makeCallback(arguments[arguments.length - 1]);
    …
    req.oncomplete = wrapper;
    if (buffer instanceof Buffer) {
      …
+     callback = maybeCallback(callback);
      return binding.writeBuffer(…);
    }
    …
+   callback = maybeCallback(position);
    return binding.writeBuffer(fd, buffer, offset, …);
  }
```

```
  fs.read = function(fd, buffer, offset, length, position, callback) {
-   callback = makeCallback(arguments[arguments.length - 1]);
    …
  // an edit that developers forgot to apply:
  //+  callback = maybeCallback(callback);
    req.oncomplete = wrapper;
    binding.read(fd, buffer, offset, …);
  }
```

Figure 2.1: A program commit should add one function and change eight functions to invoke the newly added one. However, developers forgot to change one of the eight functions—`fs.read(...)` [1].

compilation or syntax error triggered, neither can existing bug detectors reveal the problem. CoRec is able to detect this bug, because its model is built on many other correct co-changes. Furthermore, with CoRec, developers can avoid this bug, before it is introduced.

Chapter 3

Concepts

This section first introduces concepts relevant to JS programming, and then describes the terminology used in our research.

ES6 and ES5. ECMA Script is the standardized name for JavaScript [3]. ES6 (or ECMAScript 2015) refers to version 6 of the ECMA Script programming language, which was released in 2015. ES6 is a major enhancement to ES5, and adds many more features intended to make large-scale software development easier. Major web browsers support some features of ES6. However, developers sometimes adopt transpilers (e.g., Babel [6]) to convert ES6 code into ES5, because ES5 is better supported on most browsers. Our research is applicable to both ES5 and ES6 programs.

Software Entity. When developers write JS code, they can define ***classes***, ***functions***, and ***variables*** in multiple alternative ways. For instance, a class can be defined with a class expression (see Figure 3.1 (a)) or a class declaration (see Figure 3.1 (b)). Similarly, a function can be defined with a function expression or function declaration. A variable can be defined with a variable declaration statement; the statement can either use keyword `const` to declare a constant variable, or use `let` or `var` to declare a non-constant variable. Outside the definition of classes, functions, and variables, developers can also define independent ***statement blocks*** (i.e., blocks of statements) as needed, which are units of code that can be executed. Therefore, in our research, we use ***program entities*** or ***software entities*** to refer to the JS classes, functions, variables, and statement blocks that developers usually define.

Edited Entity. When maintaining JS software, developers may add, delete, or change one or more entities. Therefore, as with prior work [59], we defined a set of ***edited entities*** to describe the possible entity-level edits, including *Added Class* (**AC**), *Deleted Class* (**DC**), *Added Function* (**AF**), *Deleted Function* (**DF**), *Changed Function* (**CF**), *Added Variable* (**AV**), *Deleted Variable* (**DV**), *Changed Variable* (**CV**), *Added Statement Block* (**AB**), *Deleted Statement Block* (**DB**), and *Changed Statement Block* (**CB**). For example, if a new class is declared to have a constructor and some other methods, we consider the revision to have one AC, multiple AFs, and one or more AV (depending on how many fields are defined in the constructor).

Multi-Entity Edit and CDG. As with prior work[69], we use ***multi-entity edit*** to refer to any commit that has two or more ***edited entities***. We use ***change dependency***

```javascript
const Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  area() {
    return this.height * this.width;
  }
};

console.log(new Rectangle(5, 8).area());
```

(a)

```javascript
class Rectangle{
  constructor(height, width) {
    this.area = height * width;
  }
}

console.log(new Rectangle(5, 8).area);
```

(b)

Figure 3.1: A JS class can be defined with an expression (see (a)) or a declaration (see (b)).

graph (CDG) to visualize the the relationship between co-changed entities in a commit. Specifically, each CDG has at least two nodes and one edge. Each node represents an edited entity, and each edge represents the referencer-referencee relationship between entities (e.g., a function calls another function). Namely, if an edited entity E_1 refers to another edited entity E_2, we say E_1 depends on E_2. For each program commit, we may create zero, one, or multiple CDGs.

Chapter 4

Characterization Study

This section introduces our study methodology (Section 4.1) and explains our empirical findings (Section 4.2). The purpose of this characterization study is to identify ***recurring change pattern (RCP)*** of JS programs. A RCP is a CDG subgraph that is commonly shared by the CDGs from at least two distinct commits. RCPs define different types of edits, and serve as the templates of co-change rules. Our approach in Section 5 mines concrete co-change rules for the most common RCPs.

4.1 Study Methodology

We implemented a tool to automate the analysis. Given a set of program commits in JS repositories, our tool first characterizes each commit by extracting the edited entities (Section 4.1.1) and constructing CDG(s) (Section 4.1.2). Next, it compares CDGs across commits to identify RCPs (Section 4.1.3).

4.1.1 Extraction of Edited Entities

Given a program commit c, this step first locates the old and new versions of each edited JS file. For each edited file (f_o, f_n), this step adopts Esprima [9] and typed-ast-util [23] to generate abstract syntax trees (ast_o, ast_n). Specifically, Esprima is a high performance, standard-compliant JavaScript parser that supports the syntax of both ES5 and ES6; however, it cannot infer the static type binding information of any referenced class, function, or variable. Meanwhile, given JS files and the project's `package.json` file, typed-ast-util produces ASTs annotated with structured representations of TypeScript types, which information can facilitate us to precisely identify the referencer-referencee relationship between edited entities. We decided to use both tools for two reasons. First, when a project has `package.json` file, we rely on Esprima to identify the code range and token information for each parsed AST node, and rely on typed-ast-util to attach relevant type information to those nodes. Second, when a project has no `package.json` file, Esprima is still used to generate ASTs but we defined a heuristic approach (to be discussed later in Section 4.1.2) to identify the referencer-referencee relationship between entities with best efforts.

For each pair of generated ASTs (ast_o, ast_n), this step extracts the entity sets (ES_1, ES_2).

The biggest technical challenge here is that due to the flexibility of JS programming, there is ambiguity related to the entity types of some code blocks. For instance, a variable declaration statement can be treated as a variable-typed entity or a statement block. To eliminate ambiguity and avoid any overlap between differently typed entities, we classified and extracted entities in the following way:

- A code block is treated as a function definition if it satisfies either of the following two criteria. First, the AST node type is "`FunctionDeclaration`" or "`MethodDefinition`". Second, (1) the block is either a "`VariableDeclaration`" statement (e.g., `const getRectArea = function(...){...};`) or an"`Assignment`" expression (e.g., `val = foo()`); and (2) the right-side operand is either "`FunctionExpression`", or "`CallExpression`" that outputs another function as return value of the called function. In particular, if any defined function has its prototype property explicitly referenced (e.g., `foo.prototype` in code) or is used as a constructor to create any object (e.g., `var obj = new foo(...)`), we reclassify the function definition as a class definition, because the function usage is actually more like the usage of a class.

- A code block is considered to be a class definition if it satisfies either of the following two criteria. First, the block uses keyword `class`. Second, the block defines a function, while the codebase either references the function's prototype or uses the function as a constructor to create any object.

- A code block is treated as a variable declaration if (1) it is either a "`VariableDeclaration`" statement or an "`Assignment`" expression, (2) it does not define a function or class, (3) it does not belong to the definition of any function (except for class constructors), and (4) it does not declare a required module (e.g., `const config=require('path/to/file')`). Particularly, when variable declaration is an assignment inside a class constructor (e.g., `this.name = name`), it is similar to field declaration in Java.

- A code block is treated as a statement block if (1) it purely contains statements, (2) it does not define any class, function, or variable, and (3) it does not belong to the definition of any class or function.

To identify any edited entity between ES_1 and ES_2, we first matched the definitions of functions, variables, and classes across entity sets based on their signatures. If any of these entities (e.g., a function definition) solely exists in ES_1, an entity-level deletion (e.g., DF) is inferred; if an entity (e.g., a variable definition) solely exists in ES_2, an entity-level insertion (e.g., AV) is inferred. Next, for each pair of matched entities, we further exploited a fine-grained AST differencing tool—GumTree [30]—to identify expression-level and statement-level edits. If any edit is reported, we inferred an entity-level change (e.g., CF or CV). Additionally, we matched statement blocks across entity sets based on their string similarities. Namely, if a statement block $b_1 \in ES_1$ has the longest common subsequence with a block $b_2 \in ES_2$ and the string similarity is above 50%, we considered the two blocks to match. Furthermore, if the similarity between two matched blocks is not 100%, we inferred a block-level change CB.

4.1.2 CDG Construction

For each program commit, we built CDGs by representing the edited entities as nodes, and by connecting edited entities with directed edges if they have either of the following two types of relationship:

- **Access.** If an entity E_1 accesses another entity E_2 (i.e., by reading/writing a variable, invoking a function, or using a class), we consider E_1 to be dependent on E_2.
- **Containment.** If the code region of E_1 is fully covered by that of E_2, we consider E_1 to be dependent on E_2.

The technical challenge here is how to identify the relationship between edited entities. We relied on ESprima's outputs to compare code regions between edited entities and to identify the containment relationship. Additionally, when `package.json` file is available, we leveraged the type binding information inferred by typed-ast-util to identify the access relationship. For instance, if there is a function call `bar()` inside an entity E_1 while `bar()` is defined by a JS module `f2`, then typed-ast-util can resolve the fully qualified name of the callee function as `f2.bar()`. Such resolution enables us to effectively link edited entities no matter whether they are defined in the same module (i.e., JS file) or not.

Since some JS projects have no `package.json` file, we could not adopt typed-ast-util to resolve bindings in such scenarios. Therefore, we also built a simpler but more applicable approach to automatically speculate the type binding information of accessed entities as much as possible. Specifically, suppose that file `f1` defines E_1 to access E_2. To resolve E_2 and link E_1 with E_2's definition, this intuitive approach first scans all entities defined in `f1` to see whether there is any E_2 definition locally. If not, this approach further examines all `require` and `import` statements in `f1`, and checks whether any required or imported module defines a corresponding entity with E_2's name; if so, this approach links E_1 with the retrieved E_2's definition. Compared with typed-ast-util, our approach is less precise because it cannot infer the return type of any invoked function. For instance, if we have `const foo = bar()` where `bar()` returns a function, our approach simply assumes `foo` to be a variable instead of a function. Consequently, our approach is unable to link `foo`'s definition with any of its invocations.

