UTANGO: Untangling Commits with Context-Aware, Graph-Based, Code Change Clustering Learning Model

Yi Li
New Jersey Institute of Technology
New Jersey, USA
yl622@njit.edu

Shaohua Wang*
New Jersey Institute of Technology
New Jersey, USA
davidsw@njit.edu

Tien N. Nguyen
University of Texas at Dallas
Texas, USA
tien.n.nguyen@utdallas.edu

ABSTRACT

During software evolution, developers make several changes and commit them into the repositories. Unfortunately, many of them tangle different purposes, both hampering program comprehension and reducing separation of concerns. Automated approaches with deterministic solutions have been proposed to untangle commits. However, specifying an effective clustering criteria on the changes in a commit for untangling is challenging for those approaches.

In this work, we present **UTango**, a machine learning (ML)-based approach that learns to untangle the changes in a commit. We develop a *novel code change clustering learning model* that learns to cluster the code changes, represented by the embeddings, into different groups with different concerns. We adapt the agglomerative clustering algorithm into a supervised-learning clustering model operating on the learned code change embeddings via trainable parameters and a loss function in comparing the predicted clusters and the correct ones during training. To facilitate our clustering learning model, we develop a *context-aware, graph-based, code change representation learning model*, leveraging Label, Graph-based Convolution Network to produce the *contextualized embeddings for code changes*, that integrates program dependencies and the surrounding contexts of the changes. The contexts and cloned code are also explicitly represented, helping UTango distinguish the concerns. Our empirical evaluation on C# and Java datasets with 1,612 and 14k tangled commits show that it achieves the accuracy of 28.6%–462.5% and 13.3%–100.0% relatively higher than the state-of-the-art commit-untangling approaches for C# and Java, respectively.

CCS CONCEPTS

• **Software and its engineering → Software evolution**.

KEYWORDS

Commit Untangling; Deep Learning; Code Change Embeddings

ACM Reference Format:
Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. UTANGO: Untangling Commits with Context-Aware, Graph-Based, Code Change Clustering Learning Model. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549171

*Corresponding Author

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549171

1 INTRODUCTION

During software evolution, developers make changes over time to perform software maintenance tasks. The changes to source files committed to the repository in the same transaction are referred to as a *change set* or a *commit*. For separation of concerns, each commit should be about one purpose or concern regarding the programming task at hand. Unfortunately, it has been reported that many commits tangle different concerns including the changes for bug-fixing, refactoring, enhancements, improvements, or documentation [10, 11, 19, 21, 26]. Such change sets are called *tangled code changes* or *tangled commits* [10, 11]. The prior work reported two reasons for tangled commits from developers' perspective: time pressure in committing the changes, and unclear relations between the concerns for code changes [22].

Tangled commits pose several issues in software development. First, they affect software quality as they hamper program comprehension [26] and reduce the separation of concerns in code changes [22]. Second, the tangled commits might contain the bug-fixing changes for one bug that are mixed with the fixes for other bugs as well as other types of changes for refactoring, enhancements, or documentation [10, 11, 21]. Those tangled commits have negative impact on the accuracy of bug prediction or bug localization models that rely on the changes mined from the repository [10, 11]. Those models are significantly affected by the tangled commits as they consider an entire commit as for fixing or non-fixing.

Recognizing the need of tools that untangle, i.e., decompose a commit into untangled changes, researchers have proposed several approaches. The automated commit-untangling approaches can be broadly classified into two categories: *mining software repositories* [6, 10, 11, 14, 15], and *program analysis* [4, 20, 22, 24].

First, the earlier approaches leveraged *mining software repositories (MSR)* techniques to untangle commits. Herzig *et al.* [10, 11] utilize a confidence voter technique with agglomerative clustering on change operations for untangling. The voters consider data dependencies, call graphs, change couplings, and distances. Kirinuki *et al.* [14, 15] consider a commit as tangled if it includes another commit in the past. However, there are tangled commits whose parts have not occurred in the past. Dias *et al.* [6] use confidence voters on the fine-grained change events in an editor. The confidence scores are converted into the similarity scores via a Random Forest Regressor, which are then used in an agglomerative clustering algorithm to partition the tangled changes.

The second category of untangling approaches leverage the *static analysis* techniques. Roover *et al.* [20] use program slicing to segment a commit across a Program Dependency Graph (PDG). However, it is limited in handling inter-procedural and cross-file dependencies. Barnett *et al.* [4] use def-use chains, and cluster them. If the def-use chain all falls into a method, it is considered as trivial, otherwise, non-trivial. Because ignoring the trivial clusters, it can miss tangled concerns. Flexeme [22] uses multi-version PDG augmented with name flows in the edges, and applies agglomerative clustering using graph similarity on that graph to untangle the commits. SmartCommit [24] uses a graph-partitioning algorithm on a graph representation to capture the relations among code changes (hard and soft links, refactoring links, cosmetic links, etc.).

Despite their successes, the state-of-the-art untangling commit techniques still have limitations. First, the boundaries across concerns in a commit do not necessarily and naturally map to clustering criteria of a clustering algorithm running on the PDG, name flows, program slices, change operations, or the changes themselves. The concerns might be linked via multiple edges. To apply a clustering algorithm on the PDG, program slices, or change graphs, it is challenging to deterministically specify the right criteria to obtain perfect partitions matching with the concerns. Second, the goal is to decompose the changes in a commit. However, the existing approaches *do not consider a change w.r.t. the context of surrounding code with a clear distinction of the changed elements and the unchanged ones in the context.* Such context could help distinguish the concerns for the changes. Finally, not all the changes in the same concern need to have program dependencies among them. The logic connection among the co-changed code in the same commit could be due to the reasons different than program dependencies. For example, two pieces of cloned code realizing the same bubble sorting algorithm have the same bug, e.g., at the comparison operator. They might be changed for the same concern to fix that logic bug despite that they have no data/control dependency.

To address those challenges, in this paper, we propose **UTango**, a **novel code change clustering learning model** that learns to untangle a commit by clustering the code changes (represented by the embeddings) into groups for different concerns. While deterministic clustering criteria on the PDG, slices, or change operations do not always produce the clusters that naturally map to the boundaries between the concerns, a machine learning (ML) model is expected to learn to cluster the changes, thus, untangling a commit. Our ML model learns from the changes belonging to the same concerns in the version history. To build the training data for such learning, we adapt Herzig *et al.* [10]'s method to mine the changes for the same concerns in the version history (Section 3). To facilitate learning to cluster code changes, we develop a **context-aware, graph-based, code change representation learning (RL) model**, leveraging Label, Graph-based Convolution Network [5] to produce the *contextualized embeddings for code changes*.

Our clustering learning model and context-aware, graph-based RL model have the following unique characteristics that facilitates the untangling of code changes. First, we use Label-GCN (Graph Convolutional Network) [5] to model the changes and surrounding code by integrating both the versions before and after changes in a multi-version program dependence graph, δ-PDG [22]. δ-PDG encodes the *program dependencies among the changed and unchanged statements*. Second, to decompose the changes, we *explicitly represent the surrounding code context of each change*. The explicit representation of the context could help UTango learn the important features of a change (e.g., code structures, data/control dependencies) to distinguish its concern among others. Third, UTango also considers an implicit relationship: the cloned code that is similar to the changed code under study. The idea is that the *two cloned code with similar logic might be changed in the same manner in the same concern in a commit.* Fourth, to untangle a commit, in our code change clustering learning model, agglomerative clustering runs on *contextualized embeddings for code changes* that integrates richer, encoded information than the PDG or program slices, thus, helping better distinguish the concerns of the changes. Finally, we *adapt the agglomerative clustering algorithm* into a *supervised-learning clustering model* with trainable parameters and a loss function to adjust the parameters by comparing the predicted clusters and the correct ones during training.

We have conducted several experiments to evaluate UTango. Our experimental results on a real-world C# dataset with 1,612 tangled commits show that UTango achieves the accuracy of 36.4%, 28.6%, 55.2%, and 462.5% relatively higher than the baseline approaches Flexeme [22], δ-PDG+CV [22], Herzig *et al.* [10], and Barnett *et al.* [4]. UTango can correctly cluster 39% of the changed statements into correct concerns. We also evaluated UTango in a Java dataset with 14k+ tangled commits. The results show that UTango achieves 13.3%–100% accuracy relatively higher than the state-of-the-art approaches including SmartCommit [24]. Our sensitivity analysis shows that all designed components in UTango contributes positively to its accuracy. We show that the changed statements in the same concerns are projected nearer to each other than the ones in different concerns, helping UTango better in change clustering. The key contributions of this work include:

1. UTango: an ML-based, commit-untangling approach with a novel code change clustering learning model. It is the first ML model that learns to untangle the commit by learning to cluster the code changes. UTango learns from the changes belonging to the same concern in the version history. We adapt agglomerative clustering into a supervised-learning clustering model.

