Harnessing Syntactic Feature for Code Representation Learning

Bryan Clevor *, Rodolfo Patel, Wendy Slyder

Posted Date: 20 December 2023

doi: 10.20944/preprints202312.1463.v1

Keywords: code edit classification; abstract syntax tree; code structure representation

Article

Harnessing Syntactic Feature for Code Representation Learning

Bryan Clevor *, Rodolfo Patel and Wendy Slyder

Briar Cliff University

* Correspondence: bclevor@briarcliff.edu

Abstract: The paradigm of leveraging *code as a dataset* has recently gained traction, offering innovative solutions in domains such as automated commit message generation, pull request description automation, and program repair mechanisms. Consider the challenge in generating commit messages: traditional methods treat source code as a mere token sequence, applying neural machine translation models. This approach, however, overlooks the critical syntactic structures inherent in programming languages, which could offer deeper insights and improved accuracy. Building upon prior research, specifically the Code2Seq framework, which utilized Abstract Syntax Tree (AST) structural data for source code representation to automate method name generation, this paper extends and refines this concept. We introduce "CSR", a novel methodology adapted to represent code edits effectively. This paper investigates the impact of employing syntactic structure, focusing on the classification of code edits. Drawing inspiration from Code2Seq, "CSR" utilizes AST's structural properties, particularly the paths connecting leaf nodes, to enhance the task of *code edit classification*. This approach is rigorously evaluated on two distinct datasets, comprising fine-grained syntactic edits. Our comprehensive experiments reveal that incorporating syntactic structures does not significantly outperform simpler methodologies. While methods like Code2Seq and our proposed "CSR" show potential, our findings highlight that there is considerable scope for improvement and refinement before such techniques can be universally applied for learning representations of code edits. We anticipate that our findings will spark further research in this field, paving the way for more effective use of syntactic structures in code representation.

Keywords: code edit classification; abstract syntax tree; code structure representation

1. Introduction

The burgeoning growth of open-source software and the widespread adoption of version-control platforms like GitHub [1] have ushered in a new era of publicly accessible code. This phenomenon has birthed the concept of *code as data*, which refers to the utilization of code as a dataset for machine learning algorithms to derive meaningful insights and create innovative tools. Deep learning, a powerhouse in areas such as natural language processing, image recognition, and speech analysis, is now increasingly being applied to the realm of coding.

A notable area of exploration is *code embeddings*. Drawing parallels to word embeddings in natural language processing (NLP) [2–5], recent studies have demonstrated how code embeddings can efficiently summarize code [6–9], automate documentation generation [10,11], and identify code clones [14,15]. The "naturalness hypothesis" posits that despite software's inherent complexity, statistical models can substantially capture its regular patterns [16], fueling the adoption of embedding-based approaches for code.

Despite these advancements, significant discrepancies remain between natural language and code. Code not only possesses a unique syntactic structure but also demonstrates extensive long-range correlations and a larger vocabulary compared to natural language, showing less tolerance to minor alterations [16]. Prior research has addressed these complexities by incorporating syntactic structures such as data and control-flow graphs [21–23], and abstract syntax trees (AST) [24,25], typically focusing on whole code snippets like methods or classes [26–28].

 doi:10.20944/preprints202312.1463.v1

This paper probes the question: *"Can techniques capturing syntactic structures effectively interpret code edits, as opposed to entire code snippets?"* It's imperative to distinguish between code edits and code snippets; the former can span multiple files and methods without a fixed context, while the latter, as analyzed in previous studies, usually encompasses a single method or class [26,28].

Inspired by Code2Seq [28–30], which harnessed programming languages' syntactic structures to summarize code snippets, we adopt a similar approach, now termed Code Structure Representation (CSR). CSR, akin to Code2Seq, is language-agnostic and represents code edits as a collection of path-contexts within the AST. It compresses each path into a fixed-length vector using bi-directional LSTMs and employs an attention mechanism to obtain a weighted average of these path vectors for representing code edits in a vector space. This representation is then utilized for classifying code edits.

Our exploration centers around two principal tasks: bug-fix classification and code transformation classification. Despite Code2Seq's success in method name generation [28] and its purported generalizability, our examination of CSR in the realm of code edit classification offers a new perspective. We benchmark CSR against two models: a) treating code as a token collection in a Bag-of-Words (BoW) model, and b) viewing code as a sequence of tokens in an LSTM model.

Our findings indicate that the syntactic structures, such as path contexts, do not significantly enhance code edit classification efficacy. We observed that:

- Previous studies utilizing syntactic structures [26,28] focused on method name prediction, benefiting from identifiers in AST terminal nodes. In contrast, code edit classification does not gain from specific identifier names.
- Training models with syntactic constructs for code edit classification likely requires more data than currently available: CSR is trained on over 15 million data points, whereas our edit datasets are orders of magnitude smaller.

We conclude that while syntactic structures are effective for code snippet analysis, their application in learning code edit representations is still in its nascent stage.