In Figure 4.1 and Figure 4.2, we present a simplified program revision to Meteor [17] and the related CDG. According to Figure 4.1, the program commit modifies file `tools/buildmessage.js` by defining a new function `spaces(...)` and updating an existing function `capture(...)` to invoke the new function. It also changes file `tools/commands-package.js` by updating the function invocation of `capture(...)` inside a statement block (i.e., `main.registerCommand(...);`). Given the old and new versions of both edited JS files, our approach can construct the CDG shown in Figure 4.2. In this CDG, each directed edge starts from a dependent entity E_1, and points to the entity on which E_1 depends. Each involved function, variable, or class has

```
+ var spaces = function (n) {
+   return _.times(n, function() { return ' ' }).join('');
+ };

  var capture = function (option, f) {
    …
-   console.log("START CAPTURE", nestingLevel, options.title, "took " + (end - start));
+   console.log(spaces(nestingLevel * 2), "START CAPTURE", nestingLevel, options.title, "took " + (end - start));
    …
}
```

tools/buildmessage.js

```
  main.registerCommand({…}, function (options) {
    …
-   var messages = buildmessage.capture(function () {
+   var messages = buildmessage.capture({ title: 'Combining constraints' }, function(){
      allPackages = project.getCurrentCombinedConstraints();
  });
    …
```

tools/commands-packages.js

Figure 4.1: A simplified program commit that adds a function `spaces(...)`, changes a function `capture(...)`, and changes a statement block [11]

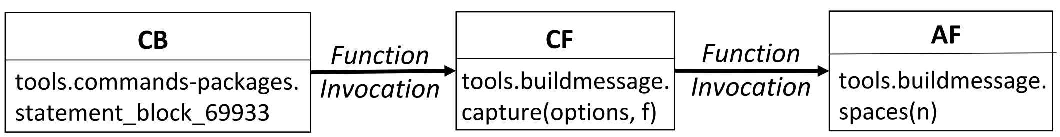


Figure 4.2: The CDG corresponding to the program commit shown in Figure 4.1

its fully qualified name included in the CDG for clarity. As statement blocks have no fully qualified names, we created a unique identifier for each block with (1) the module name (e.g., `tools.commands-packages`) and (2) index of the block's first character in that module (e.g., `69933`).

4.1.3 Extraction of Recurring Change Patterns (RCP)

As with prior work [69], we extracted RCPs by comparing CDGs across program commits. Intuitively, given a CDG g_1 from commit c_1 and the CDG g_2 from commit c_2, we matched nodes based on their edit-entity labels (e.g., AF) while ignoring the concrete code details (e.g., `tools.buildmessage.spaces(n)` in Figure 4.2). We then established edge matches based on those node matches. Namely, two edges are matched only if they have matching starting nodes and matching ending nodes. Next, based on all established matches, we identified the largest common subgraph between g_1 and g_2 using the off-the-shelf subgraph isomorphism algorithm VF2 [28]. Such largest common subgraphs are considered as RCPs because they commonly exist in CDGs of different commits.

Table 4.1: Subject projects

Project	Description	# of KLOC	# of Fixes	# of Edited Entities
Node.js	Node.js [19] is a cross-platform JS runtime environment. It executes JS code outside of a browser.	1,755	2,701	11,287
Meteor	Meteor [17] is an ultra-simple environment for building modern web applications.	255	3,011	10,274
Ghost	Ghost [13] is the most popular open-source and headless Node.js content management system (CMS) for professional publishing.	115	1,263	5,142
Habitica	Habitica [14] is a habit building program that treats people's life like a Role Playing Game.	129	1,858	6,116
PDF.js	PDF.js [18] is a PDF viewer that is built with HTML5.	104	1,754	4,255
React	React [10] is a JS library for building user interfaces.	286	1,050	4,415
Serverless	Serverless [20] is a framework used to build applications comprised of microservices that run in response to events.	63	1,110	3,846
Webpack	Webpack [25] is a module bundler, which mainly bundles JS files for usage in a browser. assets.	37	1,099	3,699
Storybook	Storybook [22] is a development environment for UI components.	43	528	2,277
Electron	Electron [8] is a framework that supports developers to write cross-platform desktop applications using JS, HTML, and CSS.	35	673	1,898

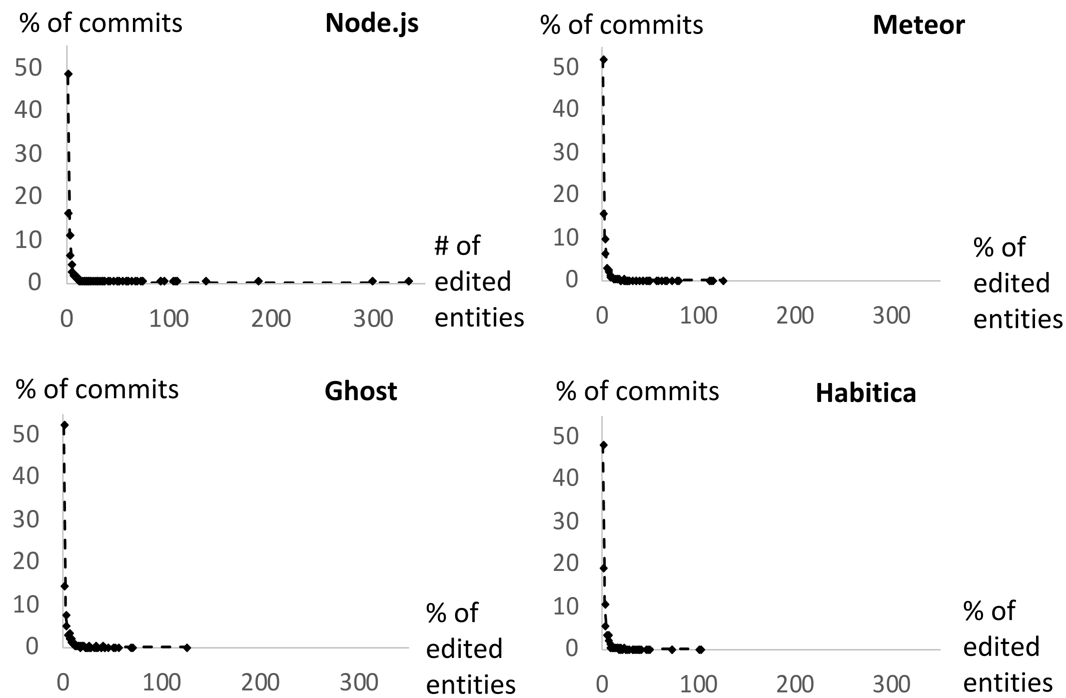


Figure 4.3: Commit distributions based on the number of edited entities each of them contains

4.2 Empirical Findings

We analyzed the bug-fixing commits of 10 open-source projects, as shown in Table 4.1. We chose these projects because they are popularly used and from different application domains. We mainly focused on bug-fixing commits because based on our experience, developers are more likely to check in related changes for one software maintenance task (i.e., bug fixing) in each of such commits. To identify bug-fixing commits, we searched for commits whose commit messages contain any of the following keywords: "bug", "fix", "error", "adjust", and "failure".

In Table 4.1, column **# of KLOC** presents the code size of each project (i.e., the number of kilo lines of code (KLOC)). Column **# of Fixes** reports the number of bug-fixing commits identified via our keyword-based search. Column **# of Edited Entities** reports the number of edited entities in the bug-fixing commits extracted from each project repository. According to the table, the code size varies significantly from 35 KLOC to 1755 KLOC. Among the 10 projects, 528–3011 bug-fixing commits were extracted, and 1898–11287 edited entities were included for each project. Within these projects, only Node.js has no `package.json` file, so we adopted our intuitive approach mentioned in Section 4.1.2 to link edited entities. For the remaining nine projects, as they all have `package.json` files, we leveraged the type binding information inferred by typed-util-ast to connect edited entities.

Table 4.2: Multi-entity edits and created CDGs

Project	# of Multi-Entity Edits	# of Multi-Entity Edits with CDG(s) Extracted	% of Multi-Entity Edits with CDG(s) Extracted
Node.js	1401	785	56%
Metoer	1445	670	46%
Ghost	604	356	59%
Habitica	962	345	36%
PDF.js	711	372	52%
React	538	320	60%
Serverless	480	171	36%
Webpack	483	253	52%
Storybook	243	119	49%
Electron	277	123	44%

4.2.1 Commit Distributions Based on The Number of Edited Entities

We first clustered commits based on the number of edited entities they contain. Due to the space limit, we are unable to present all distribution figures in this paper. Because the commit distributions of different projects are very similar to each other, we illustrate such distributions for four projects in Figure 4.3. Among the 10 projects, 41–52% of commits are multi-entity edits. Specifically, 15–19% of commits involve two-entity edits, and 7–10% of commits are three-entity edits. The number of commits decreases as the number of edited entities increases. The maximum number of edited entities appears in Node.js, where a single commit modifies 335 entities. We manually checked the commit on GitHub [2], and found that four JS files were added and three other JS files were changed to implement HTTP/2.

