Classifying Code Commits with Convolutional Neural Networks

Na Meng
Department of Computer Science
Virginia Tech
Blacksburg, United States
nm8247@vt.edu

Zijian Jiang
Department of Computer Science
Virginia Tech
Blacksburg, United States
wz649588@vt.edu

Hao Zhong
Department of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai, China
zhonghao@sjtu.edu.cn

***Abstract*—Developers change software programs for various purposes (e.g., bug fixes, feature additions, and code refactorings), but the intents of code changes are often not recorded or are poorly documented. To automatically infer the change intent of each program commit (i.e., a set of code changes), existing work classifies commits based on commit messages and/or the sheer counts of edited files, lines, or abstract syntax tree (AST) nodes. However, none of these tools reason about the syntactic or semantic dependencies between co-applied changes, neither do they adopt any deep learning method. To better characterize program commits, in this paper, we present CClassifier—a new approach that classifies commits by (1) using advanced static program analysis to comprehend relationship between co-applied edits, (2) representing edits and their relationship via graphs, and (3) applying convolutional neural networks (CNN) to classify those graphs.**

Compared with prior work, CClassifier extracts a richer set of features from program changes; it is the first to classify program commits using CNN. For evaluation, we prepared a benchmark that contains 7,414 code changes from 5 open-source Java projects. On this benchmark, we empirically compared CClassifier and the state-of-the-art approach with five-fold cross validation. On average, when predicting bug-fixing commits within the same projects, CClassifier improved the prediction accuracy from 70% to 72%. More importantly, prior work seldom identifies feature-addition commits; CClassifier can successfully identify such commits in a lot more scenarios. Our evaluation shows that CClassifier outperforms prior work due to its usage of advanced program analysis and CNN.

***Index Terms*—Program commit, classification, deep learning**

I. Introduction

During software maintenance, developers commit a variety of code changes to fix bugs, add new features, or improve code implementation. Understanding the change intents of these program commits can facilitate developers to review other developers' code changes, and to prioritize and better process pull requests; it can also enable researchers to effectively study program data. However, manually reading code and inferring the change intents can be challenging and time-consuming. Although some projects (e.g., Apache Mahout [1]) adopt issue tracking systems (e.g., Jira [2]) to explictly describe change intents, they do not record such information for all commits. Furthermore, many popular projects (e.g., Linux) even do not use issue trackers. Even though developers sometimes describe change intents in commit messages, there are still many commits whose low-quality commit messages mention nothing about the change intents.

To automatically infer change intents from commits, researchers built machine learning-based approaches [3]–[6]. For example, Hindle et al. [3] extracted coarse-grained features from changes such as authors and file types, and Tian et al. [4] extracted fine-grained features like the number of added loops; they all trained models with the extracted features using traditional algorithms (e.g., decision tree and SVM). Jiang et al. [5] and Loyola et al. [6] extracted added or deleted tokens from code diff files to translate code changes into commit messages via neural machine translation. However, these tools have two limitations. First, they use commit messages to train classifiers; when commit messages are missing or poorly specified, the trained classifiers can work badly. Second, these tools treat co-applied edits as independent ones although in reality, these edits are usually related [7]. When overlooking the interconnections between co-applied edits, existing tools can misclassify commits.

Intuitively, to fulfill distinct maintenance tasks, developers usually apply different sets of changes and those changes are connected in specific ways. For instance, to add new functionalities, developers often define new entities (e.g., fields and methods) and revise existing entities to access the new ones; for bug fixing, developers often change existing entities to update the program logic; and for refactoring, developers may apply semantic-preserving changes to existing code (e.g., renaming) or even define new entities as needed. These insights inspired us to develop CClassifier—an automatic approach that classifies commits based on co-applied edits and any relationship between them. Our research overcomes three technical challenges:

1) **How should we characterize program commits?** To characterize each program commit and to relate co-applied edits with each other as much as possible, CClassifier conducts static program analysis to recognize (1) the syntactic dependencies (i.e., referencer-referencee relations) between edited entities, (2) the control and data dependencies between edited statements, and (3) refactored entities (e.g., renamed methods).
2) **How can we represent characterizations uniformly?** As some characteristics involve single entities while

others involve multiple entities, novel representations are needed to uniformly encode distinct characteristics into formats processable by machine learning. CClassifier creates a graph for each commit by treating changed entities as nodes and representing the syntactic dependencies between entities as edges; it encodes the other characteristics as attributes of nodes or edges.

3) **How can we classify program commits based on code changes and their relationship?**
Since traditional machine learning algorithms are not good at classifying graphs, CClassifier uses convolutional neural networks (CNN) to classify graphs. In particular, CClassifier identifies a set of important nodes in each graph, normalizes the vector representations for graphs [8], and feeds those vectors to CNN.

For evaluation, we constructed a data set with the 7,414 program commits from 5 Apache projects, whose commit categories were manually labeled by developers. To compare with prior work, we reimplemented the state-of-the-art approach by Levin et al. [9] as our baseline based on the paper description. This baseline classifies commits based on commit messages and code changes. CClassifier outperformed the baseline by classifying changes more accurately, although it did not need any commit message. Compared with prior work, CClassifier is unique in its characterization and representation of commits.