2. A novel context-aware, graph-based representation learning for code changes. We design GCN-based model to produce the *contextualized embeddings for the code changes*, that integrates program dependencies, changes, contexts, and cloned code.

3. Extensive empirical evaluation. We evaluated UTango against the state-of-the-art approaches for untangling commits to show its better performance. Our tool and data are available at [3].

2 MOTIVATION

2.1 Motivating Examples

Let us present a few real-world examples to motivate UTango.

2.1.1 Example 1. Figure 1 shows the changes at the commit r1192 of the JHotDraw project with the log *"Fixes NPE and implements indentation of XML elements"*. This is a tangled commit with two concerns/purposes: 1) the fix for Null Pointer Exception occurred at lines 4,7, and 10 of the `Drawing` class (the `null` argument was replaced with `Collections.emptyList()`); 2) the implementation of the indentation of XML elements occurred at lines 16–18, and a few

```
1  Commit r1192: Fixes NPE and implements indentation of XML elements.
2  /trunk/jhotdraw8/src/main/java/org/jhotdraw8/draw/Drawing.java
3  ...
4  - public final static Key<List<URL» AUTHOR_STYLESHEETS = new ...
         ("authorStylesheets", List.., new Class<?>[]{URI.class},., null);
5  + public final static Key<List<URI» AUTHOR_STYLESHEETS = new ...
         ("authorStylesheets", List.class, new Class<?>[]{URI.class},...,
         Collections.emptyList());
6  ...
7  - public final static Key<List<URI» USER_AGENT_STYLESHEETS = new
         SimpleFigureKey<>("userAgentStylesheets", List.class, new
         Class<?>[]{URI.class},..., null);
8  + public final static Key<List<URI» USER_AGENT_STYLESHEETS = new
         SimpleFigureKey<> ("userAgentStylesheets", List.class, new
         Class<?>[]{URI.class},..., Collections.emptyList());
9  ...
10 - public final static Key<List<String» INLINE_STYLESHEETS=new ...
         ("inlineStylesheets",List...,new Class<?>[]{String.class},.,null);
11 + public final static Key<List<String» INLINE_STYLESHEETS=new ...
         ("inlineStylesheets", List.class, new Class<?>[]{String.class},...,
         Collections.emptyList());
12 //-----------------------------------------------------------------
13  /trunk/jhotdraw8/src/main/java/org/jhotdraw8/draw/io/SimpleXmlIO.java
14  public Document toDocument(Drawing ..., Collection<Figure> selection)...{
15    for (Figure child : ordered) {
16    - writeNodeRecursively(doc, docElement, child);
17    -}
18    - docElement.appendChild(doc.createTextNode("..."));
19    + writeNodeRecursively(doc, docElement, child, linebreak);
20    +}
21    + docElement.appendChild(doc.createTextNode(linebreak));
22    return doc; ...
23 }
```

Figure 1: A Tangled Commit at r1192 of JHotDraw

```
1   Commit r1023: Fixes bugs in FigureStyleManager
2   /trunk/jhotdraw8/src/main/java/org/jhotdraw8/draw/Drawing.java
3
4  - public final static Key<List<URI» AUTHOR_STYLESHEETS = new
         .("authorStylesheets", List.class, "<URI>",..., null);
5  + public final static Key<List<URI» AUTHOR_STYLESHEETS = new
         .("authorStylesheets", List.., new Class<?>[]{URI.class},., null);
6
7  - public final static Key<List<URI» USER_AGENT_STYLESHEETS=new
         .("userAgentStylesheets", List.class, "<URI>",..., null);
8  + public final static Key<List<URI» USER_AGENT_STYLESHEETS=new
         SimpleFigureKey<> ("userAgentStylesheets", List.class, new
         Class<?>[]{URI.class},..., null);
9
10 - public final static Key<List<String» INLINE_STYLESHEETS=new
         .("inlineStylesheets", List.class, "<String>",..., null);
11 + public final static Key<List<String» INLINE_STYLESHEETS=new
         .("inlineStylesheets",List..,new Class<?>[]{String.class},.,null);
```

Figure 2: Same Statements as in r1192 were Changed at r1023 of JHotDraw and Belonged to Only One Concern

other lines of code in the `SimpleXmlIO` class and one line of code in the `Drawing` class (not shown).

2.1.2 Example 2. Figure 2 shows the changes committed at r1023 earlier in JHotDraw. The changes at the lines 4, 7, and 10 of the `Drawing` class were to the same statements as the ones in the r1192 commit in Figure 1. However, the commit at r1023 was for only one concern as stated in the commit log *"Fixes bugs in FigureStyleManager"*. This example motivates us to build a ML model to learn from the co-changes for the same concern in the version history to untangle the current commit.

Observation 1 [Learn to Cluster Code Changes]. *History of the co-changed statements with the same concern could be a good source for a machine learning model to learn to cluster the changed statements, thus, untangling the current commit.*

2.1.3 Example 3. Figure 3 shows another example in JHotDraw project. At the commit r463, two changes at line 3 and line 13 are exactly the same with the addition of `figure.isSelectable()` to

```
1  public void mousePressed(MouseEvent evt) {
2    ...//SelectionTool.java
3  - if (figure != null)
4  + if (figure != null && figure.isSelectable())
5        newTracker = createDragTracker(figure);
6    } else {
7      if (! evt.isShiftDown()) {...}
8  }
9  //-----------------------------------------------------------------
10 protected void updateHoverHandles(DrawingView view, Figure f) {
11   ...// SelectAreaTracker.java
12   figure = f;
13 - if (figure != null)
14 + if (figure != null && figure.isSelectable())
15       hoverHandles.addAll(figure.createHandles(-1)); ...
16 }
```

Figure 3: Same Change in two Contexts for Different Concerns at r463 of JHotDraw

check whether a figure is selectable or not. However, those *two exact changes* occurred in two different methods `mousePressed` and `updateHoverHandles` for two different concerns/purposes as noted in the commit log. Line 4 was aimed to fix the tracking of a dragging object as the mouse is pressed (*"Fixed bug where setSelectable() on a dragging figure did not work"*). Line 14 was a fix for a different bug as the hovered object needs to be selected (*"Selection classes now check the flag on the hovering figures to be selected."*). The addition of `figure.isSelectable()` is common for checking if a figure is selectable in the two tasks of dragging and hovering a figure. Without the surrounding contexts, one could not tell whether two changes are for the same purpose or not.

Observation 2 [Context]. *The surrounding context consisting of un-changed code is crucial to determine the concern of a change. The same change in two contexts could be for different concerns.*

Figure 1 also gives us an interesting observation. The changes to fix the NPE at lines 4, 7, and 10 are exactly the same: `null` → `Collections.emptyList()`. The three statements are cloned code of one another to support for different stylesheets (author, user agent, and inline stylesheets). In fact, the changes in Figure 2 also occurred at the cloned statements. The three changes belong to the same concern/purpose since they serve as a fix for the same logic despite that there is no program dependency among them. The existing untangling approaches [4, 20, 22, 24] that require the explicit program dependencies among the changes with the same purpose will not classify these committed code as belonging to the same concern.

Observation 3 [Implicit Dependencies]. *Two fragments of cloned code have the same/similar logic, thus, have implicit dependencies, and could be modified in the same manner to serve the same purpose.*

2.2 Key Ideas

Inspired from the above observations, we propose UTANGO with the following ideas.

2.2.1 Key Idea 1 [Code Change Clustering Learning Model]. Instead of deciding a deterministic clustering criterion on the concrete artifacts (PDGs, program slices, code changes, operations, or change graphs), from Observation 1, we build an ML model to untangle the commits by learning to cluster code changes represented by embeddings, w.r.t. different concerns. The model learns from the history of the co-changed statements in the same commits for the same

concerns, and applies to cluster the changes in the current commit. We also modify an agglomerative clustering algorithm into a supervised-learning clustering model via trainable parameters and a loss function to compare the predicted and correct clusters.

2.2.2 Key Idea 2 [Context-aware, Graph-based Representation Learning for Code Changes]. We design a context-aware, graph-based, representation learning model to *learn the contextualized embeddings for the code changes* that integrates *program dependencies* among the program elements, and the *contexts* of code changes. Leveraging the changes in the same concern in the history, we train a Label, Graph-based Convolution Network [5] to learn the embeddings and learn to cluster them. For prediction, we build a supervised-learning agglomerative clustering algorithm that operates on the embeddings of code changes to produce the clusters for the purpose of untangling the commit. We expect that supervised-learning clustering on vectors is more effective than on those artifacts since the embeddings capture richer information integrated from dependencies and contexts.