Finding 1: *Among the 10 studied projects, 41–52% of commits are multi-entity edits. It indicates the necessity of our research to characterize multi-entity edits and recommend changes accordingly.*

4.2.2 Commit Distributions Based on The Number of CDGs

We further clustered multi-entity edits based on the number of CDGs constructed for each commit. As shown in Table 4.2, our approach created CDGs for 36–60% of the multi-entity edits in distinct projects. On average, 49% of multi-entity edits contain at least one CDG. Due to the complexity and flexibility of the JS programming language, it is very challenging to statically infer all possible referencer-referencee relationship between JS

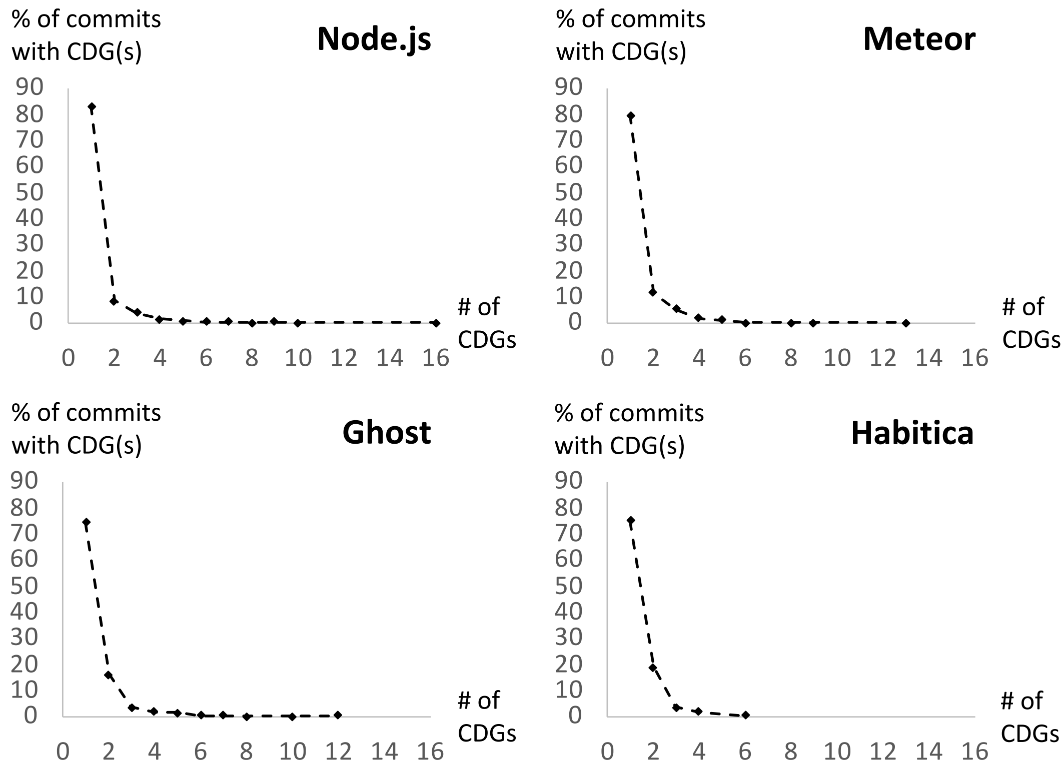


Figure 4.4: The distributions of commits with multi-entity edits based on the number of CDGs

entities. Therefore, the actual percentage of edits that contain related co-changed entities can be even higher than our measurement.

Among the commits with CDG(s) extracted, we investigated the commit distributions based on the number of CDGs created for each commit. Since the commit distributions of different projects are all similar to each other, Figure 4.4 presents the distributions for only four of the projects: Node.js, Meteor, Ghost, and Habitica. As illustrated by the figure, the number of commits decreases significantly as the number of CDGs increases. Among all 10 projects, 73–81% of commits contain single CDGs, 9–18% of commits have two CDGs extracted, and 3–7% of commits have three CDGs. The commit with the largest number of CDGs constructed (i.e., 16) is the one with the maximum number of edited entities in Node.js as mentioned above in Section 4.2.1.

Finding 2: *For 36–60% of multi-entity edits in the studied projects, our approach created at least one CDG for each commit. It means that many simultaneously edited entities are syntactically relevant to each other.*

Table 4.3: Recurring change patterns and their matches

Projects	# of RCPs	# of Commits with RCP Matches	# of Subgraphs Matching the RCPs
Node.js	221	782	2385
Metoer	200	658	1719
Ghost	133	351	1223
Habitica	116	339	706
PDF.js	86	367	640
React	110	316	899
Serverless	57	164	372
Webpack	80	243	583
Storybook	42	113	337
Electron	35	117	228

4.2.3 Identified RCPs

By comparing CDGs of distinct commits within the same project repository, we identified RCPs in all projects. As listed in Table 4.3, 35–221 RCPs are extracted from individual projects. In each project, there are 113–782 commits that contain matches for RCPs. In particular, each project has 228–2385 subgraphs matching RCPs. By comparing this table with Table 4.2, we found that 95–100% of the commits with CDGs extracted have matches for RCPs. It means that if one or more CDGs can be built for a commit, the commit is very likely to share common subgraphs with some other commits. In other words, simultaneously edited entities are usually correlated with each other in a fixed number of ways. If we can characterize the frequently occurring relationship between co-changed entities, we may be able to leverage such characterization to predict co-changes or reveal missing changes.

By accumulating the subgraph matches for RCPs across projects, we identified five most popular RCPs, as shown in Figure 4.5. Here, **P1** means that when a callee function is changed, one or more of its caller functions are also changed. **P2** means that when a new function is added, one or more existing functions are changed to invoke that new function. **P3** shows that when a new variable is added, one or more existing functions are changed to read/write the new variable. **P4** presents that when a new variable is added, one or more new functions are added to read/write the new variable. **P5** implies that when a function is changed, one or more existing statement blocks invoking the function are also changed. Interestingly, the top three patterns commonly exist in all 10 projects, while the other two patterns do not exist in some of the projects. The top three patterns all involve simultaneously changed functions.

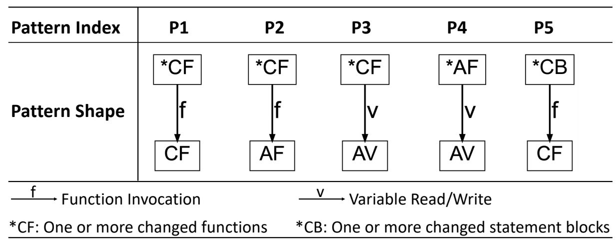


Figure 4.5: Five most popular recurring change patterns among the 10 projects

Finding 3: *Among the commits with CDGs extracted, 95–100% of commits have matches for mined RCPs. In particular, the most popular three RCPs all involve simultaneously changed functions.*

4.2.4 Case Studies for The Three Most Popular RCPs

To identify any research opportunity to recommend changes based on the mined RCPs, we conducted a case study for each of the top three popular RCPs. In each case study, we randomly picked 20 commits matching the corresponding pattern, while each commit has two or more co-changed functions (e.g., *CF) referencing another edited entity. We then inspected the co-changed functions in each commit, to decide whether they share any commonality that may indicate their simultaneous changes. In our manual analysis, we mainly focused on four types of commonality:

- **FI**: The co-changed functions commonly invoke one or more *peer functions* of the depended entity E (i.e., CF in P1, AF in P2, and AV in P3). Here, **peer function** is any function that is defined in the same file as E.
- **VA**: The co-changed functions commonly access one or more *peer variables* of the depended entity E. Here, **peer variable** is any variable that is defined in the same file as E.
- **ST**: The co-changed functions commonly share at least 50% of their token sequences. We calculated the percentage with the longest common subsequence algorithm between two token strings.
- **SS**: The co-changed functions commonly share at least 50% of their statements. We computed the percentage by recognizing identical statements between two given functions `f1(...)` and `f2(...)`. Assume that the two functions separately contain n_1 and n_2 statements, and the number of common statements is n_3. Then the percentage is calculated as

$$\frac{n_3 * 2}{n_1 + n_2} \times 100\% \tag{4.1}$$

Table 4.4: Commonality observed between the co-changed functions

Case Study	Commonality				No Commonality
	FI	VA	ST	SS	
I (for P1: *CF$\xrightarrow{f}$CF)	8	5	7	4	4
II (for P2: *CF$\xrightarrow{f}$AF)	12	7	8	6	2
III (for P3: *CF$\xrightarrow{v}$AV)	6	13	6	5	3

According to Table 4.4, 80–90% of co-changed functions share certain commonality with each other. There are only 2–4 commits in each study where the co-changed functions share nothing in common. Particularly, in the first case study, the FI commonality exists in eight commits, VA exists in five commits, ST occurs in seven commits, and SS occurs in four commits. The summation of these commonality occurrences is larger than 20, because the co-changed functions in some commits share more than one type of commonality. Additionally, the occurrence rates of the four types of commonality are different between case studies. For instance, FI has 8 occurrences in the first case study; it occurs in 12 commits of the second study and occurs in only 6 commits of the third study. As another example, most commits (i.e., 13) in the third study share the VA commonality, while there are only 5 commits in the first study having such commonality. The observed differences between our case studies imply that when developers apply multi-entity edits matching different RCPs, the commonality shared between co-changed functions also varies.

Finding 4: *When inspecting the relationship between co-changed functions in three case studies, we found that these functions usually share certain commonality. This indicates great opportunities for developing co-change recommendation approaches.*

Chapter 5

Our Change Recommendation Approach: CoRec

In our characterization study (see Section 4), we identified three most popular RCPs: $^*\mathrm{CF}\xrightarrow{f}\mathrm{CF}$, $^*\mathrm{CF}\xrightarrow{f}\mathrm{AF}$, and $^*\mathrm{CF}\xrightarrow{v}\mathrm{AV}$. In all these patterns, there is at least one or more changed functions (i.e., *CF) that references another edited entity E (i.e., CF, AF, or AV). In the scenarios when two or more co-changed functions commonly depend on E, we also observed certain commonality between those functions. This section introduces our recommendation system—CoRec—which is developed based on the above-mentioned insights. As shown in Figure 5.1, CoRec has three phases. In the following sections (Sections 5.1-5.3), we explain one phase in each section.

5.1 Phase I: Commit Crawling

Given the software repository of a project P, Phase I crawls commits to locate any data usable for machine learning. Specifically, for each commit in the repository, this phase reuses the first two steps of our study approach (see Sections 4.1.1 and 4.1.2) to extract edited entities and to create CDGs. If a commit c has any subgraph matching P1, P2, or P3, this phase recognizes the entity E_m matching E (i.e., an entity matching CF in P1, matching AF in P2, or matching AV in P3) and any co-changed function matching *CF. We denote these co-changed function(s) with $CF\_Set = \{cf_1, cf_2, \ldots\}$, and denote the unchanged function(s) in edited JS files from the same commit with $UF\_Set = \{uf_1, uf_2, \ldots\}$. If CF_Set has at least two co-changed functions, CoRec considers the commit to be usable for model training and passes E_m, CF_Set, and UF_Set to the next phase.