The remainder of this paper is structured as follows: Section 2 reviews related work. Section 3 describes the architectures of different models for this task. Section 4 details the tasks and datasets used and discusses our evaluation methods. And finally, Section 5 concludes the paper and calls for further research in code edit classification.

2. Related Work

Our research intersects three key areas of computational study: applying Natural Language Processing (NLP) techniques to code, leveraging syntactic structures for code embeddings, and exploring diverse methods for learning distributed representations of code edits. This section elaborates on the significant contributions in these domains.

The concept of code as a natural language, inspired by the naturalness hypothesis, has gained momentum due to the vast availability of code in public repositories. This has led to a surge in applying NLP-based embedding techniques to code, treating it as a sequence of tokens. A notable example is `CODE-NN` by Iyer et al. [7], which introduces a neural attention model utilizing LSTMs for summarizing C# and SQL code, surpassing traditional tf-idf based models. Li et al. [31] further explored this realm, comparing `CODE-NN` with other techniques for neural code search. They found that attention-based weighting schemes on code embeddings exceeded the performance of complex models like `CODE-NN`. However, these approaches primarily treat code as a linear sequence of tokens, typically focusing on well-defined code blocks such as functions or classes, and often overlook the syntactic nuances of the code.

As highlighted in Section 1, code is distinct from natural language due to its intricate syntactic structure and extensive long-range correlations. Previous research has endeavored to capture these attributes by employing data and control-flow graphs, abstract syntax trees (ASTs), and other such structures [21–25]. Two pivotal works in this area are `code2vec` [26] and `code2seq` [28].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 December 2023 doi:10.20944/preprints202312.1463.v1

Both methodologies represent code snippets as sets of path-contexts derived from ASTs, utilizing an attention mechanism over these contexts for method name prediction. `code2vec` treats each path-context as a single token, while `code2seq` interprets them as sequences of tokens. The success of `code2seq` in code summarization, particularly in method name prediction, has inspired us to adapt its principles to our Code Structure Representation (CSR) model for code edit embeddings.

Gated Graph Neural Networks (GGNNs) [35] present another approach, extending ASTs into more comprehensive graphs by incorporating various code dependencies as edges. The tree-based capsule network [33] represents an alternative, capturing both syntactic structures and dependencies in code without explicitly modifying trees or breaking down larger structures. These methods, however, primarily focus on representing entire code snippets, not specifically addressing the nuances of code edits.

Research on code edits has predominantly targeted specific applications like commit message generation and automatic program repair. Early works in commit message generation by Loyola et al. [37] and Jiang et al. [38] utilized attentional encoder-decoder architectures to generate messages from git diffs. Liu et al. [40] assessed NMT-based techniques for this task, proposing a simpler Nearest Neighbour algorithm (`NNGen`) based approach.

A significant contribution in this area is `commit2vec` [41], where Lozoya et al. employed AST-based code representations learned by `code2vec` for binary classification of security-related commits, using transfer learning from pre-trained embeddings. Although they achieved promising results on a modest dataset, their approach and ours differ in scale and complexity; we employ the CSR model for multi-class classification on larger datasets in multiple programming languages.

`DeepBugs` [43], a bug-detection technique relying on identifier name embeddings, and Allamanis et al.'s [44] framework for learning distributed representation of edits, offer alternative perspectives. Both approaches, however, tend to overlook the syntactic structure of code in their embedding learning process.

In summary, our work extends these foundational studies by employing CSR to learn code edit representations. We specifically focus on the application of syntactic structure in understanding and classifying code edits, a relatively unexplored domain in the intersection of code analysis and machine learning.

3. Methodology

In our study, we have developed three distinct models to assess the effectiveness of utilizing AST (Abstract Syntax Trees) for learning enhanced representations of code edits. These models serve as classifiers and vary in their approach to representing the input code snippets, both pre- and post-edit. We detail the architecture and construction of these models here.

3.1. `edit2vec`

`edit2vec` employs syntactic structure by representing code through paths between terminal nodes in ASTs, akin to the approach used in the `code2seq` model. While our model is inspired by `code2seq` [28], it diverges in two significant aspects:

- **Characterizing Code Edits:** Unlike `code2seq`, which only inputs a single code snippet, `edit2vec` inputs a pair of code snippets to encapsulate both pre- and post-edit states. This differentiation is crucial for capturing the nature of the edit.
- **Classification over Generation:** Our model is designed for classification tasks, not generation. Consequently, we replace the decoder in `code2seq` with a classification layer, specifically employing a `softmax` layer for multi-class classification.