- Characterization. CClassifier is the first to characterize commits by conducting static inter-procedural program analysis on edited Java files. Compared with prior work that characterizes program changes based on token sequences or abstract syntax trees (ASTs), our characterization is deeper. It models the syntactic dependencies between co-edited entities, the semantic relevance between co-edited statements, and code refactoring operations.
- Representation. CClassifier is the first to introduce graphic representations into the prediction of change intents. Compared with prior work that represents changes with vectors of tokens or ASTs, we believe that graphs are more suitable because they have been widely used in static program analysis to visualize code syntax and semantics [10], [11]. Our application of CNN to the graphical representations has never been explored before.

At https://figshare.com/s/2c04d6bde90e761b11a3, we open-sourced our program and data.

II. Related Work

The related work of our research includes change categorization, automatic change comprehension, and program representation for deep learning.

Change Categorization. Tools were built to classify commits [3], [4], [9], [12], [13]. For instance, Mockus et al. [12] and Levin et al. [9] separately developed approaches to identify three possible reasons for software changes: adaptive, corrective, and perfective. In particular, Mockus and Votta extracted keywords from commit messages and defined classification rules to categorize changes [12]. Existing tools adopt commit messages to classify commits, so they can work poorly when the messages are missing or misleading.

Automatic Change Comprehension. Researchers have proposed approaches to automatically comprehend program changes [5], [14]–[17]. For instance, Jackson and Ladd detected differences between two program versions, and conducted control/data dependencies analysis to summarize semantic impacts of applied changes [14]. DeltaDoc leverages symbolic execution and code summarization to describe (1) the runtime conditions necessary for control flows to reach any changed statement, and (2) the effect of changes on program behaviors [15]. We were inspired by these approaches, although none of them classifies commits.

Program Representation for Deep Learning. Researchers explored various ways to represent programs or code changes for deep learning [18]–[21]. For example, Peng et al. parsed an AST for each Java file; they defined a deep neural network to learn the vector representation for each AST node such that the embedding of a parent node is computable based on the embeddings of all children [18]. Gated Graph Neural Networks (GGNN) model programs as graphs [22]. In each graph, a node represents an AST node, while an edge represents either (1) the parent-child relationship, (2) sequential ordering between children of the same parent, or (3) the def-def, def-use, or use-use relationship of variables. However, no existing work represents commits with graphs, or models the relationship between co-applied edits with edges or attributes.

III. CClassifier

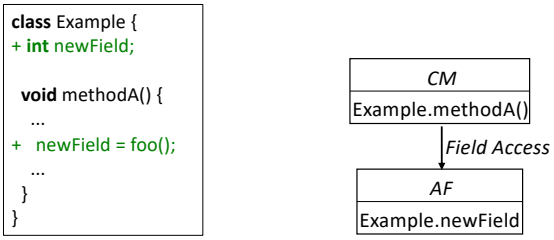
There are three components in CClassifier. Given a commit, the first component extracts edited program entities and identifies their relationship. The second component models all extracted information as a graph; and the third component classifies commits based on those graphs. The following subsections explain each component with more details.

A. Extracting Changes and Their Relations

Given a commit, CClassifier first uses ChangeDistiller [23] to detect changed program entities and statement-level changes, and then conducts static program analysis to identify any syntactic or semantic relation between co-applied edits.

1) Syntactic Program Differencing: For a commit C, CClassifier first retrieves the old and new versions of any edited Java files by C. For each pair of changed files (f_o, f_n), CClassifier uses ChangeDistiller to create an AST for each version; ChangeDistiller then compares the ASTs to generate an edit script consisting of node insertion(s), deletion(s), update(s), and move(s). We leveraged ChangeDistiller to reveal changes at two levels: entity-level and statement-level. Namely, with minor extension, ChangedDistiller can report eight types of entity-level edits: changed method (*CM*), added method (*AM*), deleted method (*DM*), changed field (*CF*), added field (*AF*), deleted field (*DF*), added class (*AC*), and deleted class (*DC*).

Additionally, inside each edited method, ChangeDistiller also detects finer-grained statement-level changes. Because (1)



(a) (b)

Fig. 1: A program commit and its CDG

there are 15 kinds of Java statements (e.g., `if`- and `while`-statements) and (2) ChangeDistiller can generate 4 possible edit operations for each statement: *Insert*, *Delete*, *Update*, and *Move*, in total, ChangeDistiller can report 60 (i.e., $15*4$) types of statement-level operations (e.g., insert an `if`-statement).

2) Static Program Analysis: Based on the detected entity-level changes and statement-level changes, CClassifier applies static program analysis to connect edits based on four types of information: syntactic dependencies, control dependencies, data dependencies, and refactoring operations.