5.2 Phase II: Training

This phase has two major inputs: the software repository of program P, and the extracted data from each relevant commit (i.e., E_m, CF_Set, and UF_Set). In this phase, CoRec first creates positive and negative training samples, and then extracts features for each sample. Next, CoRec trains a machine learning model by applying Adaboost (with Ran-

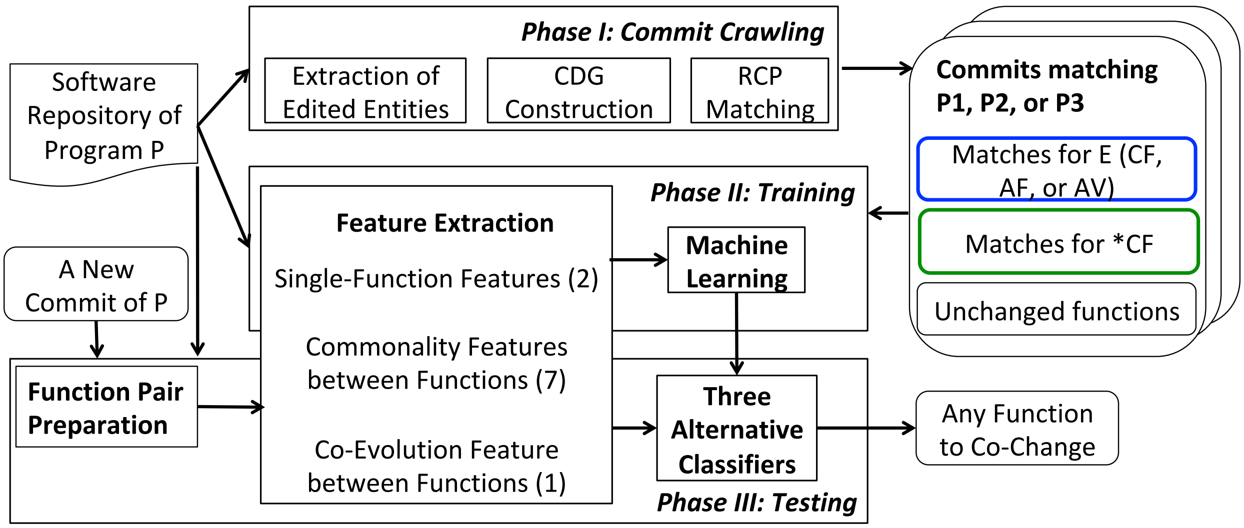


Figure 5.1: CoRec consists of three phases: commit crawling, training, and testing

Table 5.1: A list of features extracted for function pair (f_1, f_2)

Id	Feature	Id	Feature
1	Number of E_m-relevant parameter types in f_2	6	Whether f_1 and f_2 have the same return type
2	Whether f_2 has the E_m-related type	7	Whether f_1 and f_2 are defined in the same way
3	Number of common peer variables	8	Token similarity
4	Number of common peer functions	9	Statement similarity
5	Number of common parameter types	10	Co-evolution frequency

dom Forest as the "weak learner") [33] to the extracted features. Specifically, to create positive samples, CoRec enumerates all possible function pairs in CF_Set, because each pair of these functions were co-changed with E_m. We represent the positive samples with $Pos = \{(cf_1, cf_2), (cf_2, cf_1), (cf_1, cf_3), \ldots\}$. To create negative samples, CoRec pairs up each changed function $cf \in CF\_Set$ with an unchanged function $uf \in UF\_Set$, because each of such function pairs were not co-changed. Thus, we denote the negative samples as $Neg = \{(cf_1, uf_1), (uf_1, cf_1), (cf_1, uf_2), \ldots\}$. By preparing positive and negative samples in this way, given certain pair of functions, we expect the trained model to predict whether the functions should be co-changed or not.

CoRec extracts 10 features for each sample. As illustrated in Figure 5.1, two features reflect code characteristics of the second function in the pair, seven features capture the code commonalities between functions, and one feature focuses on the co-evolution relationship between functions. Table 5.1 presents more details of each feature. Specifically, the 1^{st} and 2^{nd} features are about the relationship between f_2 and E_m. Their values are calculated as below:

- When E_m is CF or AF, the 1^{st} feature records the number of types used in f_2 that match any declared parameter type of E_m. Intuitively, the more type matches, the more likely that f_2 should be co-changed with E_m. The 2^{nd} feature checks whether the code in f_2 uses the return type of E_m.

- When E_m is AV, the 1^{st} feature is set to zero, because there is no parameter type involved in variable declaration. The 2^{nd} feature checks whether the code in f_2 uses the data type of the newly added variable.

The 3^{rd} and 4^{th} features were calculated in similar ways. Specifically, depending on which JS file defines E_m, CoRec locates peer variables (i.e., variables defined within the same file as E_m) and peer functions (i.e., functions defined in the same file). Next, CoRec identifies the accessed peer variables (or peer functions) by each function in the pair, and intersects the sets of both functions to count the commonly accessed peer variables (or peer functions). Additionally, the 7^{th} feature checks whether f_1 and f_2 are defined in the same manner. In our research, we consider the following five ways to define functions:

(1) via `FunctionDeclaration`, e.g., `function foo(...){...}`,

(2) via `VariableDeclaration`, e.g., `var foo = function(...){...}`,

(3) via `MethodDefinition`, e.g., `Class A {foo(...){...}}`,

(4) via `PrototypeFunction` to extend the prototype of an object or a function, e.g., `x.prototype.foo = function(...){...}`, and

(5) via certain `exports`-related statements, e.g., `exports.foo = function(...){...}` and `module.exports = {foo: function(...){...}}`.

If f_1 and f_2 are defined in the same way, the 7^{th} feature is set to `true`. Finally, the 10^{th} feature assesses in the commit history, how many times the pair of functions were changed together before the current commit. Inspired by prior work [73], we believe that the more often two functions were co-changed in history, the more likely that they are co-changed in the current or future commits.

Depending on the type of E_m, CoRec takes in extracted features to actually train three independent classifiers, with each classifier corresponding to one pattern among P1–P3. For instance, one classifier corresponds to P1: $*\text{CF} \xrightarrow{f} \text{CF}$. Namely, when E_m is CF and one of its caller functions cf is also changed, this classifier predicts whether there is any unchanged function uf that invokes E_m and should be also changed. The other two classifiers separately predict functions for co-change based on P2 and P3. We consider these three binary-class classifiers as an integrated machine learning model, because all of them can take in features from one program commit and related software version history, in order to recommend co-changed functions when possible.

5.3 Phase III: Testing

This phase takes in two inputs—a new program commit c_n and the related software version history, and recommends any unchanged function that should have been changed by that commit. Specifically, given c_n, CoRec reuses the steps of Phase I (see Section 5.1) to locate E_m, CF_Set, and UF_Set. CoRec then pairs up every changed function $cf \in CF\_Set$ with every unchanged one $uf \in UF\_Set$, obtaining a set of candidate function pairs $Candi = \{(cf_1, uf_1), (uf_1, cf_1), (cf_1, uf_2), \ldots\}$. Next, CoRec extracts features for each candidate p and sends the features to a pre-trained classifier depending on E_m's type. If the classifier predicts the function pair to have co-change relationship, CoRec recommends developers to also modify the unchanged function in p.

Chapter 6

Evaluation

In this section, we first introduce our experiment setting (Section 6.1) and the metrics used to evaluate CoRec's effectiveness (Section 6.2). Then we expound on the effectiveness comparison between CoRec and a well known co-change recommendation tool ROSE [73] (Section 6.3). Next, we present the comparison between CoRec and a variant approach that trains one unified classifier instead of three distinct classifiers to recommend changes (Section 6.4). Finally, we explain our investigation with alternative ML algorithms and present CoRec's sensitivity to the the adopted ML algorithms (Section 6.5).

6.1 Experiment Setting

We mined repositories of the 10 open-source projects introduced in Section 4, and found three distinct sets of commits in each project that are potentially usable for model training and testing. As shown in Table 6.1, in total, we found 280 commits matching P1, 232 commits matching P2, and 182 commits matching P3. Each of these pattern matches has at least two co-changed functions (*CF) depending on E_m. In our evaluation, for each data set of each project, we could use part of the data to train a classifier and use the remaining data to test the trained classifier. Because Storybook and Electron have few commits, we excluded them from our evaluation and simply used the identified commits of the other eight projects to train and test classifiers.

Table 6.1: Numbers of commits that are potentially usable for model training and testing

Project	**# of Commits Matching P1**	**# of Commits Matching P2**	**# of Commits Matching P3**
Node.js	92	77	65
Meteor	67	59	39
Ghost	21	24	28
Habitica	11	8	5
PDF.js	14	12	14
React	18	12	5
Serverless	26	12	8
Webpack	22	24	8
Storybook	2	1	4
Electron	7	3	6
Sum	**280**	**232**	**182**

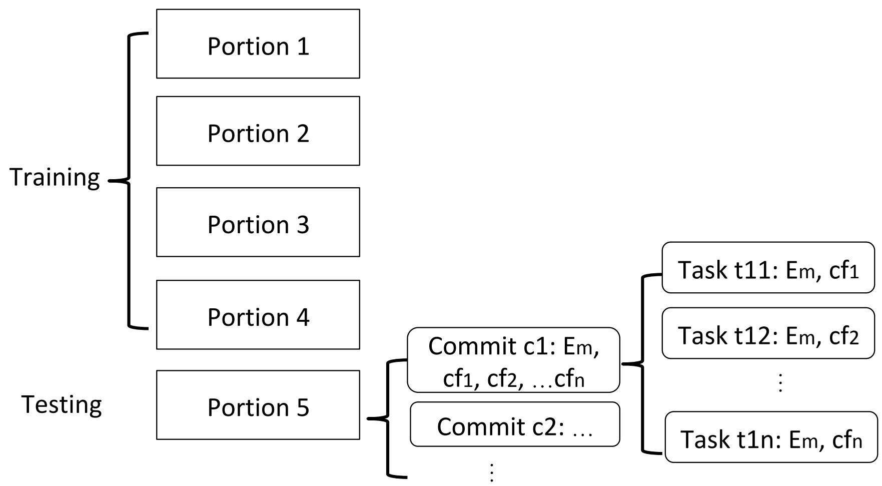


Figure 6.1: Typical data processing for each fold of the five-fold cross validation

We conducted five-fold cross validation to evaluate CoRec's effectiveness. Specifically, for every data set of each project, we split the mined commits into five portions roughly evenly, and each fold uses a unique portion for testing. As shown in Figure 6.1, in each fold, we used four portions to train a classifier, and revised the fifth portion to test the classifier. In particular, for each commit in the fifth portion, suppose that there are n co-changed functions (*CF) depending on E_m, i.e., $CF\_Set = \{cf_1, cf_2, \ldots, cf_n\}$. We created n prediction tasks such that in each task, there is only one changed function cf_i ($i \in [1, n]$) co-applied with E_m while all the other changed functions are kept unchanged. We regarded the $(n-1)$ functions as ground truth to assess how accurately CoRec can recommend co-changed functions given E_m and cf_i.