We illustrate the overall design of `edit2vec`. This model is versatile, being language agnostic and adaptable to code changes of varying lengths within a single file. We now elucidate how a code edit is represented as an input to our model.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 December 2023 doi:10.20944/preprints202312.1463.v1

Path-Context Extractor

A code edit is characterized as the pair $\{c_{old}, c_{new}\}$, where c_{old} and c_{new} denote the code before and after the edit, respectively. Corresponding ASTs, $\{t_{old}, t_{new}\}$, are generated for these code snippets. In line with `code2seq`, the syntactic structure is captured using path-contexts, defined as the shortest path connecting two terminal (leaf) nodes in the AST. These paths encompass sequences of terminal and non-terminal nodes, starting with a terminal node (the left-context), proceeding through non-terminal nodes (the path), and ending at another terminal node (the right-context).

Given the potentially large number of path-contexts in an AST, especially for extensive code changes, we select a fixed number of path-contexts (40 in our case) to ensure a comprehensive yet manageable representation. This approach is particularly effective for minor code edits, usually spanning 1-2 lines. For smaller ASTs with fewer than 40 path-contexts, dummy values are used for padding, maintaining uniform input size.

Thus, for each pair of ASTs $\{t_{old}, t_{new}\}$, we derive $\{P_{old}, P_{new}\}$, where $P_{old} = \{p_1^{old}, \ldots, p_{40}^{old}\}$ and $P_{new} = \{p_1^{new}, \ldots, p_{40}^{new}\}$ represent sets of 40 path-contexts from t_{old} and t_{new}, respectively. Table 1 provides an example of this process.

Table 1. Example illustrating Path Context Encoder. The names of the non-leaf tokens are abbreviated; for example, `NE0` stands for `NameExpression0`.

Old code	New code
c_{old} = `processURL(message, depth, baseURL, url)`	c_{new} = `processURL(message, depth, url, baseURL);`
P^{old} ={ `processURL, NE0, MCE, NE3, baseURL` `processURL, NE0, MCE, NE2, depth` `message, NE1, MCE, NE2, depth` ...}	P^{new} ={ `processURL, NE0, MCE, NE4, baseURL` `processURL, NE0, MCE, NE2, depth` `message, NE1, MCE, NE2, depth` ...}
PCE Output = $\{CPCV_1^{old}, CPCV_3^{old}, \ldots. CPCV_{40}^{old}\}$.	PCE Output = $\{CPCV_1^{new}, CPCV_3^{new}, \ldots. CPCV_{40}^{new}\}$
CE output = r_{old} [160-D vector]	CE output = r_{new} [160-D vector]

Path-Context Encoder (PCE)

The path-contexts are encoded using a technique akin to `code2seq` into a 128-dimensional vector, termed Compact Path-Context Vector (CPCV). The encoding process for a given path-context involves splitting terminal nodes into sub-tokens and embedding them, while non-terminal nodes are processed through a bi-directional LSTM layer. The resultant vector, a combination of these embeddings, efficiently encapsulates the syntactic structure of the path-context.

Code Encoder (CE)

With no more than 40 path-contexts per code snippet, the PCE outputs 40 CPCVs for each of c_{old} and c_{new}. These CPCVs are then fed into the Code Encoder (CE), which employs an attention mechanism (as used in `code2seq`) to encode each set of CPCVs into a 160-dimensional vector, represented as $\{r_{old}, r_{new}\}$.

Classifier

The concatenated vectors r_{old} and r_{new} from the Code Encoder are passed through a neural network classifier. This classifier comprises a `tanh` layer followed by a `softmax` layer, outputting the class of the code edit.

Model Hyperparameters

The `edit2vec` model is optimized end-to-end for 100 epochs, minimizing categorical cross-entropy loss. We employ the Adam optimizer, with dropout layers strategically placed to mitigate overfitting. The hyperparameters, including the number of hidden units, vector dimensions, and dropout rates, were determined through extensive cross-validation.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 December 2023 doi:10.20944/preprints202312.1463.v1

3.2. LSTM

As a comparative benchmark to `edit2vec`, we constructed an LSTM-based model. This model views code as a linear sequence of tokens, omitting the syntactic structure. It processes the tokenized code snippets through an LSTM layer, followed by dropout, and outputs two 196-dimensional vectors. These vectors are then classified using the same neural network as in `edit2vec`.

3.3. Bag-of-words

Our baseline model, `Bag-of-words`, disregards both syntactic structure and token sequence, treating code purely as a collection of words. It employs count-based and tf-idf vectorization techniques, followed by SVM classifiers. Unlike the other models, `Bag-of-words` does not distinguish between pre- and post-edit code snippets, instead merging their tokens for a unified representation. This approach allows the model to grasp the context surrounding the edit, enhancing its generalizability. The BoW model serves as our baseline, employing classical machine learning techniques that overlook both the syntactic structure and the sequence of code tokens. It treats code purely as a collection of individual words, disregarding their order within the code snippet.