(i) Extraction of Syntactic Dependencies. An entity E_1 is **syntactically dependent** on another entity E_2 if the compilation of E_1 depends on that of E_2 [24]. There are four types of syntactic dependencies: (i) *Contained by*, (ii) *Overridding*, (iii) *Method Invocation*, and (iv) *Field Access*. CClassifier extracts syntactic dependencies because they help explain why certain entities are co-changed. To extract syntactic dependencies, CClassifier adopts a tool InterPart [25], which takes in the old and new versions of every changed Java file and applies inter-procedural Class Hierarchy Analysis (CHA) to each version.

With the analysis results by InterPart, CClassifier creates a **change dependency graph (CDG)** for each commit. A CDG has nodes to represent changed entities, and edges to represent dependencies. For instance, Fig. 1(a) shows that a commit adds a new field and changes a method to access the field. CClassifier generates a CDG (see Fig. 1(b)), where a "*Field Access*" link starts from a *CM* and ends at an *AF* to visualize the dependence relation. If co-changed entities have no syntactic relevance with each other, our CDG will contain a set of nodes without any edge.

(ii) Extraction of Control & Data Dependencies. Both control and data dependencies reflect semantic relevance between statements. A statement S_1 (e.g., `return`-statement) is **control dependent** on another statement S_2 (e.g., `if`-statement), if whether or not S_1 is executed depends on the execution result of S_2 [26]. A statement S_1 is **data dependent** on another statement S_2, if S_1 uses a variable whose value is defined by S_2 [27]. We capture the direct control and data dependencies between edited statements in each method, because such dependencies reflect how control or data flows may be affected by edits. For instance, when an `if`-statement is added to an existing method, new control dependencies may be introduced and the program semantics is modified. Here, we use a static analysis framework WALA [28] to conduct intra-procedural control and data dependency analysis.

(iii) Identification of Refactoring Operations. **Refactoring** applies a series of behavior-preserving transformations to an existing codebase in order to improve the software design [29]. A refactoring operation may edit one or more program entities. CClassifier detects refactoring operations with a state-of-the-art tool: RefactoringMiner [30]. The tool detects 17 types of refactorings. 5 of the 17 types mainly edit single entities. For example, *Extract Variable* is a refactoring to define a variable that holds the evaluation result of an expression. 12 of the 17 refactoring types mainly edit multiple entities. For example, *Move Operation* is a refactoring that moves a method from one class to another class. The detected refactoring operations help CClassifier interpret why some entities were co-changed in certain ways.

B. Graph Modeling

To represent all extracted change data in a uniform way, CClassifier converts CDGs to vectors. This design choice is meaningful for two reasons. First, CDGs overview changed entities, and all statement-level edit operations can be mapped to corresponding changed entities. Second, CDGs demonstrate the structural connections between changed entities, while the entity-level semantic relationship is often established upon such structural relations. For instance, if a field f is not accessed by a method $m()$, it is unlikely that the data/control dependencies or refactorings in $m()$ will have any relevance to f. Specifically, in our graph modeling, we use a 75-attribute vector to characterize each node and a 72-attribute vector to characterize each edge in CDG.

(i) Node Embedding (Vector Representation for Node). As shown in Fig. 2, the first eight one-hot attributes reflect the entity-level change types. Each attribute corresponds to one change type (e.g., *CM*), and is set to "1" if the node has the type. Since each node has only one change type, there is only one "1" among the values of these attributes. The subsequent five attributes show whether any of the five single-entity refactorings is applied. If a refactoring is applied, the corresponding attribute is set to "1" ("0" otherwise).

Next, two attributes are based on the data dependency analysis for each edited method. Specifically, one attribute counts the number of variable definitions involved in edited statements ("*defCounter*" for short), while the other counts the number of variable uses touched by edits ("*useCounter*" for short). In our implementation, CClassifier applies data dependency analysis to both the old and new versions of each edited method, and counts how many variable definitions/uses are covered by edits in respective versions. For instance, when statement `a=1` is deleted and statement `b=2` is inserted, CClassifier increments *defCounter* by 2. The last 60 attributes correspond to statement-level edit operations reported by ChangeDistiller. Given the edit script generated by ChangeDistiller for any edited method, CClassifier clusters operations based on their types, and counts the number of operations for each type.

(ii) Edge Embedding (Vector Representation for Edge). As shown in Fig. 3, the first four attributes reflect the edge types. For example, if an edge has the type "*Overriding*", the 2^{nd}

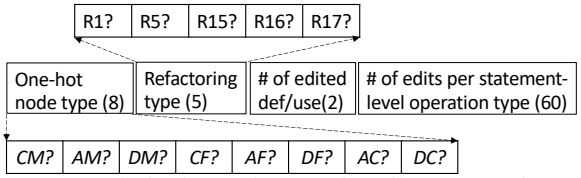


Fig. 2: A node is denoted as a 75-attribute numeric vector

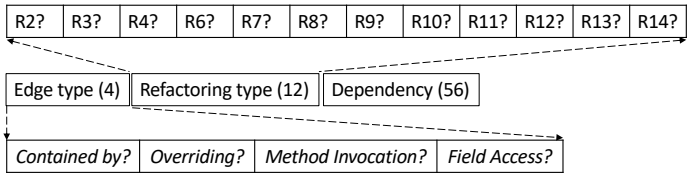


Fig. 3: An edge is denoted as a 72-attribute numeric vector

attribute is set to "1"; otherwise, the attribute is set to "0". The subsequent 12 attributes reflect whether any of the 12 multi-entity refactorings is applied.

