社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

# 変更個所の構造的特徴の学習に基づく複合コミットの分割

眞田　行隆†　　小林　隆志†

† 東京工業大学 情報理工学院 情報工学系　〒 152–8552 東京都目黒区大岡山 2–12–1
E-mail: sanada@sa.cs.titech.ac.jp, tkobaya@c.titech.ac.jp

**あらまし**　バグ修正とリファクタリングを並行して行うなど，複数のタスクの結果として生まれたコミットは複合コミットと呼ばれ，後にレビューを行う際や過去の状態に戻す際に問題を複雑化させる．この問題に対処する為に，自動で複合コミットを分割する様々な研究が行われている．しかしながら，既存研究は経験則によって選択された特徴量に基づくものであり，複雑な特徴を認識しきれない問題がある．本研究では，過去の変更における変更個所の構造的特徴を分析し機械学習することで，チャンク間の関係を距離として算出し，これに基づいてクラスタリングを行うことでコミットを分割する手法を提案する．5 つの OSS プロジェクトのデータに提案手法を適用し，チャンク同士の結合・分離の推定精度およびコミット分割の精度を計測し先行研究と比較を行う．

**キーワード**　コミット分割，Tangled Commit，Change Partitioning，改版履歴分析，機械学習

## 1. はじめに

ソフトウェアは継続的な修正や機能追加によって進化する．チームの共同開発における変更管理のために，ソフトウェアの進化は，ソースコード変更の履歴として版管理システムによって保存される．これらの履歴はプログラム理解の際に参照されるほか，類似プロジェクトへのパッチ作成などに用いられて来た．近年では，履歴を解析・学習することでバグ予測 [1] や変更漏れの通知 [2] などを行う手法が多く提案されている．

版管理システムにおける管理単位であるコミットは，バグ修正やリファクタリングといった作業単位 (タスク) ごとに構成されるべき [3] とされており，このようなコミットを**タスクレベルコミット** (以降 TLC) と呼ぶ．しかしながら，実際の開発では複数タスクの作業結果をまとめてコミットする**複合コミット** (以降 CC) が行われることも多い．CC は Tangled Change，Mixed-purpose commit とも呼ばれ，変更履歴の理解や変更内容の共有をするうえで悪影響を与えることが知られている [4]．例えば，変更内容のレビューの際や変更を取り消したい際に CC が存在していると，タスクと無関係な変更や取り消す対象でない変更が含まれているため問題を複雑化させてしまう．また，変更履歴を用いた支援手法についても悪影響があることが知られている [5]．これまでの研究で CC はプロジェクトの規模の大小に関係なく起こりうることが知られており [4]，Tao らの調査では対象の 4 つのプロジェクトは最大で 29%のバグ修正コミットに無関係の修正を含んでいたと報告されている [6]．

この問題に対処するために，自動で CC を分割する手法が提案されてきた．Kawrykow は自動で CC を検知し各 TLC へ行単位で分割する手法を提案した [7]．また，Herzig らが提案した手法 [4] では，追加行または削除行のまとまり（以降チャンクと表記）を単位としてその関係性を推定し，それらをクラスタリングを行うことでコミットを分割する．

様々な分割手法が提案されてきたが，いずれも経験則によって選択された関係性の強度や特徴量を線形結合して利用している．このため，複雑な特徴を認識しきれない可能性がある．機能追加，バグ修正，リファクタリングなど，コミットの意図は様々であり，含まれる変更の特徴はその意図ごとに異なる．様々な意図が含まれたコミットにおいても，変更部分ごとの意図を認識，分類することでさらなる精度向上が期待できる．

近年では，ソースコードから様々な特徴を抽出しニューラルネットワークを用いて分散表現を生成する研究が盛んに行われている [8]．例えば，code2vec [9] は，メソッド定義とメソッド名の特徴を学習し，コードからメソッド名を予測するモデルであり，既存研究を大きく上回る約 63%という精度を達成している．また，ASTNN [10] は，AST ベースの Embedding でソースコードの特徴を学習し，コード分類やコードクローン検知の領域で 90%を超える精度を達成しいる．しかしながら，コミット分割においては，機械学習を使用した前例は存在するが [4, 7, 11]，我々の知る限りニューラルネットワークを用いた例は未だ存在しない．