Tokenization and Vectorization

For any given pair of code snippets, c_{old} and c_{new}, which represent the code before and after an edit, we first tokenize them into discrete units. These tokens are then subjected to two distinct vectorization strategies using BoW-based vectorizers:

1. *Count-based Vectorizer:* This vectorizer translates the tokens into vectors based on their frequency counts. It emphasizes tokens that appear more frequently, potentially capturing dominant features in the code.
2. *Tf-idf Vectorizer:* The tf-idf vectorizer assigns weights to tokens not just based on their frequency in a single snippet but also considering their commonness across all code snippets. This approach helps in reducing the impact of universally common tokens, like standard data types in programming languages.

Classification

After vectorizing c_{old} and c_{new} separately, we concatenate these vectors and feed them into a classifier. The classifiers explored in this study include linear-kernel and RBF-kernel Support Vector Machine (SVM) classifiers. The choice of SVM is motivated by its effectiveness in handling high-dimensional data, which is typical in vectorized code representations.

Contextual Consideration

Unlike the CSR and LSTM models, which utilize just the differential tokens (tokens that were added or deleted in the edit), the BoW model incorporates both c_{old} and c_{new} in their entirety. This approach is designed to capture the contextual environment surrounding the code edit. For instance, consider the following examples:

- **Example 1:** `os.file(path)` → `os.folder(path)`
- **Example 2:** `file.getSize()` → `folder.getSize()`

In both cases, the differential tokens are {`file`, `folder`}. However, while Example 1 falls under the category of '*different method same args*', Example 2 is classified as '*change caller in function call*'. Utilizing both c_{old} and c_{new} allows the BoW model to capture these nuances and accurately classify such edits.

Dataset and Sample Distribution

Table 2 presents various bug templates used in our study, along with a brief description and the number of samples associated with each template. These templates range from changing numerical

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 December 2023 doi:10.20944/preprints202312.1463.v1

values to swapping arguments in function calls, providing a comprehensive dataset for evaluating the effectiveness of the BoW model in classifying diverse code edits.

Table 2. Summary of different bug-fix templates used in the dataset, including their descriptions and the corresponding number of samples for each category.

Bug Category	Description	Sample Count
Function Caller Modification	Verifies if the calling object in a function invocation has been substituted with a different one.	1488
Numerical Literal Alteration	Identifies changes where one numerical literal is replaced by another.	4779
Operand Correction	Examines if any operand in a binary operation was modified.	741
Operator Substitution	Determines if one binary operator was mistakenly exchanged with another of the same category.	1711
Incorrect Method Call	Checks cases where an erroneous function was invoked.	9383
Conditional Statement Expansion	Verifies the addition of an alternative condition (‘||‘ operator) in an if statement.	2095
Conditional Statement Restriction	Assesses the insertion of an additional condition (‘&&‘ operator) in an if statement.	1836
Reduced Argument Method Overload	Checks whether a method with fewer arguments (overloaded) was called.	1040
Additional Argument Method Overload	Identifies if an overloaded version of a function with more arguments was used.	3820
Argument Swap in Function Call	Verifies cases where two arguments in a function call were interchanged.	536
Boolean Literal Switch	Determines whether a Boolean literal was substituted with another.	1531

4. Experimental Setup

This section delineates the methodology for our evaluation of the three models, including CSR. We elaborate on two distinct code edit classification tasks and the corresponding datasets utilized in our analysis.

4.1. Code Edit Classification Task

The central objective of the code edit classification task is to infer the type of modification applied to a piece of source code, given its state before (c_{old}) and after (c_{new}) the edit. We explore this through two specific scenarios:

4.1.1. Bug-fix Classification

Utilizing the *ManySStuBs4J dataset* [45], which aggregates 63,923 single-line bug-fix changes from over 1000 renowned open-source Java projects, our analysis categorizes bug-fixes into 16 distinct templates. The task challenges our models, including CSR, to predict the correct bug-template corresponding to the transformation from c_{old} to c_{new}.

Data Preparation and Selection Criteria

The initial phase of data processing involves tokenizing source code snippets into distinct lexical elements using Python's `javalang`[1] parser. We exclude 3,739 data-points that were not amenable to tokenization. After further refining the dataset by omitting entries from specific bug templates due to missing data or incompatibility with the Path-Context Extractor, we are left with 28,960 data-points spanning 11 bug categories. These steps ensure that each instance in our refined dataset is well-suited for analysis with the CSR model as well as the other approaches.

[1] https://pypi.org/project/javalang/

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 December 2023 doi:10.20944/preprints202312.1463.v1

Data Distribution and Stratification

For a balanced evaluation, we partition the dataset into 26,322 training and 2,638 test samples, maintaining an equal representation of each bug template across both sets. Table 2 offers a comprehensive overview of the bug categories and their frequencies.

4.1.2. Code Transformation Classification

Our second dataset comprises 12,784 code edits, derived from top 250 GitHub C# projects, using the Roslynator analyzers [46]. These analyzers identify and rectify non-compliant code segments based on predefined rules. The challenge here for CSR and the other models is to predict the specific analyzer responsible for each code transformation from c_{old} to c_{new}.