2.2.3 Key Idea 3 [Explicit Context Representation as a Weight to Compute Vectors for Code Changes]. As seen in Observation 2, the contexts of the code changes can help distinguish their concerns in the commits. We represent code changes and the surrounding context of a change via the multi-version program dependence graph, δ-PDG [22], consisting of the elements of both versions before and after the changes, and program dependencies. The context is defined as the surrounding nodes of the changed statement node in that graph. The Label-GCN is used to model the statements and their program dependencies in δ-PDG as well as to learn the vector representing the context for a change. The context vectors are then used as the weights in learning contextualized embeddings for the code changes.

2.2.4 Key Idea 4 [Implicit Dependencies among Cloned Code]. As seen in Observation 3, the cloned code exhibits implicit dependencies with regard to whether they can be changed in the same commits for the same concerns. Thus, during the process of producing the final result, we also integrate the code clone relationships to adjust the clusters produced by the clustering learning model.

3 APPROACH OVERVIEW

3.1 Important Concepts

Let us first define some important concepts used in UTango.

Definition 1 (Program Dependence Graph). *The **program dependency graph (PDG)** [7] is a directed graph with a set N of nodes and a set E of edges such that each node $n \in N$ represents a program statement or a conditional expression; each edge $e \in E$ represents the data or control flow among the statements.*

Figure 4 shows the code change in a commit in which the statement `int next = i + 1;` is replaced by `var next = i + 1;`. Figure 4 (a) and (b) display the PDGs of the method `FormatYear` before and after the change. All the nodes of the PDG before the change are marked with i, and those of the PDG after the change are marked with j.

Definition 2 (Multi-version Program Dependence Graph). *(δ-**PDG**). A δ-PDG^{i,j} is a directed graph generated from the disjoint union of all nodes and edges in the PDGs at versions i and j [22].*

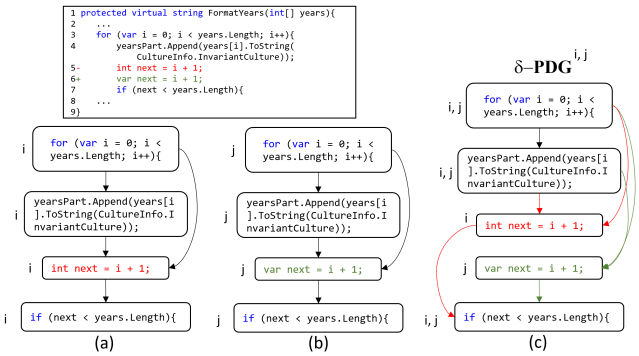


Figure 4: Multi-Version Program Dependence Graph

Figure 4(c) displays the multi-version PDG^{i,j} (δ-PDG^{i,j}) that are built from the two versions i and j of the method `FormatYear` before and after the change. In δ-PDG^{i,j}, the nodes labeled with either i or j appear only in the PDG for the version i or the version j. The nodes labeled with i,j appear in the PDGs at both the versions.

Definition 3 (Changed/Un-changed Nodes). *In the multi-version PDG, δ-PDG^{i,j} for the versions before and after the change, the changed nodes represent the changed statements, and are labeled with either i or j, while the un-changed nodes are labeled with i, j.*

In Figure 4(c), the node labeled with i represents the deleted statement, the node labeled with j represents the added one, while the nodes labeled with i, j are for the un-changed statements.

Definition 4 (Context). *The context C of a changed node n is a sub-graph of the multi-version δ-PDG^{i,j} that includes all the un-changed nodes within the k-hop neighbors of the changed node n, together with all the inducing edges among them.*

In Figure 4(c), when k=1, the context for the changed node/statement at line 5 consists of all three nodes labeled with i, j because they are one hop from the changed node for '`int next = i + 1;`'.

3.2 Architecture Overview

Figure 5 illustrates the overview of our model, UTango.

3.2.1 Step 1. Building Multi-version PDG (δ-PDG^{i,j}) and Contexts. The first step is to build the δ-PDG^{i,j} graph and extract the context sub-graphs for each changed statement from the two versions i and j before and after the changes. We adopt the multi-version graph building algorithm from Flexeme [22]. Specifically, we first generate the PDGs for both versions i and j. We use the Git diff tool on the source code to determine the changed and unchanged nodes for the statements. The added nodes are kept in δ-PDG^{i,j} with the labels j as they appear in the newer version j. We also retain the deleted nodes and use the label i for them. The unchanged nodes between the versions are matched by using string similarity among the respective statements to filter the candidates and line-span proximity to rank them. When considering the edge changes, we back-propagate the delete nodes to the edges flowing into them. We also add all the unmatched edges in the newer version j to the multi-version PDG^{i,j} as the edges relevant to the added nodes. Details on building δ-PDG^{i,j} can be found in another document [22].

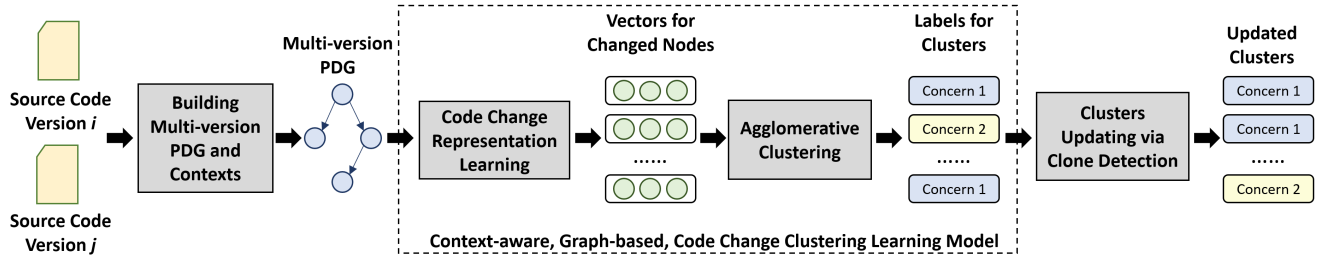


Figure 5: UTango: Architecture Overview

After constructing δ-PDG^{i,j}, for each changed node in the graph, we collect all unchanged nodes within the k-hops and all their inducing edges to build a sub-graph as the context for the changed node. δ-PDG^{i,j} and the contexts for the changed nodes will be used as the input for the next step. This step of building δ-PDG^{i,j} and contexts is used in both training and predicting processes.

3.2.2 Step 2. Context-aware, Graph-based, Code Change Clustering Learning Model. The task of this model is to *learn to cluster the code changes*, represented by the changed nodes and corresponding contexts in δ-PDG^{i,j}. For that, UTango first learns to construct the **contextualized embeddings** to represent the code changes via our novel **context-aware, graph-based code change representation learning model**. In that model, to build the contextualized embeddings, we leverage the Label-GCN [5] that can deal with the nodes with multiple labels (used to denote the versions i, j) to learn the representation vector v for each node n in the graph. For a changed node n_c, we collect the vectors for the un-changed nodes in the context for n_c into a matrix. We use a fully connected layer to convert the matrix into a vector v_{ctx} to encode the contextual information for the node n_c. The context vector v_{ctx} is then used as *a weight to represent the impact of the context on the learning to produce the final vector for the changed node* n_c.

With the contextualized embeddings built for all the changed nodes in δ-PDG^{i,j}, our **code change clustering learning model** uses the hierarchical agglomerative clustering algorithm to cluster the changed nodes represented by their embeddings. That clustering algorithm is adapted into a supervised-learning clustering model as follows. During training, we know the correct clusters of the changed nodes from the training data. We define a *trainable threshold* for the linkage when merging the smaller clusters into a larger one. The trainable parameters of the model and the trainable threshold are computed over many iterations in training as the predicted clusters are compared against the correct clusters in the oracle. We design a loss function considering such comparison to adjust the model's parameters over the iterations. For predicting, the trained model is used to cluster the changed nodes in δ-PDG^{i,j}. The changed nodes in the same cluster are treated as having the same concern.

3.2.3 Step 3. Updating Clusters via Code Clone Detection. After having the resulting clusters from Step 2, we use a code clone detection tool to detect if there is a cloned statement s'_m of a changed statement s_m in the δ-PDG^{i,j}. If so, we check the clusters containing s_m and s'_m. When the clusters are different, we merge those clusters for one concern if needed. After iterating over all the changed statements s_m, we obtain the final resulting clusters.