6.2 Metrics

Four metrics were defined by us to measure the capability of a tool capability to recommend co-changed functions: coverage, precision, recall, and F-score. The weighted average was also defined to measure the overall effectiveness among all subject projects.

Coverage (Cov) is the percentage of tasks for which a tool can provide suggestion. Given a task, a tool may or may not recommend any change to complement the already-applied edit, so this metric assesses the tool applicability. Intuitively, the more tasks for which a tool can recommend one or more changes, the more applicable this tool is.

$$Cov = \frac{\text{\# of tasks with a tool's suggestion}}{\text{Total \# of tasks}} \times 100\% \tag{6.1}$$

Coverage varies within [0%, 100%]. If a tool always recommends some change(s) given a task, its coverage is 100%. All our later evaluations for precision, recall, and F-score are limited to the tasks covered by a tool. For instance, suppose that given 100 tasks, a tool can recommend changes for 10 tasks. Then the tool's coverage is $10/100 = 10\%$, and the evaluations for other metrics are based on the 10 instead of 100 tasks.

Precision (Pre) measures among all recommendations by a tool, how many of them are correct:

$$Pre = \frac{\#\text{ of correct recommendations}}{\text{Total }\#\text{ of recommendations by a tool}} \times 100\% \tag{6.2}$$

This metric evaluates how precisely a tool recommends changes. If all suggestions by a tool are contained by the ground truth, the precision is 100%.

Recall (Rec) measures among all the expected recommendations, how many of them are actually reported by a tool:

$$Rec = \frac{\#\text{ of correct recommendations by a tool}}{\text{Total }\#\text{ of expected recommendations}} \times 100\% \tag{6.3}$$

This metric assesses how effectively a tool retrieves the expected co-changed functions. Intuitively, if all expected recommendations are reported by a tool, the recall is 100%.

F-score (F1) measures the accuracy of a tool's recommendation:

$$F1 = \frac{2 \times Pre \times Rec}{Pre + Rec} \times 100\% \tag{6.4}$$

F-score is the harmonic mean of precision and recall. Its value varies within [0%, 100%]. The higher F-scores are desirable, as they demonstrate better trade-offs between the precision and recall rates.

Weighted Average (WA) measures a tool's **overall effectiveness** among all experimented data in terms of coverage, precision, recall, and F-score:

$$\Gamma_{overall} = \frac{\sum_{i=1}^{8} \Gamma_i * n_i}{\sum_{i=1}^{8} n_i}. \tag{6.5}$$

In the formula, i varies from 1 to 8, representing the 8 projects used in our evaluation (Storybook and Electron were excluded). Here, $i = 1$ corresponds to Node.js and $i = 8$ corresponds to Webpack; n_i represents the number of tasks built from the i^{th} project. Γ_i represents any measurement value of the i^{th} project for coverage, precision, recall, or F-score. By combining such measurement values of eight projects in a weighted way, we were able to assess a tool's overall effectiveness $\Gamma_{overall}$.

6.3 Effectiveness Comparison with ROSE

In our evaluation, we compared CoRec with a popularly used co-change suggestion tool ROSE [73]. ROSE mines the association rules between co-changed entities from projects' software version history. An exemplar mined association rule is shown below:

Table 6.2: Comparison results between CoRec and ROSE for $^*\mathrm{CF}\xrightarrow{f}\mathrm{CF}$ tasks

Project	CoRec				ROSE			
	Cov (%)	Pre (%)	Rec (%)	F1 (%)	Cov (%)	Pre (%)	Rec (%)	F1 (%)
Node.js	77	68	69	69	61	24	56	34
Meteor	88	72	70	71	46	16	43	24
Ghost	73	67	74	71	50	20	53	29
Habitica	80	80	78	79	40	7	37	12
PDF.js	71	77	81	79	29	27	41	33
React	91	86	76	81	32	59	70	64
Serverless	84	77	79	78	64	20	75	32
Webpack	89	71	81	75	50	7	29	12
WA	**83**	**72**	**73**	**73**	**53**	**21**	**52**	**29**

$$\begin{array}{l}\{(\_Qdmodule.c, func, GrafObj\_getattr())\} \Rightarrow \\ \{\ (qdsupport.py, func, outputGetattrHook()).\ \}\end{array} \tag{6.6}$$

This rule means that whenever the function `GrafObj_getattr()` in a file _Qdmodule.c is changed, the function `outputGetattrHook()` in another file qdsupport.py should also be changed. Based on such rules, given a program commit, ROSE tentatively matches all edited entities with the antecedents of all mined rules and recommends co-changes if any tentative match succeeds. In our comparative experiment, we applied ROSE to the constructed recommendation tasks and version history of each subject project. We configured ROSE with $support = 1$ and $confidence = 0.1$, because the ROSE paper [73] mentioned this setting multiple times.

As shown in Table 6.2, CoRec outperformed ROSE in terms of all measurements. Take Webpack as an example. Among the 138 $^*\mathrm{CF}\xrightarrow{f}\mathrm{CF}$ prediction tasks in this project, CoRec provided change recommendations for 89% of tasks; with these recommendations, CoRec achieved 71% precision, 81% recall, and 75% accuracy. On the other hand, ROSE recommended changes for only 50% of tasks; based on its recommendations, ROSE acquired only 7% precision, 29% recall, and 12% accuracy. Among the eight subject projects, the weighted average measurements of CoRec include 83% coverage, 72% precision, 73% recall, and 73% accuracy. Meanwhile, the weighted average measurements of ROSE include 53% coverage, 21% precision, 52% recall, and 29% accuracy. Such measurement contrasts indicate that CoRec usually recommended more changes than ROSE, and CoRec's recommendations were more accurate.

In addition to $^*\mathrm{CF}\xrightarrow{f}\mathrm{CF}$ tasks, we also compared CoRec with ROSE for $^*\mathrm{CF}\xrightarrow{f}\mathrm{AF}$ and $^*\mathrm{CF}\xrightarrow{v}\mathrm{AV}$ tasks, as shown in Tables 6.3 and 6.4. Similar to what we observed in Table 6.2, CoRec outperformed ROSE in terms of all metrics for both types of tasks. With more details, given $^*\mathrm{CF}\xrightarrow{f}\mathrm{AF}$ tasks, on average, CoRec achieved 81% coverage, 76% precision, 80% recall, and 78% accuracy; while ROSE acquired 54% coverage, 21% precision, 48% recall, and 28% accuracy (see Table 6.3). Given $^*\mathrm{CF}\xrightarrow{v}\mathrm{AV}$ tasks (see Table 6.4), CoRec obtained 79% coverage, 76% precision, 81% recall, and 78% accuracy; while ROSE obtained only 45% coverage, 17% precision, 54% recall, and 25% accuracy. In particular, as shown in

Table 6.3: Comparison results between CoRec and ROSE for $*CF \xrightarrow{f} AF$ tasks

Project	**CoRec**				**ROSE**			
	Cov (%)	**Pre** (%)	**Rec** (%)	**F1** (%)	**Cov** (%)	**Pre** (%)	**Rec** (%)	**F1** (%)
Node.js	79	69	74	72	59	20	52	29
Meteor	86	77	82	80	40	22	44	29
Ghost	85	86	85	85	46	18	46	26
Habitica	87	77	85	81	56	4	23	7
PDF.js	65	87	88	87	22	9	28	14
React	71	84	82	83	16	66	7	13
Serverless	84	71	85	77	73	19	59	29
Webpack	75	79	85	82	53	16	46	24
WA	**81**	**76**	**80**	**78**	**54**	**21**	**48**	**28**

Table 6.4: Comparison results between CoRec and ROSE for $*CF \xrightarrow{v} AV$ tasks

Project	**CoRec**				**ROSE**			
	Cov (%)	**Pre** (%)	**Rec** (%)	**F1** (%)	**Cov** (%)	**Pre** (%)	**Rec** (%)	**F1** (%)
Node.js	79	72	77	74	55	20	65	31
Meteor	72	77	84	81	26	2	14	4
Ghost	84	75	81	78	46	18	46	26
Habitica	89	82	85	83	27	20	45	28
PDF.js	78	87	84	85	20	4	28	8
React	89	73	78	76	36	8	33	13
Serverless	70	80	85	82	34	0	0	-
Webpack	87	86	83	85	36	8	33	13
WA	**79**	**76**	**81**	**78**	**45**	**17**	**54**	**25**

Table 6.4, for Serverless, CoRec achieved 70% coverage, 80% precision, 85% recall, and 82% accuracy. Meanwhile, ROSE only provided recommendations for 34% of the tasks, and none of these recommendations is correct.

By comparing the evaluation results reported in Tables 6.2–6.4, we found that the effectiveness of CoRec and ROSE seem to be stabilized across different types of prediction tasks. Specifically, among the three kinds of tasks, on average, CoRec achieved 79–83% coverage, 72–76% precision, 73–81% recall, and 73–78% accuracy. On the other hand, ROSE achieved 45–54% coverage, 17–21% precision, 48–54% recall, and 25–29% accuracy. The consistent comparison results imply that CoRec usually recommended co-changed functions for more tasks than ROSE, and CoRec's recommendations usually had higher quality.

Two major reasons can explain why CoRec worked better. First, ROSE purely uses the co-changed entities in version histories to recommend changes. When the history data is incomplete or some entities were never co-changed before, ROSE may lack evidence to predict co-changes, and thus obtains lower coverage and recall rates. In comparison, CoRec extracts nine features from a given commit and one feature from the version history; even though history data provides insufficient indication on the potential co-change relationship between entities, the other extracted features serve as additional indicators. Second, ROSE observes no syntactic or semantic relationship between co-changed entities; thus, it can infer incorrect rules from co-changed but unrelated entities, and achieves lower precision. In comparison, CoRec observes the syntactic relationship between co-changed entities by tracing the referencer-referencee relations; it also observes the semantic relationship by extracting features to reflect the commonality (1) between co-changed functions (*CF), and (2) between any changed function cf and the changed entity E on which cf depends (E is CF in P1, AF in P2, and AV in P3).

Although CoRec outperformed ROSE in our experiments, we consider CoRec as a complementary tool to ROSE. This is because CoRec bases its change recommendations on the three most popular RCPs we found. If some changes do not match any of the RCPs, CoRec does not recommend any change but ROSE may suggest some edits.