本研究では，チャンク間の複雑な関係をニューラルネットワークによって学習・推定することで，CC を自動分割する手法を提案する．Self Attention Network (SAN) [12] を用いることで，特徴とその重要性を同時に学習し，経験則によらないチャンク間関係の推定を行う．提案手法では，チャンクに含まれるソースコードの抽象構文木 (AST) の情報を利用する．コミットに含まれるチャンク対の全組み合わせを生成し，それぞれのチャンクで変更された AST ノード列を入力として SAN によって関係性を学習する．チャンクには，追加・削除といった変更種別と，変更された位置に応じて異なる特徴があることに着目し，変更種別と変更位置によってチャンク対を分類し，それぞれを別々に学習する．既知の TLC と CC を用いて学習したモデルを利用することで，コミットに含まれるチャンク間の関係を推定し，その関係性をもちいてクラスタリングすることでコミットを分割する．

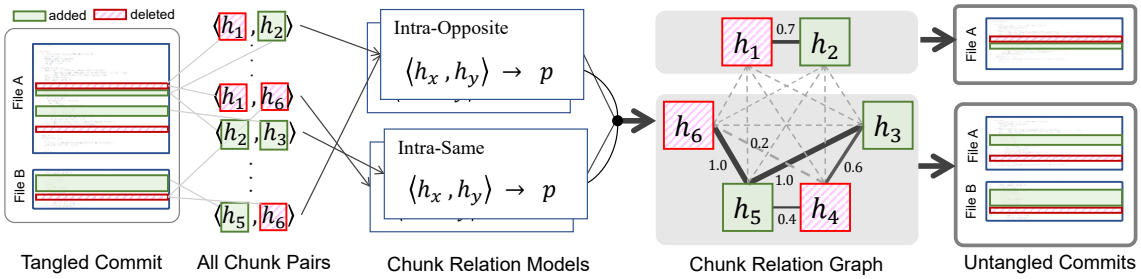


図 1　提案手法の全体概要図

本研究では，提案手法の有用性を評価するために本手法を実装し，Herzig らの研究 [4] と同じデータセットを用いて実験を行った．実験の結果，チャンク間の関係に対する推定精度は 67%であり，この関係推定をもとにコミット分割を行った際の精度は 74%と経験則に基づく手法と同等程度であった．

## 2. 関 連 研 究

自動コミット分割の代表的な手法は，Herzig らのチャンク単位でのコミット分割 [4, 13] である．この手法では，コミット内のチャンクのペアを作り，それらが同時にコミットされるべきか別々にコミットされるべきかを判定する．チャンク対の判定では，経験則に基づいて選択された複数のチャンク間関係を算出し，それらを集約することでチャンクペアの結合度を推定する．結合度をもとにチャンク間関係グラフを構築しマルチレベルグラフ分割 [14] を応用してコミットを分割する．チャンク間関係には，同一ファイル内での距離，パッケージ距離，変更チャンクによって変更されたメソッド同士のコールグラフ内での距離， Change Coupling [15] の確信度，同じ変数へのアクセスの有無の 5 種類が定義されており，線形回帰によって最適な集約方法を学習する [4].

また，Herzig らは人工的に CC を生成する手法も提案しており，2 つの TLC から生成された人工的な CC を分割する実験において，約 73%のチャンクが正解のコミットへと分類できることを示している．

Tao らは，コードレビューの改善を目的として，行単位で分割するコミット分割手法を提案している [6]．彼らの手法では，変更行間の関連性として，フォーマットのみの変更，依存関係の存在，類似した変更，の 3 種類を定義し，Change Distiller [16] とプログラムスライシングを用いてこれらの関連性を検出する．4 つの OSS リポジトリから抽出した 78 の CC に適用した結果，約 69%の CC を正確に分割できたことが報告されている．

Muylaert らの研究 [17] では，プログラムスライシングを用いて AST レベルでのコミット分割を行っている．Herzig らのデータセットの一部に対して適用し，プログラムスライシングを用いる方法は，コミットを細かく分割する傾向があることを明らかにしている．

自動コミット分割の手法には，コミットのみを分析するのではなく，開発者の IDE 操作履歴を解析して支援を行うものも提案されている．Dias らは，プログラム変更時にリアルタイムで分割を支援する手法を提案している [11]．提案手法では，コミットよりも細かい編集や保存のタイミングで得られる 6 つの特徴量を機械学習して CC を分割する．3 種類の分類器について性能を比較し，Random Forest が最も優れており，90%を超える精度での分割が可能であることを示している．

## 3. 提案手法: SAN によるチャンク間関係学習に基づくコミット分割

図 1 に提案手法の概要を示す．提案手法は，Herzig らの研究 [4] と同様に，コミットをチャンク単位で分割し，全てのチャンク対に対して関係を推定することでチャンク関係グラフを構成する．このグラフを関係の強度に応じて分割することで CC を TLC に分割する．複数のプロジェクトの改版履歴から抽出したチャンク対の関係を学習することで，プロジェクトを横断した汎用的なチャンク間関係を学習する．