The last 56 attributes show the control and data dependency analysis results for individual edited methods. The dependencies exist either between two edited methods or between an edited method and an edited field. For instance, as shown in Fig. 4, two methods are co-changed, and `m2()` is modified to invoke `m1()`. According to the control dependency result for `m2()`, there is an edited statement (i.e., `return;`) control dependent on the invocation of `m1()`. Thus, we set the attribute *cdep(m2, m1, m2→m1)* to "1". Generally speaking, the tuple "$cdep(A, B, A \rightarrow B)$" means that entity A is control dependent on entity B when A invokes B. Since A (or B) can be either *AM*, *CM*, or *DM*, there are seven possible combinations between the change types of both methods:

{*CM_CM, CM_AM, CM_DM, AM_CM, AM_AM, DM_CM, DM_DM*}.

Therefore, we have seven attributes defined for the seven scenarios, with each attribute implying whether or not the corresponding relation exists. Symmetrically, we have additional seven attributes defined for the tuple format "$cdep(B, A, A \rightarrow B)$", seven attributes for "$ddep(A, B, A \rightarrow B)$" (data dependency), and seven attributes for "$ddep(B, A, A \rightarrow B)$".

Similarly, when the control dependencies occur between an edited method and an edited field, we can have the tuple format "$cdep(E, F, E \rightarrow F)$", meaning that method E is control dependent on field F when E accesses F. There are also seven possible combinations between the change types of entities:

{*CM_CF, CM_AF, CM_DF, AM_CF, AM_AF, DM_CF, DM_DF*}.

Symmetrically, we define other seven attributes in the tuple format "$cdep(F, E, E \rightarrow F)$", seven attributes for "$ddep(E, F, E \rightarrow F)$", and seven attributes for "$ddep(F, E, E \rightarrow F)$".

To sum up, 28 attributes are defined to characterize the potential control/data dependencies between two edited methods, and 28 attributes are used to capture the potential dependencies between one edited method and one edited field. For the example shown in Fig. 4, all the 28 attributes about method-field relations are set to "0", because the example illustrates method-method relations.

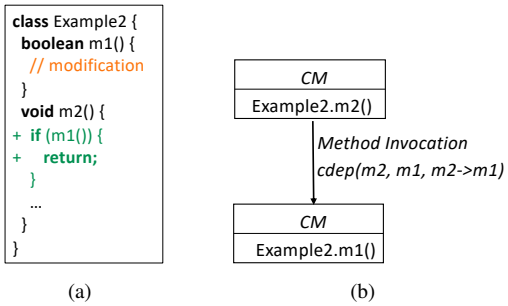
In our implementation, we used the data structures defined in a python library NetworkX [31] to store graphs. Specifically, each graph is saved as a hash map of nodes M_n together with a hash map of edges M_e. M_n uses a unique ID allocated for each node as the key, and the corresponding 75-attribute vector as value. Since every directed edge has a source node and a sink node, M_e uses the unique IDs of both source and sink nodes of each edge as key, and uses the corresponding 72-attribute edge vector as value.

C. Classification with CNN

A convolutional neural network (CNN) is a class of deep neural networks, mostly applied to analyze visual imagery [32]. The general architecture of a CNN usually has three components: (1) a convolutional layer to scan input images for patterns and produce a feature map, (2) a pooling layer (optional) to reduce the feature map dimensionality for computational efficiency, and (3) a fully connected network to output an N-dimensional vector for an N-class classification problem. CNN is used for many applications, including image classification [33], facial recognition [34], and object detection [35]. To classify program commits, we cannot naïvely apply a basic CNN to our graphs for two reasons:

- **Varying graph sizes.** CNN takes images with fixed sizes, while CDGs have a variety of sizes.
- **Embeddings for nodes and edges.** CNN is good at classifying images based on arrays. However, our graph representation contains hash maps for nodes and edges; such representation cannot be easily converted to arrays.

To overcome both challenges, we reused a state-of-the-art approach—Patchy-San [8]—to convert graphs to inputs acceptable by CNN. This section will first introduce Patchy-San and then present our CNN built for graph classification.