3.3 Training and Predicting Processes

3.3.1 Training and Predicting. All the steps in Figure 5 are shared between the training and predicting processes. The key difference is that in training, the ground truth labels with respect to the clusters (concern 1, concern 2, etc.) for the changed statements are known, while in predicting, UTango predicts the clusters (concern 1, concern 2, etc.), and finally updates them via our code clone detection process. During clustering the changes in a new commit, the number of clusters is unknown and will be decided by our model.

3.3.2 Training Corpus. To train our clustering learning model, we need to have a corpus of the past commits that were un-tangled into multiple clusters, each for a different concern. We reuse the data collection methodology by Herzig *et al.* [10], which was later used by Partachi *et al.* [22], to build the artificial corpus of tangled commits. The method mimics the way of a developer committing multiple consecutive work units as a single patch, thus, mimics the tangled commits that (s)he makes. The goal is to compose the tangled commits from the atomic ones that were mined from the repositories.

Specifically, we detected the atomic commits and tangled them to produce the tangled commits in a training dataset. We consider the commits to satisfy the following conditions: 1) the changes have been committed by the same developer within 14 days with no commit by the same developer in-between them; 2) the change namespaces whose names have a large prefix match; 3) the commits contain frequently-changed-together files; and 4) the commit logs do not contain "fix", "bug", "feature", etc. multiple times. With this data collection methodology, we can obtain the tangled commits consisting of the clusters of atomic changes for training.

4 CONTEXT-AWARE, GRAPH-BASED, CODE CHANGE, CLUSTERING LEARNING MODEL

After the first step, we obtain the multi-version δ-PDG^{i,j} and the contexts of the changes (Section 3). We present in this section our context-aware, graph-based, code change (CC) clustering learning model. Our clustering learning model has two tasks: 1) taking the computed δ-PDG^{i,j} to learn the representation vectors (embeddings) for the changed code, and 2) performing clustering on those embeddings to cluster the changed statements. During training, we have the ground truth on the clusters of the changed statements, thus, we have the cluster labels (concern 1, concern 2, etc.) for the changed nodes in δ-PDG^{i,j}. During predicting (i.e., clustering), the input δ-PDG^{i,j} will be fed into the trained clustering learning model to produce the cluster labels for the changed nodes. Note: for a given δ-PDG^{i,j}, the number of clusters is unknown to our model,

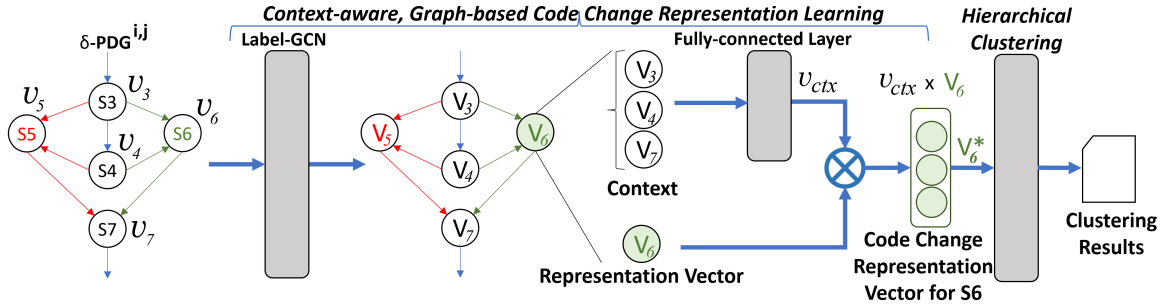


Figure 6: Context-aware, Graph-based, Code Change, Clustering Learning Model

and will be decided by the model during agglomerative clustering. Let us detail those two tasks of our clustering learning model.

4.1 Representation Learning for Code Changes

This task aims to build the vector representations (i.e., embeddings) for the code changes (i.e., changed statements) represented by the changed nodes in δ-PDG^{i,j}. A characteristic of the embeddings for code changes is *context-aware* or *contextualized*: the same code change in different contexts will have different embeddings.

4.1.1 Label, Graph Convolutional Network (Label-GCN). To achieve that, UTango first feeds the multi-version δ-PDG^{i,j} to a graph-based ML model to learn the contextualized embeddings for the nodes in the graph. Since δ-PDG^{i,j} contains the labels (i.e., i, j, (i, j)) representing the (un)changes, we use Label-GCN [5] to learn the graph structure and the node features with change labels.

Similar to GCN [13], Label-GCN [5] takes the graph with node features as input and produces the vectors for the nodes, when considering the features of the neighboring nodes of each node. In addition, for the current node, in the first layer, Label-GCN considers the change labels of the neighboring nodes as part of the feature vectors. It computes the vectors in the first layer as follows:

$$H^1 = \sigma[(\hat{A}X - diag(\hat{A}) \sum_{j=1}^{K} e_j e_j^T)W^0] \quad (1)$$

$$\hat{A} = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} \quad (2)$$

$$\tilde{A} = A + I \quad (3)$$

Where H is the output for the first hidden layer; A is the adjacency matrix; $\tilde{D}$ is the diagonal node degree matrix; W is the weight matrix; X is the input and $X \in R^{nx(d+K)}$; n is the number of nodes; d is the dimension of node features; K is the number of types of change labels in the input; $e_j e_j^T$ a single-entry matrix; and $-diag(\hat{A}) \sum_{j=1}^{K} e_j e_j^T$ is used to eliminates the self-loops for the components of the feature vectors corresponding to the labels.

In Label-GCN, the following layers after the first layer follow the same process as in the GCN model [13] to compute the hidden states. The computation in a following layer l is as follows:

$$H^{l+1} = \sigma(\hat{A}H^l W^l), l \geq 1 \quad (4)$$

4.1.2 Using Label-GCN to Model δ-PDG^{i,j}. Let us explain how we process the multi-version δ-PDG^{i,j} to produce the input for the Label-GCN model. For each node n of a statement s in δ-PDG^{i,j}, we break s down into the code tokens t, and build the vector e_t for t using a word embedding technique [23]. The vector for the node n is the average vector Avg_n of the vectors of all the tokens t within the corresponding statement s. Because each node n has a change label i, j, or (i, j) for the versions, we combine Avg_n with the one-hot vector of the length 3 representing the labels i, j, or (i, j). As a result, the combined vector v_n (with the length of $len(Avg_n)$+3) is the *node feature vector* for n. Then, we build the graph with the same structure as δ-PDG^{i,j} in which a node n is replaced with the node feature vector v_n, and feed that graph to the Label-GCN model to obtain the embeddings V_n for all the nodes n in δ-PDG^{i,j}.

4.1.3 Building Contexts and Contextualized Embeddings. After using Label-GCN, we obtain the vectors V_n for all the nodes n. For a changed node n_c, we collect the nodes in its context ctx (i.e., all un-changed nodes that are the k-hop neighbors of n together with all the inducing edges among them). We merge all the vectors for the nodes in ctx into a matrix accordingly to the order of the statements in source code. We then use a fully-connected layer to build the vector v_{ctx} representing the context ctx of the changed node n_c. Finally, we perform a cross-product between the context vector v_{ctx} and the vector V_{n_c} produced by Label-GCN for a changed node n_c, to build the contextualized embedding $V^*_{n_c}$ for n_c.

Figure 6 illustrates the process of building the contextualized embeddings for the code change example in Figure 4. S_3, ..., S_7 are the statements at the corresponding lines. After building δ-PDG^{i,j}, UTango goes through the process described in Section 4.1.2 (i.e., building token embeddings, statement embeddings, and combining with the labels) to produce the node feature vectors v_n for the nodes in δ-PDG^{i,j}. The Label-GCN model takes the graph with the vectors v_n to produce the graph with the same structure in which each node is represented by the vector V_n (V_3, ..., V_7). Let us use the node for V_6 as an example. The context of 1-hop neighbors includes V_3, V_4, and V_7. By merging those vectors as a matrix and passing through a fully connected layer, we obtain the vector v_{ctx} representing the 1-hop context for V_6. The cross-product vector $V^*_6 = v_{ctx} \times V_6$ is the contextualized embedding representing the changed statement S_6.

4.2 Learning to Cluster Code Changes

4.2.1 Supervised-Learning Hierarchical Agglomerative Clustering. After building the contextualized embeddings $V^*_{n_c}$ for all the changed nodes n_c in δ-PDG^{i,j}, UTango performs clustering on those vectors to untangle the commit. We modified the hierarchical agglomerative clustering algorithm [18] into a supervised-learning clustering model to cluster those vectors based on the clusters of code changes

in the training data. Specifically, the training process for our code change clustering learning model works as follows.

Step 1. We treat each changed node n_c as a separate cluster CL_c.

Step 2. We merge any two clusters whose cluster similarity is the largest and higher than a threshold T.