Dataset Overview and Sampling Strategy

The dataset encompasses edits from 10 different analyzers, each serving a unique code transformation purpose. We follow a stratified sampling technique to ensure uniform representation of each analyzer in both the training (comprising 11,617 samples) and test (comprising 1,167 samples) sets. Table 3 presents a detailed summary of each analyzer, along with the number of samples attributed to it.

Table 3. Overview of C# code transformation analyzers with respective descriptions and sample distribution.

Analyzer tag	Description	No of samples
RCS1001	Add braces (when expression spans over multiple lines)	443
RCS1032	Remove redundant parentheses	516
RCS1049	Simplify boolean comparison	574
RCS1085	Use auto-implemented property	2163
RCS1123	Add parentheses according to operator precedence	1428
RCS1124	Inline local variable	1067
RCS1146	Use conditional access	3368
RCS1163	Rename unused parameter to '_'	2053
RCS1168	Change parameter name to base name when they are not the same	816
RCS1220	Use pattern matching instead of combination of 'is' operator and cast operator	356

5. Evaluation Results

This section presents a comparative analysis of the performance of the CSR model (formerly `edit2vec`) against baseline models. We assess these models on the code edit classification tasks outlined in Section 4. We employ a rigorous evaluation methodology, training each model with the optimally tuned hyperparameters. The performance is gauged using the average classification accuracy across three iterations of 10-fold cross-validation. Table 4 encapsulates the results for both the bug-fix and code transformation classification tasks. The table format includes the model names in the first column, followed by the average accuracy values for bug-fix classification (with and without canonicalization), and similar columns for code transformation classification.

Table 4. Comparative classification accuracy for the bug-fix and code transformation tasks, evaluated across different models.

Model	Bug-fix		Code Transformation	
	Accuracy	Accuracy (Canon.)	Accuracy	Accuracy (Canon.)
tf-idf SVM (RBF)	32.34%	58.81%	26.31%	73.82%
tf-idf SVM (linear)	85.30%	67.37%	85.30%	69.49%
count SVM (RBF)	32.34%	72.06%	34.24%	76.31%
count SVM (linear)	86.69%	76.08%	88.39%	74.78%
LSTM	94.47%	99.21%	92.55%	92.77%
code2seq	93.17%	98.44%	92.28%	92.59%

 doi:10.20944/preprints202312.1463.v1

5.1. Insights into Model Performance

Our findings reveal that the LSTM model, which interprets code as a sequence of tokens, notably outperforms other models in accuracy for both tasks (without canonicalization). This superiority can be attributed to the LSTM's capacity to capture the sequence and position sensitivity inherent in many code edits. For instance, in cases like the '*swap arguments*' class, where the primary change is the order of method arguments, the LSTM model effectively recognizes such nuances, a feat that the BoW model cannot achieve due to its inherent design of treating code as an unordered set of tokens.

While the CSR model, utilizing path-contexts to represent syntactic structure, does show an improvement in classification accuracy over the BoW model, it falls short of surpassing the LSTM model. This outcome was initially surprising, as we anticipated an enhancement in accuracy with the incorporation of syntactic structure. To validate these results, we conducted statistical significance tests, including the D'Agostino-Pearson normality test and Student's t-test, which confirmed the statistical significance of the difference in performance between the LSTM and CSR models.

To delve deeper into why LSTM outperforms CSR, we visually inspected the separation and clustering of outputs from both models using t-SNE visualizations. These visualizations revealed that while certain edit classes, such as '*swap arguments*', were distinctly clustered, others like '*change caller in function call*' and '*different method same args*' exhibited considerable overlap, indicating challenges in their classification.

5.2. Canonicalization and Its Impact

In pursuit of understanding the reliance of models on token names, we replicated our experiments with canonicalized context tokens. Surprisingly, canonicalization led to a marked improvement in CSR's performance in bug-fix classification, albeit not enough to surpass LSTM. This improvement was not as pronounced in code transformation classification, possibly due to the larger code snippets involved. Our further manual analysis of misclassified examples suggested that while CSR captures higher-level syntactic representations, it is not as effective in distinguishing between more nuanced edits, which LSTM handles more adeptly.

5.3. Further Analysis

The Results for both LSTM and CSR models offer insightful contrasts. While LSTM forms distinct sub-clusters within classes based on the number of method arguments, such subdivisions are not observed in CSR's outputs. This indicates that CSR captures a different, possibly higher-level syntactic representation of code, which, however, does not always align with the requirements of accurate code edit classification.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 December 2023 doi:10.20944/preprints202312.1463.v1

c_{old} : `url.toDecodedString();`
c_{new} : `url.toString();`
left-context$_{old}$: url
left-context$_{new}$: url
right-context$_{old}$: toDecodedString
right-context$_{new}$: toString
path$_{old}$: NameExpression0, MethodCallExpression , NameExpression2
path$_{new}$: NameExpression0, MethodCallExpression, NameExpression2
tag$_{predicted}$: *different method same args*
tag$_{actual}$: *different method same args*