(a) (b)

Fig. 4: A *CM* is control dependent on another *CM*

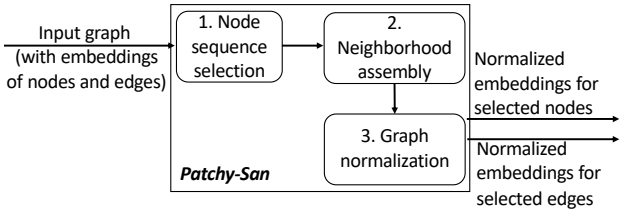


Fig. 5: There are three steps in Patchy-San to normalize input graphs to vectors acceptable by CNN

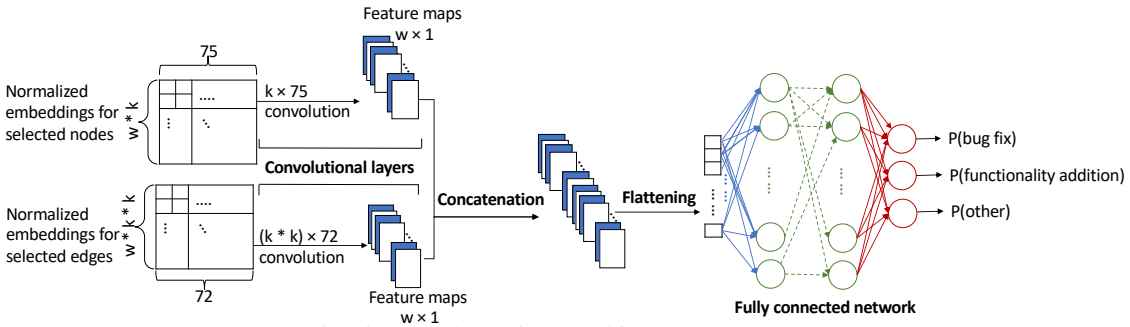


Fig. 6: Overview of CClassifier's CNN structure

1) Our Usage of Patchy-San [8]: As shown in Fig. 5, Patchy-San contains three steps that normalize input graphs to vectors accepted by CNN. Given an input graph G that is represented with a hash map of node embeddings and a nested hash map of edge embeddings, *Step 1* selects the most important w nodes based on the **betweenness centrality** [36] metric, which is calculated as:

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}. \quad (1)$$

In the formula, σ_{st} is the total number of shortest paths from node s to node t, and $\sigma_{st}(v)$ is the number of those paths that go through v. Intuitively, the more shortest paths go through a given node v, the higher $g(v)$ value we obtain, and the more central or important v is.

Step 1 picks a fixed number of nodes with the highest values. This is because when graphs have varying sizes, we always need to create a fixed size of uniform representation for them such that CNN is applicable. By default, Patchy-San sets $w = 18$. We reuse this default setting because in our experiment data, 88% of graphs have at most 18 nodes. We use zero paddings if a graph has fewer nodes, and pick the most important ones when a graph has more nodes. In this way, we ensure the normalized representation to cover all nodes for most graphs.

To reflect the surrounding context of each selected node v, *Step 2* assembles a local neighborhood N for v. Specifically, starting from v, this step conducts breadth-first search, explores reachable nodes with increasing distances from v, and adds nodes into the set N until (1) k nodes are included, or (2) there is no more neighbor to add. By default, $k = 10$.

To normalize the representation for selected nodes and their neighborhoods, *Step 3* ranks the neighbors of each selected node v in the ascending order of their distances to v, and appends dummy nodes if there are insufficient neighbors. Additionally, for each neighbor node $n_n \in N$, this step identifies at most k edges starting from n_n and ending at any neighbor $n_p \in N$, and ranks these edges based on the ranking of their ending nodes. Intuitively, such normalization captures the subgraphs composed by neighbors of w nodes and their edges. Finally, this step outputs (1) $(w*k)$-length node vectors with each node represented with a 75-attribute vector, and (2) $(w*k*k)$-length edge vectors with each edge represented as a 72-attribute vector.

2) The Structure of Our CNN: Fig. 6 shows how our CNN takes normalized input vectors to classify graphs. Specifically, two separate convolutional layers are used to independently take in the normalized embeddings for nodes and edges, as the two types of embeddings have totally different formats. Each convolutional layer outputs 16 $w \times 1$ feature maps. The two sets of maps are then concatenated to obtain 32 $w \times 1$ maps. Next, flattening combines all maps into a single-dimensional vector to create a unified representation for both nodes and edges. Afterwards, the vector is sent to a fully connected network, whose output layer contains three nodes corresponding to three categories. CClassifier categorizes commits by reporting the one whose probability is the highest.

We implemented our CNN based on TensorFlow [37]. CClassifier takes a supervised learning approach to apply CNN for commit classification, so there are two phases in CClassifier: training and testing. Phase I trains a CNN model based on labeled commits, while Phase II exploits the trained model to predict the category for any given program commit. Notice that we chose Patchy-San CNN over other graph neural networks (e.g., GGNN [22]) mainly because the other neural networks cannot take edges with attributes as inputs.

IV. Experimental Evaluation

In this section, we introduce the data set for evaluation, and then present the comparison between CClassifier and a baseline technique. Finally, we discuss how the parameter setting in CNN influences CClassifier's effectiveness.