> **Finding 1:** *CoRec outperformed ROSE when predicting co-changed functions based on the three recurring change patterns (P1–P3). CoRec serves as a good complementary tool to ROSE.*

6.4 Comparison with A Variant Approach

Readers may be tempted to train a unified classifier instead of three separate classifiers, because the three classifiers all take in the same format of inputs and output the same types of predictions (i.e., whether to co-change or not). However, as shown in Table 4.4, the commonality characteristics between co-changed functions vary with RCPs. For instance, the co-changed functions in P2 usually commonly invoke peer functions (i.e., FI), the co-changed functions in P3 often commonly read/write peer variables (i.e., VA), while the co-changed functions in P1 have weaker commonality signals for both FI and ST (i.e., common token subsequences). If we mix the co-changed functions matching different patterns to train a single classifier, it is quite likely that the extracted features between co-changed functions become less informative, and the trained classifier has poorer prediction power.

To validate our approach design, we actually also built a variant approach of CoRec—CoRec$_u$—that trains a unified classifier with the program commits matching either of the three RCPs (P1–P3) and predicts co-change functions with the single classifier. To evaluate CoRec$_u$, we clustered the data portions matching distinct RCPs for each project, and conducted five-fold cross validation. As shown in Table 6.5, on average, CoRec$_u$ recommended changes with 70% coverage, 56% precision, 61% recall, and 59% accuracy. These measured values are much lower than the weighted averages of CoRec reported in Table 6.2–Table 6.4. The empirical comparison corroborates our hypothesis that when data matching distinct RCPs are mixed to train a unified classifier, the classifier works less effectively.

Table 6.5: The effectiveness of CoRec_u when it trains and tests a unified classifier

Project	Cov (%)	Pre (%)	Rec (%)	F1 (%)
Node.js	72	50	57	53
Meteor	77	59	58	59
Ghost	53	61	70	65
Habitica	55	53	68	60
PDF.js	29	60	73	66
React	76	75	73	74
Serverless	54	47	61	53
Webpack	66	54	63	58
WA	**70**	**56**	**61**	**59**

Finding 2: *CoRec_u worked less effectively than CoRec by training a unified classifier with data matching distinct RCPs. This experiment validates our approach design of training three separate classifiers.*

6.5 Sensitivity to The Adopted ML Algorithm

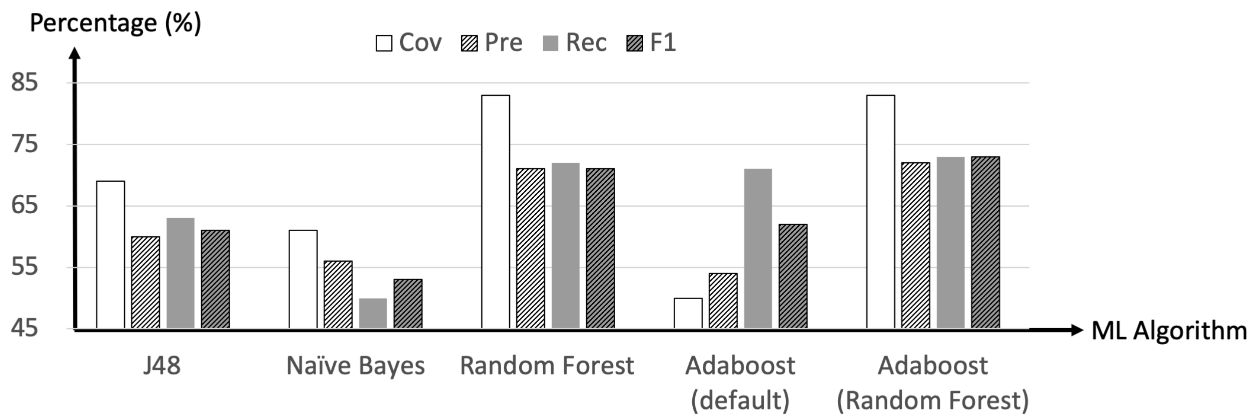
To understand how sensitive CoRec is to the selection of ML algorithms, in addition to Adaboost (Random Forest), we also explored four other algorithms: J48 [57], Random Forest [45], Naïve Bayes [44], and Adaboost (default).

- **J48** builds a decision tree as a predictive model to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves).

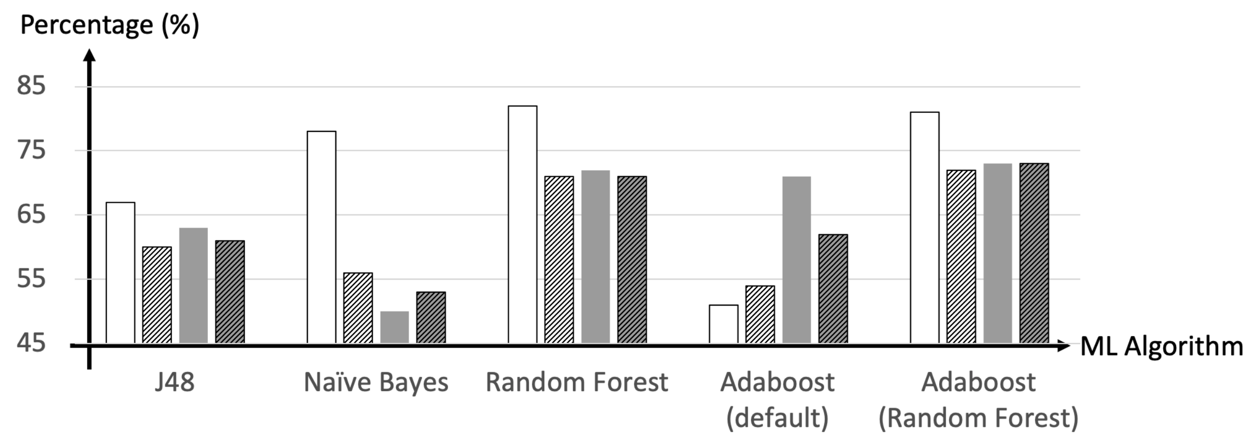
- **Naïve Bayes** calculates the probabilities of hypotheses by applying Bayes' theorem with strong (naïve) independence assumptions between features.

- **Random Forest** is an ensemble learning method that trains a model to make predictions based on a number of different models. Random Forest trains a set of individual models in a parallel way. Each model is trained with a random subset of the data. Given a candidate in the testing set, individual models make their separate predictions and Random Forest uses the one with the majority vote as its final prediction.

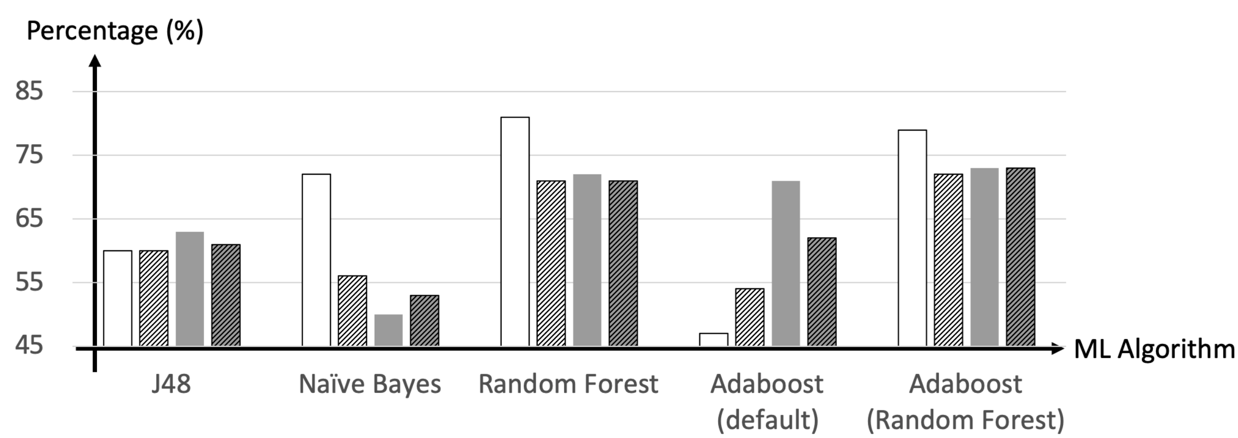
- **Adaboost** is also an ensemble learning method. However, different from Random Forest, Adaboost trains a bunch of individual models (i.e., weak learners) in a sequential way. Each individual model learns from mistakes made by the previous model. We tried two variants of Adaboost: (1) Adaboost (default) with decision trees as the weak learners, and (2) Adaboost (Random Forest) with Random Forests as the weak learners.



(a) The *CF$\xrightarrow{f}$CF data



(b) The *CF$\xrightarrow{f}$AF data



(c) The *CF$\xrightarrow{v}$AV data

Figure 6.2: Comparison between different ML algorithms on different data sets

Figure 6.2 illustrates the effectiveness comparison when CoRec adopts different ML algorithms. The three subfigures (Figure 6.2 (a)–(c)) separately present the comparison results on the data sets of *CF$\xrightarrow{f}$CF, *CF$\xrightarrow{f}$AF, and *CF$\xrightarrow{v}$AV. We observed similar phenomena in all subfigures. First, Adaboost (Random Forest) and Random Forest achieved very similar effectiveness, and both of them considerably outperformed the other algorithms. Second, compared with Random Forest, Adaboost (Random Forest) obtained better precision, better recall, better accuracy, but equal or slightly lower coverage; thus, we chose Adaboost (Random Forest) as the default ML algorithm used in CoRec. Third, among all data sets, Naïve Bayes obtained the lowest recall and accuracy rates; while Adaboost (default) obtained the lowest coverage rates.

Our results imply that although ensemble learning methods generally outperform other ML algorithms, their effectiveness also depends on (1) what weak learners are used and (2) how we organize weak learners. Between Adaboost (Random Forest) and Adaboost (default), the only difference exists in the used weak learner (Random Forest vs. Decision Tree). Our evaluation shows that Random Forest helps improve Adaboost's performance when it is used as the weak learner. Additionally, between Random Forest and Adaboost (default), the only difference is how they combine decision trees as weak learners. Our evaluation shows that Random Forest outperforms Adaboost by training weak learners in a parallel instead of sequential way.