チャンク間関係を推定する際に，チャンク対を 4 つに分類し，分類ごとに SAN を用いたチャンク関係モデルを構築することで，経験則によらない特徴発見と学習を実現する．

### 3.1 チャンク対の生成と分類

コミットが含有する全てのチャンクについて，変更位置 (ファイル，行数) と変更の種別 (追加・削除) を取得する．この情報に基づいてコミット内のチャンクの全組み合わせのペアを作成する．この際に，2 つのチャンクの変更種別が同種 (Same) か異種 (Opposite) かの分類と，同一ファイルへの変更 (Intra) かそうでない (Inter) かの分類の 2 つの基準に沿ってデータを 4 種類に分類する．

学習の際には，各ペアが同一コミットに属すべきかどうかの正解ラベルが必要となる．提案手法では，既知の TLC と CC を対象に正解ラベルを抽出する．TLC 内のチャンク対は，全ての組み合わせに “結合”(1.0) のラベルを付与する．CC 内のチャンク対については，正しく TLC に分割された結果を用いて，分割後に同一の TLC に属するペアを “結合”(1.0), そうでないものは “分離”(0.0) として扱う．

### 3.2 チャンク間関係モデル

図 2 に提案手法のチャンク間関係推定の概要を示す．提案手法は，変更行に対応する AST ノードに着目し，チャンクの情報として固定長 $L$ の AST ノード列を利用する．各 AST ノードに対する $d$ 次元の埋め込み表現とノード列に含まれる特徴を SAN で学習することによって，任意のチャンク対 $\langle h_x, h_y \rangle$ の関係強度 $p$ を推定する．

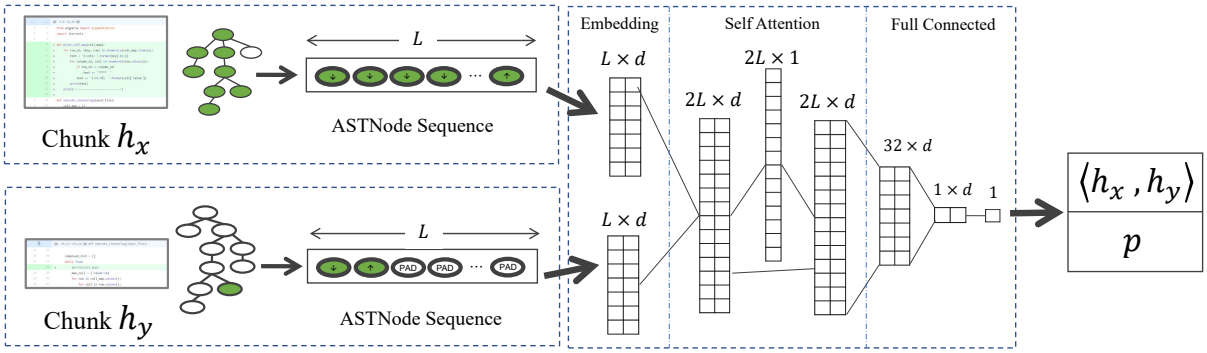


図 2　チャンク間関係の推定

#### 3.2.1　AST ノード列の抽出

チャンクの特徴量として，AST ノード列を利用する．ソースコードから AST を生成し，行番号で特定することで，AST のうちチャンクに内包されているノード群を抽出する．このノード群から，チャンク内のコードの構造的特徴を学習できるよう AST ノード列を抽出する．ノードの親子関係を明確にするために，各ノードを開始ノード (↓) と終了ノード (↑) に分解し，深さ優先探索の順序で並べて AST ノード列を生成する．プログラムユニット開始部分がチャンクに含まれている場合，抽出されるノード列にはそのユニットの開始ノードが含まれる．一方，プログラムユニット終了部分がチャンクに含まれている場合は，ユニットの終了ノードが含まれる．AST の深さ優先探索時に，降下しながらそのノードを通過した時，開始ノードを取得し，逆に上昇しながらそのノードを通過した時，終了ノードを取得する．開始ノードと終了ノードを別々に要素として扱うことで，チャンクの範囲を厳密にデータに埋め込むことを狙う．

本研究で扱うモデルは，長さ $L$ の固定長 AST ノード列を入力として受け付ける．そのため，AST ノード列をモデルに与えられるように整形する．AST ノード列が長さ $L$ を超えていた場合は，AST ノード列の長さが $L$ になるように列の両端からノードを捨てていくトリミング処理を行う．また，AST ノード列の長さが $L$ 未満であった場合，AST ノード列の後ろに，PAD という特殊ノードを追加し長さを $L$ にするパディング処理を行う．この PAD ノードは，のちの工程においてモデルがこれを学習しないように処理される．