A. Apache Code Change Data

Our data includes commits from five open-source Apache projects: ActiveMQ [38], Aries [39], CarbonData [40], Cassandra [41], and Mahout [1]. We included the commit data of these five projects for three reasons. First, they are well-maintained and from different application domains. Second, the quality of commit messages is quite good, which can facilitate the baseline technique [9] to work well. Third, many commit messages refer to issue IDs, whose corresponding reports in the issue tracking system have category labels manually defined by developers. When constructing the dataset, we removed some commits from each project if (1) they correspond to duplicated issues, (2) they are not related to any issue, or (3) they cannot be processed by InterPart or RefactoringMiner due to technical issues. As shown in Table I, our data set has 7,414 labeled commits, including 3,902 bug

TABLE I: Labeled program commits of five projects

Project	Bug Fix	Functionality Addition	Other	Total
ActiveMQ	1,003	156	520	1,679
Aries	459	272	368	1,099
CarbonData	170	22	162	354
Cassandra	1,905	329	1,223	3,457
Mahout	365	89	371	825
Sum	3,902	868	2,644	7,414

fixes, 868 functionality additions, and 2,644 commits of other types (e.g., refactoring).

Due to the time limit, we did not further expand our dataset by collecting more labeled commits. Actually, this dataset is already larger than those used by prior work [3], [4], [9], [42]. We also contacted the authors of prior work to reuse their datasets, but were unable to obtain the data.

B. Experiment Settings

Levin et al.'s approach (we use "Baseline" for short) [9] uses traditional classification algorithms (e.g., J48), and extracts 68 features from commit messages and code changes. In particular, 48 features are based on the output by ChangeDistiller; they correspond to the change types applied to different program elements (e.g., methods, parameters, and return types), with each feature counting the frequency of one type. Another 20 features correspond to a predefined list of 20 keywords, with each feature counting the frequency of one keyword in any commit message.

We chose Levin et al.'s approach as the baseline, because it is the state-of-the-art commit classification technique that classifies commits into three categories. Since the implementation of Levin et al. is not publicly available[1], we carefully rebuilt the tool based on the paper description. In that paper, the category "adaptive" is about new feature introduction; "corrective" corresponds to bug fixes; and "perfective" refers to system improvements. Thus, we naturally map these category labels generated by Baseline to our classes: "Bug Fixes (BF)", "Functionality Addition (FA)", and "Other(O)".

There are two settings in our comparative experiment: **within-project** and **cross-project**. For the within-project setting, we evenly split the commit data of each project into five portions and conducted five-fold cross validation [43]. Namely, we ran each tool five times. Each run used four portions of data for training and the remaining one portion for testing. Afterwards, we calculated the average precision, recall, and F-score values among all runs for each project. For the cross-project setting, we also ran both tools five times. Each run used the data of four different projects for training and the data of a fifth project for testing.

1) The Results of Within-Project Setting: Table II presents the effectiveness comparison between CClassifier and Baseline per commit category per project. For bug fixes, on average, CClassifier acquired a higher F-score than Baseline (72% vs. 70%). Among the five projects, CClassifier identified bug fixes with higher F-scores in four projects (i.e., ActiveMQ, Aries, Cassandra, and Mahout), while Baseline achieved a higher F-score in CarbonData (74% vs. 71%).

For functionality additions (FA), CClassifier worked significantly better than Baseline in all projects. Especially, CClassifier obtained 17% average F-score among all projects, while Baseline could not identify any FA commit in four projects. Even though Baseline labeled some commits in Aries as FA, its F-score 24% is much lower than that of CClassifier (44%). For the commits related to other purposes (O), CClassifier outperformed Baseline by obtaining much higher F-scores in three projects (i.e., ActiveMQ, Aries, and Cassandra), while Baseline worked better in the other two projects (i.e., CarbonData and Mahout). On average, CClassifier achieved a considerably higher F-score for O commits than Baseline (i.e., 42% vs. 22%).

Finding 1: *For the within-project setting, CClassifier worked better than Baseline by classifying commits more accurately in most scenarios. CClassifier predicated commits of FA and O categories more effectively than Baseline, although it only extracts features from code changes while Baseline also extracts features from commit messages.*

2) The Results of Cross-Project Setting: Similar to what we observed for the within-project setting, we saw similar comparison results in Table III for the cross-project setting. Specifically for bug fixes, on average, CClassifier acquired higher precision (60% vs. 56%), lower recall (87% vs. 94%), and higher F-score (71% vs. 70%). For functionality additions, CClassifier managed to identify such changes in two projects, while Baseline could not detect such changes for any project. The poor results by both tools imply that it is very challenging to identify commits of functionality additions. For O commits, CClassifier obtained lower precision (50% vs. 53%), higher recall (33% vs. 18%), and higher F-score (38% vs. 26%). CClassifier generally worked better than Baseline in most classification scenarios.