Step 3. We repeat the merging in Step 2 to form larger clusters until there is no cluster that can be merged. After this step, we obtain the predicted clusters CL at the current iteration.

Step 4. The predicted clusters CL at this iteration are compared against the correct clusters CL_{oracle} in the ground truth. We develop a loss function for our training to minimize the differences between the predicted clusters CL and the correct clusters CL_{oracle}. The parameters and the trainable threshold T will be updated accordingly to the loss function for the next iteration. The training will stop when the process converges and we obtain the most suitable parameters for our clustering learning model.

Next, let us explain the key components in our algorithm.

4.2.2 Cluster Similarity. To compute the similarity between two clusters CL_1 and CL_2, we take all the pairs of the changed nodes (n_1, n_2) where $n_1 \in CL_1$ and $n_2 \in CL_2$. We then compute the cosine similarity between the corresponding vectors $V^*_{n_1}$ and $V^*_{n_2}$ for each pair. The similarity between two clusters is calculated as the average of all the similarity scores of all the pairs (n_1, n_2).

4.2.3 Trainable Threshold T. In our model, we treat the merging threshold T between smaller clusters as a *trainable parameter*. T is updated after each iteration in accordance with the criteria defined in the loss function as any other parameters of the model.

4.2.4 Loss Function. We need to have a loss function that minimizes the differences between the predicted set of clusters CL=$\{CL_1, CL_2, ..., CL_M\}$ and the correct set $CL_{oracle} = \{Co_1, Co_2, ..., Co_N\}$ in the oracle. Because the predicted and correct sets might have different numbers of clusters ($M \neq N$), we first make them have the same size $\mathcal{M} = max(M, N)$ by adding the empty clusters to the smaller set between CL and CL_{oracle}. Because we do not know what predicted cluster CL_i in CL is mapped to a cluster Co_j in the correct set C_{oracle}, we consider all possible orders of the clusters in both CL and CL_{oracle}. The number of clusters is usually small (≤ 6 as reported in [21]), thus, it is manageable to consider all $\mathcal{M}!$ possible orders in CL and all $\mathcal{M}!$ possible orders in C_{oracle}.

Let us consider an order in $CL = \{CL'_1, ..., CL'_{\mathcal{M}}\}$, and an order in $C_{oracle} = \{Co'_1, ..., Co'_{\mathcal{M}}\}$. For a changed statement s_c with the corresponding changed node n_c in δ-PDG^{i,j}, we build the 1-hot vector X of the length $\mathcal{M}$ that represents the cluster for the node n_c predicted by the model as follows. If n_c is predicted to belong to a cluster CL'_i, the value at the i^{th} position of the vector X will be set to 1, otherwise it is set to 0. For example, if $\mathcal{M}$=4 and the changed node n_c is predicted to belong to the cluster CL'_3, the vector X is $\{0, 0, 1, 0\}$. Similarly, we build the 1-hot vector Y to represent the correct cluster for n_c in the oracle: if n_c belongs to a cluster Co'_i, the value at the i^{th} position of the vector Y is 1, otherwise it is 0.

For a specific order in CL=$\{CL'_1, ..., CL'_{\mathcal{M}}\}$, and a specific order in C_{oracle}=$\{Co'_1, ..., Co'_{\mathcal{M}}\}$, for a changed node n_c, we use the cross-entropy loss function in the multi-class classification: ($X = \{x_1, ..., x_{\mathcal{M}}\}$, Y=$\{y_1, ..., y_{\mathcal{M}}\}$ are the predicted and correct vectors for n_c)

$$Loss(X, Y) = -\sum_{i=1}^{\mathcal{M}} W_i log \frac{exp(x_i)}{exp(\sum_{j=1}^{\mathcal{M}} x_j)} y_i \quad (5)$$

To adjust Formula 5 for our clustering problem, we consider all possible orders in the cluster set CL and those in C_{oracle}. The cross-entropy loss function for a changed node n_c of a changed statement s_c is the minimum value among all the values on the right-hand side of Formula 5. That is, our loss function is computed as follows.

$$Loss'(X, Y) = \min_{\substack{all (\mathcal{M}!)^2 \\ orders}} (-\sum_{i=1}^{\mathcal{M}} W_i log \frac{exp(x_i)}{exp(\sum_{j=1}^{\mathcal{M}} x_j)} y_i) \quad (6)$$

The loss function for each changed node n_c will be used for the model to adjust the parameters in the next iteration.

4.2.5 Predicting/Clustering Process. After training, we obtain all model's parameters and the trainable merging threshold. For clustering, UTANGO takes a δ-PDG^{i,j} as input and generates the resulting set of clusters CL.

5 UPDATING CLUSTERS VIA CODE CLONE DETECTION

Because δ-PDG^{i,j} might not cover the changed statements that are clones of one another, we use a code clone detection tool [25] to find the changes that might belong to the same concerns but having no data/control dependencies with one another. The clone detection tool returns a list of clone candidates in the clone groups with different sizes. First, we consider only the clone groups with the cloned fragments containing the changed statements. If multiple clone fragments contain the same changed statement, we choose the clone fragment with the largest size to avoid the duplication (because the larger clone contains the smaller one).

After predicting/clustering the changed nodes/statements as explained in Section 4, we have marked each changed statement with a concern. For each changed statement s_c in a clone group, we check the clusters of its cloned statements and update the cluster for s_c via majority voting. For example, a changed statement s_c marked with concern 1, and it belongs to a cloned fragment in which the other statements in that fragment and their clones are marked with concern 2 in their majority, then we change s_c to be about concern 2. If there is no majority, we adjust the cluster of s_c to the concern/cluster with a lower index for consistency.

6 EMPIRICAL EVALUATION

6.1 Research Questions

To evaluate UTANGO, we seek to answer the following questions:

RQ1. Comparative Study on an C# Dataset. How well does UTANGO perform in comparison with the state-of-the-art commit-untangling approaches on an C# dataset?

RQ2. Comparative Study on a Java Dataset. How well does UTANGO perform in comparison with the state-of-the-art commit-untangling approaches on a Java dataset?

RQ3. Within-Project Analysis. How well does UTANGO perform when training and testing are done in the same project?

RQ4. Sensitivity Analysis. How do the key features in UTANGO affect its overall performance?

RQ5. Code Change Embedding Analysis. What are the properties of the contextualized embeddings from our context-aware, graph-based representation learning model for code changes? The answer will show the contribution of code change embeddings.

6.2 Datasets

We have conducted our evaluation on two datasets. The first one is a C# dataset that has been used in the commit-untangling work, Flexeme [22]. This C# dataset contains 1,612 tangled commits (each has ≤ 3 concerns) in 9 Github projects. We use this C# dataset for all RQs except RQ2. In RQ2, to evaluate UTANGO on Java and compare against the state-of-the-art SmartCommit [24] (working on Java only), we collected a new dataset by following the same process in that paper. The Java dataset contains +14K tangled commits in 10 GitHub projects. The number of concerns in a commit is from 2–32.

6.3 Experimental Methodology

6.3.1 RQ1. Comparison with Existing Approaches on C# Dataset. *Baselines.* We compared UTANGO against the state-of-the-art commit-untangling approaches that work on C# (see more details in Section 8): Barnett *et al.* [4], Herzig *et al.* [10], Flexeme [22], and δ-PDG+CV [22] (δ-PDG+CV is a model introduced by the authors of Flexeme via combining their δ-PDG with confidence voting in Herzig *et al.* [10]).

Procedure. We took all the commits in the C# dataset and sorted them in the chronological order based on the creation time of the commit logs. For UTANGO, we then used 80% of the oldest commits for training, 10% of the next oldest ones for tuning, and the latest 10% of the commits for testing. For the baselines, all of them do not need training, thus, we ran them on the 10% testing data. We did not use the setting of leave-one-out by project (testing on part of a project and training on other 8 projects). The reason is that after sorting in a chronological order, we could not test on the oldest project due to no training data. Even with the 2nd-5th oldest ones, there is no sufficient training data of the commits that came from the older projects and before the test commits in those projects.

We tuned UTANGO with autoML [1] for the following key hyper-parameters to have the best performance: (1) Epoch size (100, 200, 300); (2) Batch size (64, 128, 256); (3) Learning rate (0.001, 0.003, 0.005, 0.010); (4) Vector length of word embeddings and its output (150, 200, 250, 300). We empirically set the context size of 2 (RQ4).

6.3.2 RQ2. Comparison with Existing Approaches on Java Dataset. *Baselines.* We compare UTANGO with the state-of-the-art untangling approach that works on Java source code, SmartCommit [24], and the baselines used in their paper: Base-1 [24] (a rule-based approach that puts all changes into one group), Base-2 [24] (a rule-based approach that puts the changes in each file into one group), and Base-3 [24] (a rule-based approach that considers only def-use, use-use and same-enclosing-method relations).