(a) Example correctly classified by `edit2vec`

c_{old} : `AtmResponse.create();`
c_{new} : `AtmResponse.newInstance();`
left-context$_{old}$: AtmResponse
left-context$_{new}$: AtmResponse
right-context$_{old}$: create
right-context$_{new}$: newInstance
path$_{old}$: NameExpression0, MethodCallExpression, NameExpression2
path$_{new}$: NameExpression0, MethodCallExpression, NameExpression2
tag$_{predicted}$: *change caller in function call*
tag$_{actual}$: *different method same args*

(b) Example incorrectly classified by `edit2vec`

Figure 1. Examples that are both correctly classified by `LSTM`, but only Example 1a is correctly classified by `edit2vec`.

Our analyses and results underscore the complex nature of code edit classification and the nuanced differences in how various models capture and interpret the structural and sequential aspects of code. While CSR offers a novel approach to incorporating syntactic structures, LSTM's ability to understand token sequences proves more effective in the context of our evaluation tasks.

5.4. Threats to Validity

This section explores potential limitations and areas of concern in our study, particularly focusing on how they might affect the validity of our findings in code edit classification using the CSR model.

5.4.1. Internal Validity Concerns

Data Scarcity Challenges

The effectiveness of neural network models like CSR and LSTM often hinges on the availability of substantial datasets. With 490,892 parameters, CSR demands a larger dataset for optimal training compared to the LSTM model, which has 269,147 parameters. There is a possibility that CSR's classification performance could be significantly enhanced with access to a more extensive dataset.

The crux of the challenge lies in accumulating clean and accurately labeled data for code edit classification. Commits and their descriptions by developers are frequently ambiguous or irrelevant, complicating the process of accurate labeling. Furthermore, developers often bundle multiple changes in a single commit, adding to the complexity of identifying and labeling discrete edits. Our reliance

 doi:10.20944/preprints202312.1463.v1

on the ManySStuBs4J dataset [45] for this study, though meticulously curated, is a reflection of the inherent difficulty in gathering large-scale, clean data for this domain.

Model and Encoding Considerations

In our study, the syntactic structure of code is represented through path-contexts, a decision influenced by prior successful applications of this approach [26,28]. While this method is well-established, alternate representations of syntactic structure might yield different outcomes. Furthermore, the architecture of the CSR model, though selected through comprehensive hyperparameter tuning and testing across numerous variations, might not encapsulate the full potential of syntactic structure in code edit classification.

5.4.2. External Validity Concerns

Generalizability Across Tasks and Languages

Our study focused on specific code edit classification tasks within Java and C#. It is plausible that syntactic structures could play a more pivotal role in other types of code edits or in other programming languages. Additionally, we concentrated on relatively small code edits due to data availability constraints. Larger code edits, while potentially benefiting from the use of syntactic structures, pose an even greater challenge for gathering accurately labeled data. As prior research [45] suggests, smaller, single-line edits are more common and thus may offer a more viable dataset for various code edit-related applications.

Exploring Broader Applications

While the CSR model showed limitations in our current study, there may be broader contexts or different types of code edits where its use of syntactic structure could be more advantageous. Future research may explore these avenues, potentially uncovering scenarios where CSR's approach to code representation is particularly beneficial.

In summary, while our study provides valuable insights into the application of syntactic structure in code edit classification, these insights are bounded by the constraints of data availability and the specificities of our model architecture and encoding techniques. Future explorations in this field may reveal additional dimensions where the CSR model's approach could be more effectively leveraged.

6. Conclusion and Future Directions

7. Conclusion

In this study, we introduced the Code Structure Representation (CSR) approach, a novel method for classifying code edits. CSR leverages syntactic structures in conjunction with an attention mechanism to learn distributed representations of code edits. Our rigorous experimental evaluations on two distinct tasks—bug-fix classification and code transformation classification—reveal interesting insights. When compared with baseline models such as LSTM and Bag-of-Words (BoW), we found that while CSR captures high-level syntactic representations of code, these intricate representations do not necessarily translate to superior performance, especially in the context of simpler and smaller code edits.

Our observations indicate that the current implementation of CSR, despite its innovative approach, does not outperform the LSTM model in these specific tasks. This might be partly attributed to the nature of the code edits under examination, which are relatively straightforward and may not require the complex syntactic understanding that CSR provides. However, this opens up several avenues for future exploration in this domain.

7.1. Future Work

Expanding the Scope of CSR.

Future research can extend the application of CSR to more complex code edits or entirely different programming tasks. Exploring its efficacy in larger, multi-line code edits or in contexts where syntactic nuances play a pivotal role could yield different results. The adaptability of CSR to diverse programming languages and its effectiveness in broader software engineering applications, such as automatic program repair or code recommendation systems, are potential areas of interest.

Enhancing CSR with Advanced Techniques.