Finding 2: *For the cross-project setting, CClassifier outperformed Baseline in more scenarios.*

When comparing Table III with Table II, we observed that both tools worked less effectively in the cross-project setting. This is expected, because the program commits in different projects can be quite different from each other, and the developers of different projects may have distinct opinions on commit classification. For instance, as shown in Fig. 7–Fig. 9, all three commits similarly modify condition checks. The commit from ActiveMQ (see Fig. 7) inserts an `if`-condition check, and removes an `else`-branch while keeping the content statement. On the other hand, a commit from Cassandra (see Fig. 8) updates an `if`-condition, and removes an `if`-statement while keeping part of the content. Another Cassandra commit (see Fig. 9) updates an `if`-condition. However, developers of ActiveMQ classified the former commit as "FA", and Cassandra developers classified the latter two commits as "BF". In the cross-project setting, CClassifier categorized all

[1]We emailed Levin et al. to ask for the original tool implementation and their evaluation data set, but achieved no success.

TABLE II: Effectiveness (%) comparison between CClassifier and Baseline for the within-project setting

Test Project	CClassifier									Baseline								
	Bug Fix (BF)			Functionality Addition (FA)			Other (O)			Bug Fix (BF)			Functionality Addition (FA)			Other (O)		
	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
ActiveMQ	67	89	76	37	10	15	49	28	35	60	100	75	-	0	-	65	3	7
Aries	55	81	65	59	35	44	53	38	44	44	95	60	71	15	24	47	6	11
CarbonData	63	81	71	88	17	39	69	56	61	75	74	74	-	0	-	65	74	69
Cassandra	62	88	73	45	2	5	54	32	40	57	96	72	-	0	-	60	12	19
Mahout	61	80	69	49	16	24	64	53	58	63	67	65	-	0	-	59	70	64
Average	**62**	**86**	**72**	**48**	**11**	**17**	**54**	**36**	**42**	**57**	**92**	**70**	**-**	**2**	**-**	**59**	**18**	**22**

TABLE III: Effectiveness (%) comparison between CClassifier and Baseline for the cross-project setting

Test Project	CClassifier									Baseline								
	Bug Fix (BF)			Functionality Addition (FA)			Other (O)			Bug Fix (BF)			Functionality Addition (FA)			Other (O)		
	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
ActiveMQ	67	85	75	-	0	-	44	33	37	62	96	75	-	0	-	52	14	22
Aries	53	79	64	-	0	-	40	45	42	44	93	59	-	0	-	44	15	23
CarbonData	57	80	67	-	0	-	61	43	50	54	98	69	-	0	-	80	22	35
Cassandra	61	92	73	21	0	1	54	25	34	60	92	73	-	0	-	51	22	30
Mahout	55	85	67	5	1	1	59	40	48	46	98	63	-	0	-	65	9	16
Average	**60**	**87**	**71**	**-**	**0**	**-**	**50**	**33**	**38**	**56**	**94**	**70**	**-**	**0**	**-**	**53**	**18**	**26**

```
 if (!pivots.isEmpty()) {
   ...
+  if (message.size() !=0)
    return (MessageReference[])messages.toArray(
      new MessageReference[messages.size()]);
 }
- else {
     return new MessageReference[] {
       (MessageReference) messages.removeFirst()};
- }
```

Fig. 7: A commit in ActiveMQ that was manually labeled as "FA" but automatically classified as "BF" [44]

```
- if (arguments.length < 2)
+ if (arguments.length <= 2)
  { badUse("rebuild_index requires ks and cf args"); }
- if (arguments.length >= 3)
    probe.rebuildIndex(arguments[0], arguments[1], arguments[2].split(","));
- else
-   probe.rebuildIndex(arguments[0], arguments[1]);
```

Fig. 8: A commit in Cassandra that was both manually and automatically labeled as "BF" [45]

of these three commits as "BF".

Finding 3: *CClassifier and Baseline worked more effectively for within-project predictions than for cross-project predictions, due to the divergence between different projects and their developers.*

C. Sensitivity to Parameters in CNN

Four parameters were tuned to configure CClassifier. For graph normalization, there is a parameter k to decide the number of neighbors we need to capture for each selected important node. For the convolutional layer, there is a parameter n_f to decide the number of neurons used to generate feature maps. For the fully connected network, there is a parameter n_l to decide the number of hidden layers included before the output layer, and another parameter n_n to decide the number

```
-  if (t.timestamp() < getMaxPurgeableTimestamp()
-      && t.data.isGcAble(controller.gcBefore))
+  if (t.data.isGcAble(controller.gcBefore)
       && t.timestamp() < getMaxPurgeableTimestamp())
  { ...
```

Fig. 9: Another commit in Cassandra that was both manually and automatically labeled as "BF" [46]

of neurons in each hidden layer. We were curious how sensitive CClassifier is to these parameters, so we conducted experiments in the within-project setting, to investigate different parameter values.

Due to the space limit, this paper does not present the measured data of CClassifier's effectiveness with different parameter settings. Please visit our project website for more details. In summary, CClassifier's effectiveness is seldom affected by the number of neighbors k or the number of filters used n_f. As the number of hidden layers n_l increased, CClassifier worked worse. As the number of hidden nodes n_n increased, CClassifier's effectiveness increased first and then became stable. Therefore, by default, we set $k = 10$, $n_f = 16$, $n_l = 1$, and $n_n = 128$.