Procedure. We used the same procedure, tuning (with autoML [1]), and parameters as in RQ1 on the Java dataset. The baselines do not need training, thus, we ran them on the 10% testing data.

6.3.3 RQ3. Within-Project Analysis. We randomly selected one of the largest projects from each dataset: the `CommandLine` project in C# and the `netty` project in Java. For each project, we sorted all of the commits in the chronological order based on commit time. We split the commits into 80%, 10%, 10% for training, tuning, and testing, and using the older data for training and the newer data for tuning and testing. We separately trained, fine-tuned, and tested a model under study on the commits in each project.

6.3.4 RQ4. Sensitivity Analysis. We built the variants of UTANGO by removing its important components, one at a time. First, we removed from UTANGO the context by not using the context vector when computing the vector for a changed statement. Second, we removed from UTANGO the clone detection component and used the resulting clusters from the code change clustering learning model as the final results. Third, we studied the impact of the size K of a context on the accuracy. We also studied the impact of the data size. We used the C# dataset and the same setting/procedure as in RQ1.

6.3.5 RQ5. Code Change Embedding Analysis. We use statistical p-test to confirm/refute the hypothesis that the changed statements in the same concerns are projected nearer to one another than the changed ones in different concerns in the vector space.

Evaluation Metrics. We use two evaluation metrics. First, $Accuracy^c$ is defined as the percentage of the *changed statements* that are labeled with a correct cluster/concern in all the statements in a commit: $\textbf{Accuracy}^{\textbf{c}} = \frac{\#changed\ stmts\ w.\ correct\ clusters}{\#all\ changed\ stmts\ in\ commit}$. Note that a model might label the statements with a different permutation of cluster labels than the one in the ground truth. For example, we have five statements and three concerns. A model can predict [3,1,1,2,2] and the ground truth has [1,2,2,3,3]. A naive evaluation would give an accuracy of 0.0. However, a permutation of the labels for clusters would give indeed an accuracy of 1.0. Thus, we use the Hungarian Algorithm [16] to find the permutation that gives the maximum accuracy, and use that for $Accuracy^c$.

In RQ1, for the comparison with Flexeme [22], we also used another metric that was used in their paper, which is similar to $Accuracy^c$ except that it considers all statements: $\textbf{Accuracy}^{\textbf{a}} = \frac{\#stmts\ w.\ correct\ clusters}{\#all\ stmts\ in\ commit}$. We also consider all label permutations.

7 EXPERIMENTAL RESULTS

7.1 RQ1. Comparative Study on C# Dataset

7.1.1 General Results. Table 1 shows the comparison on $Accuracy^c$ when it was measured on the changed statements. As seen, UTANGO improves Barnett *et al.*, Herzig *et al.*, σ–PDG+CV, and Flexeme in overall $Accuracy^c$ by **462.5%, 55.2%, 28.6%**, and **36.4%**, respectively.

Table 2 shows the results on $Accuracy^a$. When $Accuracy^a$ was measured on all the statements (changed and un-changed) in a commit, the results for all models are higher because they have correct classifications for the changed statements by default. We include $Accuracy^a$ for the comparison purpose with Flexeme (as its authors used this metric in their paper). As seen, UTANGO also improves Barnett *et al.*, Herzig *et al.*, σ–PDG+CV, and Flexeme by **584.6%, 32.8%, 7.2%,** and **9.9%** in overall $Accuracy^a$, respectively. UTANGO's accuracies are consistently better than those of the baselines for all the projects with respect to different numbers of concerns in a commit (#*C*s=2 or 3, Table 1). Some data points are unavailable since those commits are older in the chronological order and appear only in the training data, but not in the testing data.

Table 1: RQ1. Comparison on C# Dataset (*Accuracy*^{c}%)

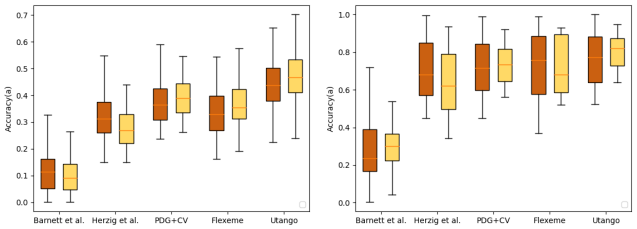
	Barnett *et al.*			Herzig *et al.*			δ–PDG + CV			Flexeme			UTango		
#Cs	2	3	OA	2	3	OA	2	3	OA	2	3	OA	2	3	OA
CL	14	*	14	28	*	28	34	*	34	34	*	34	46	*	46
CM	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
HF	10	13	11	27	29	28	34	37	35	30	35	31	43	48	45
HU	13	*	13	27	*	27	30	*	30	33	*	33	44	*	44
LE	8	6	8	29	24	29	35	34	35	33	36	33	44	47	45
NA	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
NJ	7	*	7	28	*	28	34	*	34	27	*	27	41	*	41
NI	10	*	10	26	*	26	37	*	37	32	*	32	46	*	46
RS	9	11	9	31	30	31	30	36	31	32	35	33	42	49	43
OA	8	6	8	31	25	29	35	34	35	32	36	33	**44**	**47**	**45**

CL:Commandline, CM:CommonMark, HF:Hangfire, HU:Humanizer, LE:Lean, NA:Nan cy, NJ:Newtonsoft.Json, NI:Ninject, RS:RestSharp, OA: Overall, *: No avail data point.

Table 2: RQ1. Comparison on C# Dataset (*Accuracy*^{a}%)

	Barnett *et al.*			Herzig *et al.*			δ–PDG + CV			Flexeme			UTango		
#Cs	2	3	OA	2	3	OA	2	3	OA	2	3	OA	2	3	OA
CL	18	21	19	67	48	64	77	84	80	82	92	82	90	*	90
CM	20	*	20	65	*	65	90	*	90	70	*	70	*	*	*
HF	16	13	15	70	54	64	84	88	87	86	68	79	84	91	86
HU	18	31	18	64	42	62	69	*	69	83	57	81	89	*	89
LE	19	12	18	69	62	69	84	71	84	77	82	80	87	90	88
NA	9	8	9	70	56	67	86	80	86	81	92	84	*	*	*
NJ	15	11	15	71	56	71	86	69	82	71	52	71	83	*	83
NI	14	*	14	57	*	57	94	*	94	80	*	80	91	*	91
RS	12	14	12	71	69	70	74	53	70	82	89	82	87	88	87
OA	14	11	13	69	62	67	83	84	83	81	84	81	**88**	**91**	**89**

CL:Commandline, CM:CommonMark, HF:Hangfire, HU:Humanizer, LE:Lean, NA:Nan cy, NJ:Newtonsoft.Json, NI:Ninject, RS:RestSharp, OA: Overall, *: No avail data point.



(a) *Accuracy*^{c} **(b)** *Accuracy*^{a}

Figure 7: Boxplots for Results in Table 1 and Table 2. Orange Boxes: 2-concern data, Yellow Boxes: 3-concern data

Figure 7 shows the boxplots for both *Accuracy*^{c} and *Accuracy*^{a} results in Tables 1 and 2. The orange boxes are for the commits with two concerns, while the yellow ones are for those with three concerns. As seen, the median accuracy of UTango is higher than those of all baselines on both types of two and three concerns. Moreover, the gap (in %) between UTango and the baselines in *Accuracy*^{c} is larger than the gap between them in *Accuracy*^{a} because all models have correct classifications by default for the changed statements, which are many more than the changed statements.

7.1.2 Accuracy Results for Concerns with Different Sizes. To better understand UTango's accuracy on concerns with different sizes,

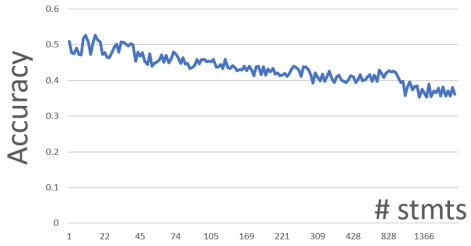


Figure 8: Accuracies for Concerns with Different Numbers of Statements in C# dataset

Table 3: RQ2. Comparison on Java Dataset (*Accuracy*^{c}%)

	Base-1	Base-2	Base-3	SmartCommit	UTango
SB	14	23	21	29	32
ES	19	22	27	34	36
RJ	18	28	24	31	33
GU	21	31	25	33	37
RE	22	24	19	29	34
DU	15	19	18	24	29
GH	22	27	24	33	35
ZX	21	29	20	34	38
DR	17	29	18	32	34
EB	13	22	21	27	31
OA	17	25	23	30	**34**

SB: spring-boot, ES: elasticsearch, RJ: RxJava, GU:guava, RE: retrofit, DU: dubbo, GH: ghidra, ZX: zxing, DR: druid, EB: EventBus, OA: Overall

we collected all concerns with different numbers of changed statements and measured the accuracies for them. As seen in Figure 8, *Accuracy*^{c} values for the concerns of small sizes (having 1–22 changed statements) are higher (ranging from 47%–54%). That is, among all the changed statements, UTango correctly classified about 50% of them into the correct clusters. For the larger concerns, *Accuracy*^{c} decreases as expected because it is more challenging to get correct classifications for more changed statements in a concern. However, *Accuracy*^{c} decreases gradually with the lowest value of $\approx$ 36%. Note: there are a few commits with the addition of large files of +1.3k lines.