Finding 3: *CoRec is sensitive to the adopted ML algorithm. CoRec obtained the lowest prediction accuracy when Naïve Bayes was used, but acquired the highest accuracy when Adaboost (Random Forest) was used.*

Chapter 7

Threats to validity

All our observations and experiment results are limited to the software repositories we used. These observations and results may not generalize well to other JS projects, especially to the projects that use the Asynchronous Module Definition (AMD) APIs to define code modules and their dependencies. In the future, we would like to include more diverse projects into our data sets so that our findings are more representative.

Given a project P, CoRec adopts commits in P's software version history to train classifiers that can recommend co-changes for new program commits. When the version history has few commits to train classifiers, the applicability of CoRec is limited. CoRec shares such limitation with existing tools that provide project-specific change suggestions based on software version history [39, 60, 73]. To further lower CoRec's requirement to available commits in software version history, we plan to investigate more ways to extract features from commits and better capture the characteristics of co-changed functions.

When creating recommendation tasks for classifier evaluation, we always assumed that the experimented commits contain accurate information of all co-changed functions. It is possible that developers made mistakes when applying multi-entity edits. Therefore, the imperfect evaluation data set based on developers' edits may influence our empirical comparison between CoRec and ROSE. We share this limitation with prior work [39, 41, 47, 60, 67, 68, 73]. In the future, we plan to mitigate the problem by conducting user studies with developers. By carefully examining the edits made by developers and the co-changed functions recommended by tools, we can better assess the effectiveness of different tools.

Chapter 8

Related work

The related work includes empirical studies on JS code and related program changes, change recommendation systems, and co-change mining.

8.1 Empirical Studies on JS Code and Related Program Changes

Various studies were conducted to investigate JS code and related changes [36, 38, 49, 64]. For instance, Ocariza et al. conducted an empirical study of 317 bug reports from 12 bug repositories, to understand the root cause and consequence of each reported bug [49]. They observed that 65% of JS bugs were caused by the faulty interactions between JS code and Document Object Models (DOMs). Gao et al. empirically investigated the benefits of leveraging static type systems (e.g., Facebook's Flow [12] and Microsoft's TypeScript [24]) to check JS programs [36]. To do that, they manually added type annotations to buggy code and tested whether Flow and TypeScript reported an error on the buggy code. They observed that both Flow 0.30 and TypeScript 2.0 detected 15% of errors, showing great potential of finding bugs. Selakovic and Pradel studied 98 fixed performance issues from 16 popular client-side and server-side JS projects [64]. They identified eight root causes of performance issues and showed that inefficient API usage is the most prevalent root cause; they also found that most issues were addressed by optimizations that modify only a few lines of code.

Our research is different from all prior studies. Instead of manually inspecting individual program changes, we adopted static program analysis to automatically extract the frequent co-change patterns between entities in a large-scale way, and developed an ML-based approach to recommend co-changes based on the observed most frequent patterns.

8.2 JS Bug Detectors

Researchers built tools to automatically detect bugs or malicious JS code [29, 31, 52, 54, 55, 56, 58, 62]. For example, EventRacer detects harmful data races in event-driven programs [58]. JSNose combines static and dynamic analysis to detect 13 JS smells in client-side code, where smells are code patterns that can adversely influence program comprehension and software maintenance [31]. TypeDevil adopts dynamic analysis to warn developers about variables, properties, and functions that have inconsistent types [55]. DeepBugs is a learning-based approach that formulates bug detection as a binary classification problem; it is able to detect accidentally swapped function arguments, incorrect binary operators, and incorrect operands in binary operations [56]. EarlyBird conducts dynamic analysis and adopts machine learning techniques for early identification of malicious behaviors of JavaScript code [62].

Some other researchers developed tools to suggest bug fixes or code refactorings [27, 32, 40, 46, 48, 50, 63]. With more details, Vejovis suggests program repairs for DOM-related JS bugs based on two common fix patterns: parameter replacements and DOM element validations [50]. Monperrus and Maia built a JS debugger to help resolve "crowd bugs" (i.e., unexpected and incorrect outputs or behaviors resulting from the common and intuitive usage of APIs) [48]. Given a crowd bug, the debugger sends a code query to a server and retrieves all StackOverflow answers potentially related to the bug fix. An and Tilevich built a JS refactoring tool to facilitate JS debugging and program repair [27]. Given a distributed JS application, the tool first converts the program to a semantically equivalent centralized version by gluing together the client and server parts. After developers fixed bugs in the centralized version, the tool generates fixes for the original distributed version accordingly. In Model-Driven Engineering, ReVision repairs incorrectly updated models by (1) extracting change patterns from version history, and (2) matching incorrect updates against those patterns to suggest repair operations [53].

Our methodology is most relevant to the approach design of ReVision. However, our research is different in three aspects. First, our research focuses on entity-level co-change patterns in JS programs, while ReVision examines consistency rules between different UML artifacts (e.g., the signature of a message in a sequence diagram must correspond to a method signature in the related class diagram). Second, the co-changed recommendation by CoRec intends to complete an applied multi-entity edit, while the repair operations proposed by ReVision tries to complete consistency-preserving edit operations. Third, we conducted a large-scale empirical study to characterize multi-entity edits and experimented CoRec with eight open-source projects, while ReVision was not empirically evaluated.

8.3 Co-Change Recommendation Systems

Some approaches were introduced to mine software version history and to extract co-change patterns [34, 35, 37, 39, 42, 43, 60, 61, 65, 66, 71, 72, 73]. Specifically, Gall et al. mined product release history to identify the co-change relationship between modules [34] and classes [35]. Some researchers developed tools (e.g., ROSE) to mine the association rules between co-changed entities and to suggest possible changes accordingly [39, 42, 60, 61, 66, 71, 73]. Some other researchers built hybrid approaches by combining information retrieval (IR) with association rule mining [37, 43, 72]. Specifically given a software entity E, these approaches use IR techniques to (1) extract terms from E and any other entity and (2) rank those entities based on their term-usage overlap with E. Meanwhile, these tools also apply association rule mining to commit history in order to rank entities based on the co-change frequency. In this way, if an entity G has significant term-usage overlap with E and has been co-changed a lot with E, then G is recommended to be changed together with E.

Shirabad et al. developed a learning-based approach that predicts whether two given files should be changed together or not [65]. In particular, the researchers extracted features from software repository to represent the relationship between each pair of files, adopted those features of file pairs to train an ML model, and leveraged the model to predict whether any two files are relevant (i.e., should be co-changed) or not. CoRec is closely related to the learning-based approach by Shirabad et al. [65]. However, it is different in two aspects. First, CoRec predicts co-changed functions instead of co-changed files. With finer-granularity recommendations, CoRec can help developers to better validate suggested changes and to edit code more easily. Second, our feature engineering for CoRec is based on the quantitative analysis of frequent change patterns and qualitative analysis of the commonality between co-changed functions, while the feature engineering by Shirabad is mainly based on their intuitions. Consequently, most of our features are about the code commonality or co-evolution relationship between functions; while the features defined by Shirabad et al. mainly focus on file names/paths, routines referenced by each file, and the code comments together with problem reports related to each file.

Wang et al. recently conducted a similar study on multi-entity edits to Java code [69], and developed CMSuggester—an automatic approach to suggest method changes accordingly [41, 68]. Although our characterization study focuses on JS instead of Java code repositories, it corroborates the observations made by Wang et al.. For instance, our study also showed that about half of bug fixes involved multi-entity edits; we found three RCPs to commonly exist in all studied projects. Different from CMSuggester, CoRec is an ML-based instead of rule-based approach; thus, CoRec requires for training data to prepare an ML model before suggesting changes while CMSuggester requires tool builders to hardcode the suggestion strategies. CoRec recommends changes based on three RCPs: *CF$\xrightarrow{f}$CF, *CF$\xrightarrow{f}$AF, and *CF$\xrightarrow{v}$AV; while CMSuggester recommends changes based on the last two patterns. Overall, CoRec is more flexible due to its usage of ML and is applicable to more types of co-change scenarios.

Chapter 9

Conclusion

It is usually tedious and error-prone to develop and maintain JS code. To facilitate program comprehension and software debugging, we conducted an empirical study on multi-entity edits in JS projects and built an ML-based co-change recommendation tool CoRec.

Our empirical study explored the frequency and composition of multi-entity edits in JS programs; we also investigated the syntactic and semantic relevance between frequently co-changed entities. In particular, we observed that (i) JS software developers frequently apply multi-entity edits while the co-changed entities are usually syntactically related; (ii) there are three most popular RCPs that commonly exist in all studied JS code repositories: $*\text{CF}\xrightarrow{f}\text{CF}$, $*\text{CF}\xrightarrow{f}\text{AF}$, and $*\text{CF}\xrightarrow{v}\text{AV}$; and (iii) among the entities matching the three RCPs, co-changed functions usually share certain commonality with each other (e.g., common function invocations and common token subsequences).

Based on our study, we developed CoRec, which tool extracts code features from the multi-entity edits that match any of the three RCPs, and trains an ML model with the extracted features to specially characterize relationship between co-changed functions. Given a new program commit or a set of entity changes that developers apply, the trained model extracts features from the program revision and recommends changes to complement applied edits as necessary. Our evaluation shows that CoRec recommended changes with high accuracy and outperformed a baseline technique. In the future, we will investigate novel approaches to provide finer-grained code change suggestions and automate test case generation for suggested changes.

Bibliography

[1] 2016. Fs: make callback mandatory to all async functions. https://github.com/nodejs/node/commit/21b0a27.

[2] 2017. Http2: introducing HTTP/2. https://github.com/nodejs/node/commit/e71e71b5138c3dfee080f4215dd957dc7a6cbdaf.

[3] 2017. What Is ES6 and What Javascript Programmers Need to Know. https://www.makeuseof.com/tag/es6-javascript-programmers-need-know/.

[4] 2019. fs: require callback in read. https://github.com/nodejs/node/commit/8e1b6e7.

[5] 2019. The 10 most popular programming languages, according to the Microsoft-owned GitHub. https://www.businessinsider.com/most-popular-programming-languages-github-2019-11.

[6] 2020. Babel. https://babeljs.io.

[7] 2020. Closure Compiler. https://developers.google.com/closure/compiler.

[8] 2020. Electron. https://github.com/electron/electron.

[9] 2020. ESprima. https://esprima.org/

[10] 2020. Facebook React. https://github.com/facebook/react.

[11] 2020. Fix some fibers vs SQLite issues. https://github.com/meteor/meteor/commit/e9a88b00b9cdd35eb281c7113fcaa5155f006ea3.