Incorporating advancements in natural language processing, such as transformer models or advanced embedding techniques, could potentially enhance CSR's capability to understand and represent code structures. Exploring hybrid models that combine CSR's syntactic analysis with token sequence-based approaches like LSTM might offer a more robust solution for code edit classification.

Dataset Development and Curation.

A key challenge highlighted in our study is the scarcity of large, well-labeled datasets for code edits. Future efforts could focus on the development and curation of extensive datasets, possibly leveraging techniques like semi-supervised learning or crowdsourcing to annotate and validate code edits. This would not only benefit CSR but also provide a valuable resource for the research community.

Interdisciplinary Approaches.

Integrating insights from fields such as cognitive science or software engineering practice into the development of models like CSR could lead to more intuitive and effective tools for code analysis. Understanding how developers conceptualize and implement code edits might inform more sophisticated model architectures and training methodologies.

In conclusion, while our findings present certain limitations of the CSR model in its current form, they also highlight the untapped potential of syntactic structure analysis in code edit classification. We are optimistic that the insights gleaned from this work will inspire and inform future research in this evolving and exciting field of study.

References

1. GitHub Inc.. https://github.com. [Online; accessed 8-May-2020].
2. Pennington, J.; Socher, R.; Manning, C.D. Glove: Global vectors for word representation. Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), 2014, pp. 1532–1543.
3. Bakarov, A. A survey of word embeddings evaluation methods. *arXiv preprint arXiv:1801.09536* **2018**.
4. Fei, H.; Ren, Y.; Ji, D. Retrofitting Structure-aware Transformer Language Model for End Tasks. Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, 2020, pp. 2151–2161.
5. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.S.; Dean, J. Distributed representations of words and phrases and their compositionality. Advances in neural information processing systems, 2013, pp. 3111–3119.
6. Allamanis, M.; Peng, H.; Sutton, C. A convolutional attention network for extreme summarization of source code. International conference on machine learning, 2016, pp. 2091–2100.
7. Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. Summarizing Source Code using a Neural Attention Model. Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers); Association for Computational Linguistics: Berlin, Germany, 2016; pp. 2073–2083. doi:10.18653/v1/P16-1195.
8. Li, J.; Xu, K.; Li, F.; Fei, H.; Ren, Y.; Ji, D. MRN: A Locally and Globally Mention-Based Reasoning Network for Document-Level Relation Extraction. Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021, 2021, pp. 1359–1370.

9. Fei, H.; Wu, S.; Ren, Y.; Zhang, M. Matching Structure for Dual Learning. Proceedings of the International Conference on Machine Learning, ICML, 2022, pp. 6373–6391.
10. LeClair, A.; Jiang, S.; McMillan, C. A neural model for generating natural language summaries of program subroutines. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019, pp. 795–806.
11. Movshovitz-Attias, D.; Cohen, W. Natural language models for predicting programming comments. Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), 2013, pp. 35–40.
12. Fei, H.; Ren, Y.; Ji, D. Boundaries and edges rethinking: An end-to-end neural model for overlapping entity relation extraction. *Information Processing & Management* **2020**, *57*, 102311.
13. Li, J.; Fei, H.; Liu, J.; Wu, S.; Zhang, M.; Teng, C.; Ji, D.; Li, F. Unified Named Entity Recognition as Word-Word Relation Classification. Proceedings of the AAAI Conference on Artificial Intelligence, 2022, pp. 10965–10973.
14. Tufano, M.; Watson, C.; Bavota, G.; Di Penta, M.; White, M.; Poshyvanyk, D. Deep learning similarities from different representations of source code. 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR). IEEE, 2018, pp. 542–553.
15. White, M.; Tufano, M.; Vendome, C.; Poshyvanyk, D. Deep learning code fragments for code clone detection. 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2016, pp. 87–98.
16. Allamanis, M.; Barr, E.T.; Devanbu, P.; Sutton, C. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* **2018**, *51*, 1–37.
17. Wu, S.; Fei, H.; Li, F.; Zhang, M.; Liu, Y.; Teng, C.; Ji, D. Mastering the Explicit Opinion-Role Interaction: Syntax-Aided Neural Transition System for Unified Opinion Role Labeling. Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence, 2022, pp. 11513–11521.
18. Shi, W.; Li, F.; Li, J.; Fei, H.; Ji, D. Effective Token Graph Modeling using a Novel Labeling Strategy for Structured Sentiment Analysis. Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2022, pp. 4232–4241.
19. Fei, H.; Zhang, Y.; Ren, Y.; Ji, D. Latent Emotion Memory for Multi-Label Emotion Classification. Proceedings of the AAAI Conference on Artificial Intelligence, 2020, pp. 7692–7699.
20. Wang, F.; Li, F.; Fei, H.; Li, J.; Wu, S.; Su, F.; Shi, W.; Ji, D.; Cai, B. Entity-centered Cross-document Relation Extraction. Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, 2022, pp. 9871–9881.
21. Allamanis, M.; Brockschmidt, M. Smartpaste: Learning to adapt source code. *arXiv preprint arXiv:1705.07867* **2017**.
22. Park, E.; Cavazos, J.; Alvarez, M.A. Using graph-based program characterization for predictive modeling. Proceedings of the Tenth International Symposium on Code Generation and Optimization, 2012, pp. 196–206.
23. Nobre, R.; Martins, L.G.; Cardoso, J.M. A graph-based iterative compiler pass selection and phase ordering approach. *ACM SIGPLAN Notices* **2016**, *51*, 21–30.
24. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; Liu, X. A novel neural source code representation based on abstract syntax tree. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019, pp. 783–794.
25. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to Represent Programs with Graphs, 2017, [arXiv:cs.LG/1711.00740].
26. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* **2019**, *3*, 1–29.
27. Fei, H.; Wu, S.; Ren, Y.; Li, F.; Ji, D. Better Combine Them Together! Integrating Syntactic Constituency and Dependency Representations for Semantic Role Labeling. Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, 2021, pp. 549–559.
28. Alon, U.; Brody, S.; Levy, O.; Yahav, E. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* **2018**.