Despite challenges with large concerns, UTango correctly classified *100% of all the changed statements* for 15 commits with up to 19 statements. Flexeme fails to reach 100% for those commits. There are only 9 commits that Flexeme correctly classified all the changed statements and UTango did not reach 100%. Both models correctly classified 100% of all the changed statements of 4 commits.

Training of UTango on 80% of C# dataset takes 1.75 hours and predicting on 10% of C# dataset takes 1.3-2.5 seconds per commit.

7.2 RQ2. Comparative Study on Java Dataset

7.2.1 General Results. Table 3 shows *Accuracy*^{c} when it was measured on the changed statements for Java code. As seen, on the Java dataset, UTango improves Base-1, Base-2, Base-3, and SmartCommit in overall *Accuracy*^{c} by **100.0%, 36.0%, 47.8%**, and **13.3%**, respectively. UTango's accuracies are consistently better than those of the baselines for all subject projects. Figure 9 shows the boxplot for *Accuracy*^{c} result in Table 3. As seen, the median accuracy of UTango is higher than those of the baselines.

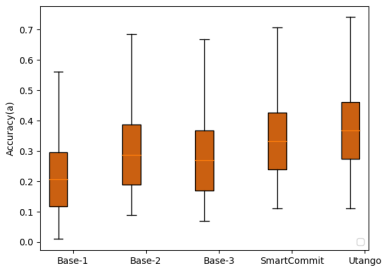


Figure 9: Boxplots for Results in Table 3

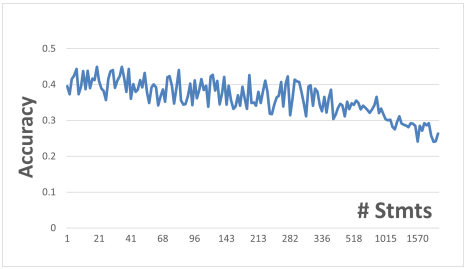


Figure 10: Accuracies for Concerns with Different Numbers of Statements in Java dataset

7.2.2 Accuracy Results for Concerns with Different Sizes. As seen in Figure 10, $Accuracy^c$ values for the concerns of smaller sizes (having 1–22 changed statements) are higher (ranging from 37%–45%). That is, among all the changed statements, UTango correctly classified 41% of them into the correct clusters. For larger concerns, $Accuracy^c$ decreases gradually. Even in the concerns with up to 336 changed statements, $Accuracy^c$ is >=30%. The lowest accuracy is 24%. Moreover, it correctly classified *100% of all the changed statements* for 95 commits in which SmartCommit failed to reach 100%. There are 88 commits that SmartCommit correctly classified all the changed statements and UTango did not reach 100%. Both models correctly classified 100% of all changed statements of 21 commits.

Training of UTango on 80% of the Java dataset takes 17 hours and predicting on 10% of Java dataset takes 1.5-4.2 seconds per commit. Note: the Java dataset is much larger than the C# dataset.

7.3 RQ3. Within-Project Analysis

Table 4 shows UTango's results in the within and cross-project settings. As seen, with sufficient within-project data, $Accuracy^c$ in the within-project setting is slightly better than that of the cross-project setting for both C# and Java projects. This is expected since the changes in the same project could be more similar than the changes across projects. Thus, UTango works well in both within- and cross-project settings for C# and Java projects.

7.4 RQ4. Sensitivity Analysis

Table 5 shows the accuracy results when the key components in UTango were removed. As seen, while both context and clone detection positively contribute to UTango's accuracy, the context plays a more important role as expected. Without the context vector,

Table 4: RQ3. Results for Within- and Cross-project Settings

Project	Within-Project	Cross-Project
CommandLine (C#)	0.47	0.46
spring-boot (Java)	0.34	0.32

Table 5: RQ4. Impacts of Key Features on Accuracy

	$Accuracy^c$		
# concerns	2	3	Overall
UTango w/o Context	0.38	0.43	0.40
UTango w/o Clone Detection	0.42	0.44	0.43
UTango	0.44	0.47	0.45

Table 6: RQ4. Impact of the Number k of Hops for Context

	$Accuracy^c$		
#concerns	2	3	Overall
UTango ($k = 1$)	0.42	0.45	0.43
UTango ($k = 2$)	0.44	0.47	0.45
UTango ($k = 3$)	0.43	0.45	0.44
UTango ($k = 4$)	0.40	0.44	0.42
UTango ($k = 5$)	0.41	0.44	0.42
UTango (Full Graph)	0.39	0.43	0.41

Table 7: Impact of the Size of Training Data

Splitting on C# dataset	80%/10%/10%	70%/15%/15%	60%/20%/20%
% $Accuracy^c$	45%	42%	37%

the accuracy decreases by 13.7%, 8.5%, and 11.1% for the commits with two, three, and all concerns, respectively. Without the clone detection, the accuracy decreases by 4.5%, 6.4%, and 4.4% for the commits with two, three, and all concerns, respectively.

Table 6 shows the accuracy when we vary the size k of a context (i.e., the number of hops from the changed node). As seen, when the number of surrounding nodes of the changed node increases from k=1–2, the accuracy increases and reaches its peak at k=2. As k=1, the immediate surrounding nodes cannot capture well the relevant nodes for UTango to distinguish the concerns of the changed statements. With two hops from the changed node, the context seems to sufficiently contain the crucial nodes w.r.t. determining the concerns. However, as the size continues to increase, the accuracy decreases. When the entire graph is used as the context, the accuracy is the lowest. If more nodes are considered in the context, more noises are added as more irrelevant data is used. The trend is the same for the commits with two, three, or all concerns.

As seen in Table 7, the training data's size has impact on accuracy. The more training data, the higher the accuracy. Even reducing it from 80% to 60%, UTango's accuracy is still higher than Flexeme's.

7.5 RQ5. Analysis on Code Change Embeddings

To study the projections of the changed statements in the same and different concerns, we first randomly chose the commits with two or more clusters/concerns and in one of the cluster/concern, there are at least two changed statements. Let us use C to denote that cluster/concern. We randomly chose two changed statements S_1 and S_2 in C. We then randomly selected another changed statement S_3 such that $S_3 \notin C$ and S_3 belongs to another cluster in the same commit with S_1 and S_2. We measured the distance $d_1(S_1, S_2)$

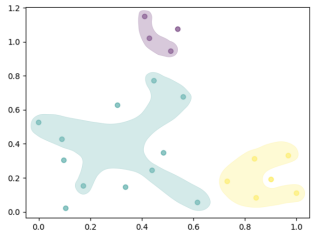


Figure 11: Code Change Embedding Visualization via t-SNE

and $d_2(S_1, S_3)$. We repeated the process for all the commits satisfying the above conditions to get 384 triples of (S_1, S_2, S_3). Based on the population in our dataset, the size of 384 samples gives the confidence level of 95% and the confidence interval of 5%. We used statistical p-value to confirm our hypothesis $H_1 : d_1(S_1, S_2) \leq d_2(S_1, S_3)$. The null-hypothesis is $H_0 : d_1(S_1, S_2) > d_2(S_1, S_3)$.

When we set the significance level $\alpha = 0.05$, the p-value is 0.03 (calculated on these 384 samples). In this case, the p-value is smaller than α, meaning the null hypothesis would be rejected at the level of $\alpha = 0.05$. Thus, our hypothesis H_1: $d_1(S_1, S_2) \leq d_2(S_1, S_3)$ is confirmed. That is, the changed statements in the same concerns are projected nearer to one another than the changed statements in different concerns. This result is an indication that our *context-aware, graph-based embeddings for code changes are helpful for* UTango *in clustering the code changes via the embeddings into different concerns.*

For illustration, we randomly picked a commit in our above sample. The commit contains three concerns in which one concern has 4 changed statements, another one has 12 changed statements, and the last one has 6 changed statements. We used the t-distributed stochastic neighbor embedding (t-SNE) [2] to visualize the embeddings of the changed statements in the vector space. t-SNE is a statistical method for visualizing high-dimensional data by giving each data point its projected location in a two-dimensional vector space. As seen in Figure 11, we marked the embeddings for the changed statements in each concern with a different color. The correctly clustered changes are within the corresponding boundary for a concern. Despite that UTango misclassified a few statements, we can observe that the changes in each concern are projected to the nearby locations in the vector space. This figure illustrates that the our code change embeddings facilitate our supervised-learning clustering model to correctly cluster the changed statements.