[12] 2020. Flow: A Static Type Checker for JavaScript. https://flow.org.

[13] 2020. Ghost. https://github.com/TryGhost/Ghost.

[14] 2020. HabitRPG Habitica. https://github.com/HabitRPG/habitica.

[15] 2020. JSHint, a JavaScript Code Quality Tool. https://jshint.com.

[16] 2020. JSLint: The JavaScript Code Quality Tool. https://jslint.com.

[17] 2020. Meteor. https://github.com/meteor/meteor.

[18] 2020. Mozilla Pdf. https://github.com/mozilla/pdf.js/.

[19] 2020. Nodejs Node. https://github.com/nodejs/node.

[20] 2020. Serverless. https://github.com/serverless/serverless.

[21] 2020. Stack Overflow - Where Developers Learn, Share, & Build Careers. https://stackoverflow.com.

[22] 2020. Storybook. https://github.com/storybookjs/storybook.

[23] 2020. typed-ast-util. https://github.com/returntocorp/typed-ast-util.

[24] 2020. TypeScript - JavaScript that scales. https://www.typescriptlang.org.

[25] 2020. Webpack. https://github.com/webpack/webpack.

[26] 2020. WebStorm: The Smartest JavaScript IDE. https://www.jetbrains.com/webstorm/.

[27] Kijin An and Eli Tilevich. 2019. Catch & Release: An Approach to Debugging Distributed Full-Stack JavaScript Applications. In *Web Engineering*, Maxim Bakaev, Flavius Frasincar, and In-Young Ko (Eds.). Springer International Publishing, Cham, 459–473.

[28] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 10 (Oct 2004), 1367–1372. https://doi.org/10.1109/TPAMI.2004.75

[29] Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2010. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. Association for Computing Machinery, New York, NY, USA, 281–290. https://doi.org/10.1145/1772690.1772720

[30] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[31] A. M. Fard and A. Mesbah. 2013. JSNOSE: Detecting JavaScript Code Smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 116–125.

[32] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. 2011. Tool-Supported Refactoring for JavaScript. *SIGPLAN Not.* 46, 10 (Oct. 2011), 119–138. https://doi.org/10.1145/2076021.2048078

[33] Yoav Freund and Robert E. Schapire. 1996. Experiments with a New Boosting Algorithm. In *Proceedings of the Thirteenth International Conference on International Conference on Machine Learning (ICML'96)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 148–156.

[34] H. Gall, K. Hajek, and M. Jazayeri. 1998. Detection of logical coupling based on product release history. In *Proc. ICSM*. 190–198.

[35] H. Gall, M. Jazayeri, and J. Krajewski. 2003. CVS release history data for detecting logical couplings. In *Proc. IWPSE*. 13–23.

[36] Z. Gao, C. Bird, and E. T. Barr. 2017. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 758–769. https://doi.org/10.1109/ICSE.2017.75

[37] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. 2012. Integrated impact analysis for managing software changes. In *Proc. ICSE*. 430–440.

[38] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, and A. Mesbah. 2019. BugsJS: a Benchmark of JavaScript Bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 90–101. https://doi.org/10.1109/ICST.2019.00019

[39] Md. Anaytul Islam, Md. Moksedul Islam, Manishankar Mondal, Banani Roy, Chanchal Kumar Roy, and Kevin A. Schneider. 2018. [Research Paper] Detecting Evolutionary Coupling Using Transitive Association Rules. In *Proc. SCAM*. 113–122.

[40] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the Eval That Men Do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 34–44. https://doi.org/10.1145/2338965.2336758

[41] Zijian Jiang, Ye Wang, Hao Zhong, and Na Meng. 2020. Automatic method change suggestion to complement multi-entity edits. *Journal of Systems and Software* 159 (2020). http://login.ezproxy.lib.vt.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-85073505497&site=eds-live&scope=site

[42] H. Kagdi, J. I. Maletic, and B. Sharif. 2007. Mining software repositories for traceability links. In *Proc. ICPC*. 145–154.

[43] Huzefa H. Kagdi, Malcom Gethers, and Denys Poshyvanyk. 2012. Integrating conceptual and logical couplings for change impact analysis in software. *Empirical Software Engineering* 18 (2012), 933–969.

[44] David D. Lewis. 1998. Naive (Bayes) at forty: The independence assumption in information retrieval. In *Machine Learning: ECML-98*, Claire Nédellec and Céline Rouveirol (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 4–15.

[45] Andy Liaw and Matthew Wiener. 2002. Classification and Regression by randomForest. *R News* 2, 3 (2002), 18–22. https://CRAN.R-project.org/doc/Rnews/

[46] Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. 2012. Eval Begone! Semi-Automated Removal of Eval from Javascript Programs. *SIGPLAN Not.* 47, 10 (Oct. 2012), 607–620. https://doi.org/10.1145/2398857.2384660

[47] Na Meng, Miryung Kim, and Kathryn McKinley. 2013. LASE: Locating and Applying Systematic Edits. In *Proc. ICSE*. 502–511.

[48] Martin Monperrus and Anthony Maia. 2014. *Debugging with the Crowd: a Debug Recommendation System based on Stackoverflow.* Research Report hal-00987395. Université Lille 1 - Sciences et Technologies. https://hal.archives-ouvertes.fr/hal-00987395

[49] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. 2013. An Empirical Study of Client-Side JavaScript Bugs. *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on, Empirical Software Engineering and Measurement (ESEM), 2012 ACM-IEEE International Symposium on* (2013), 55 – 64. http://login.ezproxy.lib.vt.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.6681338&site=eds-live&scope=site

[50] F.S. Ocariza, K. Pattabiraman, and A. Mesbah. 2014. Vejovis: Suggesting fixes for JavaScript faults. In *Proceedings - International Conference on Software Engineering*. Electrical and Computer Engineering, University of British Columbia, 837–847. http://login.ezproxy.lib.vt.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-84993660437&site=eds-live&scope=site

[51] Frolin S. Ocariza. 2016. On the detection, localization and repair of client-side JavaScript faults.

[52] F. S. Ocariza Jr., K. Pattabiraman, and A. Mesbah. 2012. AutoFLox: An Automatic Fault Localizer for Client-Side JavaScript. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 31–40.

[53] M. Ohrndorf, C. Pietsch, and T. Kehrer. 2018. ReVision: A Tool for History-Based Model Repair Recommendations. In *Proc. ICSE-Companion*. 105.

[54] Jihyeok Park. 2014. JavaScript API Misuse Detection by Using Typescript. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity (MODULARITY '14)*. Association for Computing Machinery, New York, NY, USA, 11–12. https://doi.org/10.1145/2584469.2584472

[55] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015), 314. http://login.ezproxy.lib.vt.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edb&AN=110064267&site=eds-live&scope=site

[56] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 147 (Oct. 2018), 25 pages. https://doi.org/10.1145/3276517

[57] J. Ross Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[58] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-Driven Programs. *ACM SIGPLAN NOTICES* 48, 10 (2013), 151 – 166. http://login.ezproxy.lib.vt.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edswsc&AN=000327697300008&site=eds-live&scope=site

[59] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. ACM, New York, NY, USA, 432–448. https://doi.org/10.1145/1028976.1029012

[60] T. Rolfsnes, S. D. Alesio, R. Behjati, L. Moonen, and D. W. Binkley. 2016. Generalizing the Analysis of Evolutionary Coupling for Software Change Impact Analysis. In *Proc. SANER*. 201–212.

[61] Thomas Rolfsnes, Leon Moonen, Stefano Di Alesio, Razieh Behjati, and Dave Binkley. 2018. Aggregating Association Rules to Improve Change Recommendation. *Empirical Software Engineering* 23, 2 (2018), 987–1035.

[62] Kristof Schütt, Marius Kloft, Alexander Bikadorov, and Konrad Rieck. 2012. Early Detection of Malicious Behavior in JavaScript Code. In *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence (AISec '12)*. Association for Computing Machinery, New York, NY, USA, 15–24. https://doi.org/10.1145/2381896.2381901

[63] M. Selakovic and M. Pradel. 2015. Poster: Automatically Fixing Real-World JavaScript Performance Bugs. *2015 ICSE International Conference on Software Engineering.* (2015), 811. http://search.ebscohost.com/login.aspx?direct=true&db=edb&AN=111750044&site=eds-live&scope=site.

[64] M. Selakovic and M. Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 61–72. https://doi.org/10.1145/2884781.2884829

[65] Jelber Sayyad Shirabad, T. C. Lethbridge, and S. Matwin. 2003. Mining the maintenance history of a legacy software system. In *Proc. ICSM*. 95–104.

[66] Luciana Lourdes Silva, Marco Tulio Valente, and Marcelo de Almeida Maia. 2015. Co-change Clusters: Extraction and Application on Assessing Software Modularity. *Trans. Aspect-Oriented Software Development* 12 (2015), 96–131.

[67] Tan, Ming. 2015. *Online Defect Prediction for Imbalanced Data*. Master's thesis. University of Waterloo.

[68] Ye Wang, Na Meng, and Hao Zhong. 2018. CMSuggester: Method Change Suggestion to Complement Multi-entity Edits. In *Proc. SATE*. 137–153.

[69] Ye Wang, Na Meng, and Hao Zhong. 2018. An Empirical Study of Multi-Entity Changes in Real Bug Fixes. In *Proc. ICSME*. 287–298.

[70] Xiao Xiao, Shi Han, Charles Zhang, and Dongmei Zhang. 2015. Uncovering JavaScript Performance Code Smells Relevant to Type Mutations. In *Programming Languages and Systems*, Xinyu Feng and Sungwoo Park (Eds.). Springer International Publishing, Cham, 335–355.

[71] Annie T. T. Ying, Gail C. Murphy, Raymond T. Ng, and Mark Chu-Carroll. 2004. Predicting Source Code Changes by Mining Change History. *IEEE Trans. Software Eng.* 30, 9 (2004), 574–586.

[72] Motahareh Bahrami Zanjani, George Swartzendruber, and Huzefa Kagdi. 2014. Impact Analysis of Change Requests on Source Code Based on Interaction and Commit Histories. In *Proc. MSR*. 162–171.

[73] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. 2004. Mining Version Histories to Guide Software Changes. In *Proc. ICSE*. 563–572.