29. Wu, S.; Fei, H.; Ren, Y.; Ji, D.; Li, J. Learn from Syntax: Improving Pair-wise Aspect and Opinion Terms Extraction with Rich Syntactic Knowledge. Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, 2021, pp. 3957–3963.
30. Fei, H.; Li, F.; Li, B.; Ji, D. Encoder-Decoder Based Unified Semantic Role Labeling with Label-Aware Syntax. Proceedings of the AAAI Conference on Artificial Intelligence, 2021, pp. 12794–12802.
31. Cambronero, J.; Li, H.; Kim, S.; Sen, K.; Chandra, S. When deep learning met code search. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019* **2019**. doi:10.1145/3338906.3340458.
32. Fei, H.; Wu, S.; Li, J.; Li, B.; Li, F.; Qin, L.; Zhang, M.; Zhang, M.; Chua, T.S. LasUIE: Unifying Information Extraction with Latent Adaptive Structure-aware Generative Language Model. Proceedings of the Advances in Neural Information Processing Systems, NeurIPS 2022, 2022, pp. 15460–15475.
33. Jayasundara, V.; Bui, N.D.Q.; Jiang, L.; Lo, D. TreeCaps: Tree-Structured Capsule Networks for Program Source Code Processing, 2019, [arXiv:cs.LG/1910.12306].
34. Fei, H.; Ren, Y.; Zhang, Y.; Ji, D.; Liang, X. Enriching contextualized language model from knowledge graph for biomedical information extraction. *Briefings in Bioinformatics* **2021**, *22*.
35. Li, Y.; Tarlow, D.; Brockschmidt, M.; Zemel, R. Gated Graph Sequence Neural Networks, 2015, [arXiv:cs.LG/1511.05493].
36. Fei, H.; Liu, Q.; Zhang, M.; Zhang, M.; Chua, T.S. Scene Graph as Pivoting: Inference-time Image-free Unsupervised Multimodal Machine Translation with Visual Scene Hallucination. Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2023, pp. 5980–5994.
37. Loyola, P.; Marrese-Taylor, E.; Matsuo, Y. A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)* **2017**. doi:10.18653/v1/p17-2045.
38. Jiang, S.; Armaly, A.; McMillan, C. Automatically generating commit messages from diffs using neural machine translation. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* **2017**. doi:10.1109/ase.2017.8115626.
39. Fei, H.; Zhang, M.; Ji, D. Cross-Lingual Semantic Role Labeling with High-Quality Translated Training Corpus. Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020, pp. 7014–7026.
40. Liu, Z.; Xia, X.; Hassan, A.E.; Lo, D.; Xing, Z.; Wang, X. Neural-Machine-Translation-Based Commit Message Generation: How Far Are We? Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering; Association for Computing Machinery: New York, NY, USA, 2018; ASE 2018, p. 373–384. doi:10.1145/3238147.3238190.
41. Lozoya, R.C.; Baumann, A.; Sabetta, A.; Bezzi, M. Commit2Vec: Learning Distributed Representations of Code Changes. *arXiv preprint arXiv:1911.07605* **2019**.
42. Wu, S.; Fei, H.; Qu, L.; Ji, W.; Chua, T.S. NExT-GPT: Any-to-Any Multimodal LLM. *CoRR* **2023**, *abs/2309.05519*.
43. Pradel, M.; Sen, K. Deep Learning to Find Bugs. 2017.
44. Yin, P.; Neubig, G.; Allamanis, M.; Brockschmidt, M.; Gaunt, A.L. Learning to Represent Edits, 2018, [arXiv:cs.LG/1810.13337].
45. Karampatsis, R.M.; Sutton, C. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. *arXiv preprint arXiv:1905.13334* **2019**.
46. Josef Pihrt. [Online; accessed 8-May-2020].