7.6 Discussions

7.6.1 Threats to Validity. Our study is only on C# and Java projects. Our methodology is language-independent. The datasets of tangle commits were built artificially from atomic commits. However, this methodology was used in Flexeme [22] and Herzig *et al.* [11]. Despite that we did not use the setting of leave-one-out by project, our setting reflects the actual use of our tool in which all the commits in other projects that occurred before the current commit are used.

7.6.2 Limitations. UTango is less accurate for the commits with too many changed statements (>= 40), or with too many concerns (>=20). As with any data-driven approaches, it does not work well with little data. We currently integrate only the code clone relations. We will explore other types of implicit relations, e.g., in event-driven source code, or other links as in SmartCommit.

Trade-offs between UTango and program-analysis approaches: without needing a threshold for clustering, UTango is more flexible in learning the boundaries among clusters than PA approaches, which rely on explicit program dependencies for clustering. Despite needing no training data, Flexeme requires a median time of 9.56 seconds to untangle a commit. UTango takes only 1.3-2.5 seconds.

8 RELATED WORK

Tangled commits have been reported by researchers to have negative impacts on software maintenance [10, 11, 19, 21, 22, 24, 26].

Mining Software Repositories. Herzig *et al.* [10, 11] combine confidence voting with agglomerative clustering. Each confidence voter is responsible for an important aspect, e.g., call-graphs, change couplings, data dependencies, and distance measures. In Kirinuki *et al.*'s [14, 15], if there is a commit m including the same changes as a past commit and other changes, the commit m is called inclusive change and considered as tangled. Dias *et al.* [6] uses confidence voting on fine-grained change events in an IDE and partition them.

Static Analysis. Roover *et al.*'s approach [20] builds PDG and computes the changes to the ASTs of the files in a commit. It then groups these fine-grained changes according to the slices through the PDG they belong to. ClusterChanges [4] relates separate regions of change within a changeset of a commit by using static analysis to uncover relationships such as definitions and their uses present in these regions. UTango adapts multi-version PDG from Flexeme [22], however, we build the contextualized embeddings for the code changes and a model to learn to cluster, rather than clustering using graph similarity on multi-version PFG. SmartCommit [24] uses a graph partition algorithm on code changes related via several types of links, representing different purposes.

There are a rich literature on supervised hierarchical clustering [8, 9, 12, 17, 27, 28]. The dissimilarity between cluster pairs is measured via a linkage function F [9, 28]. Learning F is performed by training the pairwise dissimilarity function to predict dissimilarity for all within- and across-cluster data pairs [12, 28]. In UTango, supervised-learning clustering is made with the loss function.

9 CONCLUSION

We present **UTango**, a ML-based approach that learns to untangle the changes in a commit. We develop a *novel code change clustering learning model* that learns to cluster the code changes, represented by the embeddings, into different groups with different concerns. UTango overcomes the key issue with the static analysis approaches in which the boundaries across concerns in a commit do not naturally map to clustering criteria. Our ML direction fits well with this problem thanks to a methodology from Herzig *et al.* [11] and Flexeme [22], to collect the changes in the same concern and merge them to build tangled commits to form a training dataset.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grants CNS-2120386, CCF-1723215, CCF-1723432, TWC-1723198, and the US National Security Agency (NSA) grant NCAE-C-002-2021 on Cybersecurity Research Innovation.

REFERENCES

[1] 2021. *The NNI autoML tool.* https://github.com/microsoft/nni

[2] 2022. *T-SNE.* https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html

[3] 2022. UTango. https://github.com/Commit-Untangling/commit-untangling

[4] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. 2015. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 134–144.

[5] Claudio Bellei, Hussain Alattas, and Nesrine Kaaniche. 2021. Label-GCN: An Effective Method for Adding Label Propagation to Graph Convolutional Networks. *CoRR* abs/2104.02153 (2021). arXiv:2104.02153 https://arxiv.org/abs/2104.02153

[6] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 341–350. https://doi.org/10.1109/SANER.2015.7081844

[7] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041

[8] Thomas Finley and Thorsten Joachims. 2005. Supervised Clustering with Support Vector Machines. In *Proceedings of the 22nd International Conference on Machine Learning* (Bonn, Germany) *(ICML '05)*. Association for Computing Machinery, New York, NY, USA, 217–224. https://doi.org/10.1145/1102351.1102379

[9] Anupam Guha, Mohit Iyyer, Danny Bouman, and Jordan L. Boyd-Graber. 2015. Removing the Training Wheels: A Coreference Dataset that Entertains Humans and Challenges Computers. In *HLT-NAACL*. 1108–1118. http://aclweb.org/anthology/N/N15/N15-1117.pdf

[10] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The Impact of Tangled Code Changes on Defect Prediction Models. *Empirical Softw. Engg.* 21, 2 (apr 2016), 303–336. https://doi.org/10.1007/s10664-015-9376-6

[11] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (San Francisco, CA, USA) *(MSR '13)*. IEEE Press, 121–130.

[12] Kian Kenyon-Dean, Jackie Chi Kit Cheung, and Doina Precup. 2018. Resolving Event Coreference with Supervised Representation Learning and Clustering-Oriented Regularization. In *Proceedings of the Seventh Joint Conference on Lexical and Computational Semantics*. Association for Computational Linguistics, New Orleans, Louisiana, 1–10. https://doi.org/10.18653/v1/S18-2001

[13] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[14] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2014. Hey! Are You Committing Tangled Changes?. In *Proceedings of the 22nd International Conference on Program Comprehension* (Hyderabad, India) *(ICPC 2014)*. Association for Computing Machinery, New York, NY, USA, 262–265. https://doi.org/10.1145/2597008.2597798

[15] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2016. Splitting Commits via Past Code Changes. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 129–136. https://doi.org/10.1109/APSEC.2016.028

[16] H. W. Kuhn. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly* 2, 1-2 (1955), 83–97.

[17] Yang Liu, Jing Liu, Zechao Li, Jinhui Tang, and Hanqing Lu. 2013. Weakly-Supervised Dual Clustering for Image Semantic Segmentation. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*. 2075–2082. https://doi.org/10.1109/CVPR.2013.270

[18] Daniel Müllner. 2011. Modern hierarchical, agglomerative clustering algorithms. arXiv:1109.2378 [stat.ML]

[19] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (2012), 5–18. https://doi.org/10.1109/TSE.2011.41

[20] Ward Muylaert and Coen De Roover. 2018. [Research Paper] Untangling Composite Commits Using Program Slicing. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 193–202. https://doi.org/10.1109/SCAM.2018.00030

[21] Hoan Anh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2013. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. 138–147. https://doi.org/10.1109/ISSRE.2013.6698913

[22] Profir-Petru Pârundefinedachi, Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2020. Flexeme: Untangling Commits Using Lexical Flows. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/3368089.3409693

[23] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. http://www.aclweb.org/anthology/D14-1162

[24] Bo Shen, Wei Zhang, Christian Kästner, Haiyan Zhao, Zhao Wei, Guangtai Liang, and Zhi Jin. 2021. SmartCommit: A Graph-Based Interactive Assistant for Activity-Oriented Commits. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 379–390. https://doi.org/10.1145/3468264.3468551

[25] Jeffrey Svajlenko and Chanchal Kumar Roy. 2017. Fast and flexible large-scale clone detection with CloneWorks.. In *ICSE (Companion Volume)*. 27–30.

[26] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How Do Software Engineers Understand Code Changes? An Exploratory Study in Industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 51, 11 pages. https://doi.org/10.1145/2393596.2393656

[27] Jun Tie, Wenying Chen, Chong Sun, Tengyue Mao, and Guanglin Xing. 2019. The application of agglomerative hierarchical spatial clustering algorithm in tea blending. *Cluster Computing* 22 (05 2019). https://doi.org/10.1007/s10586-018-1813-z

[28] Nishant Yadav, Ari Kobren, Nicholas Monath, and Andrew Mccallum. 2019. Supervised Hierarchical Clustering with Exponential Linkage. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 6973–6983. https://proceedings.mlr.press/v97/yadav19a.html