V. Conclusion

This paper presents CClassifier, a learning-based approach to classify commits purely based on program changes, without requiring for any commit message. Different from prior work, CClassifier is novel in three aspects. First, in addition to recognizing program changes, CClassifier captures various syntactic and semantic relationship between co-applied changes. By simultaneously characterizing individual subsets of related changes, CClassifier intends to model the overall modification semantics for the whole program commit. Second, CClassifier represents all changed entities and their relations as nodes, edges, and attributes in a graph; no prior work extracts such

diverse information or encodes the data in such a uniform way. Third, CClassifier adopts Patchy-San to convert graphs to vectors processable by CNN; CClassifier is the first to repurpose the technique for commit classification.

Our evaluation shows that CClassifier worked better than the state-of-the-art learning-based approach, even though CClassifier does not rely on developers to provide commit messages. CClassifier outperformed the baseline technique by correctly identifying the three types of commits in more scenarios. In the future, we will further improve CClassifier's classification accuracy by exploring the usage of other CNN architectures (e.g., residual neural networks and dense neural networks).

ACKNOWLEDGMENT

We thank anonymous reviewers for their valuable feedback. We thank Ye Wang for his initial contribution to our research. This work was supported by NSF-1845446 and National Key R&D Program of China No. 2018YFC083050.

REFERENCES

[1] "Mahout," https://mahout.apache.org/, 2019.
[2] "Jira," https://www.atlassian.com/software/jira.
[3] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "Automatic classication of large changes into maintenance categories," in *2009 IEEE 17th International Conference on Program Comprehension*, May 2009.
[4] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012.
[5] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017.
[6] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," *CoRR*, vol. abs/1704.04856, 2017. [Online]. Available: http://arxiv.org/abs/1704.04856
[7] Y. Wang, N. Meng, and H. Zhong, *CMSuggester: Method Change Suggestion to Complement Multi-entity Edits*, 2018.
[8] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning convolutional neural networks for graphs," in *International conference on machine learning*, 2016.
[9] S. Levin and A. Yehudai, "Boosting automatic commit classification into maintenance activities by utilizing source code changes," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 2017.
[10] B. G. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, 1979.
[11] D. Callahan, "The program summary graph and flow-sensitive interprocedual data flow analysis," *SIGPLAN Not.*, 1988.
[12] Mockus and Votta, "Identifying reasons for software changes using historic database," in *Proceedings 2000 International Conference on Software Maintenance*, 2000.
[13] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, 2008.
[14] Jackson and Ladd, "Semantic diff: A tool for summarizing the effects of modifications," in *Proceedings 1994 International Conference on Software Maintenance*, Sep. 1994.
[15] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2010.
[16] S. Rastkar and G. C. Murphy, "Why did this code change?" in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.
[17] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014.
[18] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management - Volume 9403*, 2015.
[19] H. Hata, E. Shihab, and G. Neubig, "Learning to generate corrective patches using neural machine translation," *CoRR*, 2018.
[20] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *CoRR*, 2018.
[21] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, 2019.
[22] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *CoRR*, 2017.
[23] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, 2007.
[24] B. G. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001.
[25] Y. Wang, N. Meng, and H. Zhong, "An empirical study of multi-entity changes in real bug fixes," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018.
[26] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, 1991.
[27] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1983.
[28] I. T. W. R. Center, "Wala," 2006. [Online]. Available: http://wala.sourceforge.net/
[29] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.
[30] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, 2018.
[31] "NetworkX," https://networkx.github.io, 2020.
[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012.
[33] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, and W. Xu, "Cnn-rnn: A unified framework for multi-label image classification," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.
[34] O. M. Parkhi, A. Vedaldi, and A. Zisserman, "Deep face recognition," in *British Machine Vision Conference*, 2015.
[35] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems 28*. Curran Associates, Inc., 2015.
[36] M. E. J. Newman, "A measure of betweenness centrality based on random walks," *Social Networks*, 2005.
[37] "TensorFlow," https://www.tensorflow.org, 2020.
[38] "Activemq," https://activemq.apache.org/, 2019.
[39] "Aries," https://aries.apache.org/, 2019.
[40] "Carbondata," https://carbondata.apache.org/, 2019.
[41] "Cassandra," https://cassandra.apache.org/, 2019.
[42] M. Yan, Y. Fu, X. Zhang, D. Yang, L. Xu, and J. D. Kymer, "Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project," *Journal of Systems and Software*, 2016.
[43] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, 1995.
[44] "[amq-3379] implement eviction strategy based on property value uniqueness," https://issues.apache.org/jira/browse/AMQ-3379, 2011.
[45] "[cassandra-7038] nodetool rebuild_index requires named indexes argument," https://issues.apache.org/jira/browse/CASSANDRA-7038, 2014.
[46] "[cassandra-11834] don't compute expensive maxpurgeabletimestamp until we've verified there's an expired tombstone," https://issues.apache.org/jira/browse/CASSANDRA-11834?attachmentOrder=desc, 2016.