#### 3.2.2　識別子ノード (NameExpr) の書き換え

code2vec [9] などのソースコードの分散表現を生成する手法では，識別子の名前情報を利用して埋め込みを行うため，一定の精度でソースコードの意味的表現を得ることが可能であるが大規模なデータセットによる学習を必要とする．

本手法は，既知の CC の情報が必要であり多様な識別子情報を活用するための大規模な学習は困難であるため，識別子の名前情報そのものは利用せず，変更チャンク同士で共通の識別子名を所持しているかどうかの情報を利用する．

識別子の名前情報を検査し，チャンク対から得た AST ノード列に対し，共通して存在する識別子は NameExpr-shared という新しいノードに置き換える．共通していない識別子については，名前情報のない NameExpr ノードに書き換える．後段の処理では，識別子は NameExpr と NamaExpr-shared の 2 種類に統合される．この書き換えにより，共通の識別子に基づいた特徴を学習する．

#### 3.2.3　学習モデル

本研究のニューラルネットワークモデルは，2 つの AST ノード列を入力として受け取り，それらのチャンク間の関係性の強度 $p$ を返すように構成されている．AST ノード列は Embedding 層でノードごとに分散表現へ変換して SAN に入力され，全結合層を経て強度 $p$ を出力する．$p$ は結合ラベルを 1.0, 分離ラベルを 0.0 で学習し，推定時には [0..1] の実数で出力される．

Embedding 層では，AST ノード辞書と Embedding Matrix を利用することで，AST ノード列を，対応する浮動小数点数型の行列へと変換する．まず，AST ノード列を入力として扱うために，学習用データに含まれる全ノード種別の辞書を作成し，ノード種別と整数値の ID を対応づける．この AST ノードの ID を Embedding Matrix に与えることで，その AST ノードを表す $d$ 次元のベクトルが得られる．Embedding Matrix は乱数で初期化され，モデルの学習時に同時に学習される．

本提案手法では，学習モデルに code2vec [9] でも採用されている Self-Attention Network(SAN) [12] を採用した．また，Attention に Positional Encoding [18] を取り入れた．AST トークン列はその順序によってチャンクの構造的な情報を表現しているが，単純な Attention だけでは入力データの順序関係を学習できない．Positional Encoding を採用することで順序関係の考慮が可能となり，モデルがチャンクの構造的情報を学習できるようになる．

AST ノード列ペアの入力に対して，Embedding 層で得られた 2 つの浮動小数点数型の $L \times d$ 次元行列のそれぞれに Positional Encoding を適用してから連結し，これを 1 つの $2L \times d$ 次元行列とする．得られた行列に Attention Weight をかける．本手法では，Self-Attention を利用しているため，Attention Weight は入力の行列を用いて計算される．Attention 適用後は，全結合層を通して強度 $p$ が出力される．学習については，code2vec と同様に二乗誤差を ADAM [19] で最適化することで行う．

### 3.3　チャンク間関係グラフの分割

4 つのチャンク間関係モデルによって，全てのチャンクペア間の関係強度 $p$ が求まり，コミット内のチャンクの関係は重み付き完全グラフとなる．提案手法では，[4] の手法と同様にマル

表 1　データセットの詳細

| | ArgoUML | GWT | Jaxen | JRuby | XStream |
|---|---|---|---|---|---|
| コード行数 (KLOC) | 165 | 266 | 21 | 102 | 22 |
| 開発年月（月） | 150 | 54 | 114 | 105 | 90 |
| 開発人数 | 50 | 120 | 20 | 67 | 12 |
| 総コミット数 | 16,481 | 5,326 | 1,353 | 11,134 | 1,756 |
| バグ修正コミットの数 | 2,945 | 809 | 105 | 2,977 | 312 |
| TLC の数 | 125 | 44 | 32 | 200 | 40 |
| TLC の割合 | 4.2% | 5.4% | 30.5% | 6.7% | 12.8% |

チレベルグラフ分割を応用して完全グラフを分割する．

最重エッジを選択してノードを統合し，統合前のエッジの中で最も強く結合・分離の判定しているものを統合後エッジとすることで，特徴が曖昧な関係の優先度を下げて利用する分割戦略を採用している．具体的には以下の手順で分割を行う．

**(手順 1)**　全チャンクを独立したコミットとみなす．

**(手順 2)**　重み $p$ が最も高い関係を持つコミットペアを統合する．統合後のコミットと残りのコミットとの関係を，統合前のコミットとの関係のうち重みが 0.5 より最も離れた値を持つ関係で置き換える．

**(手順 3)**　所定のコミット数以下になるまで手順 2 を繰り返す．

## 4. 評　　価

提案手法の特徴である，SAN を用いた関係判定とチャンクペアの分類の効果を確認するために，以下の 3 つの観点で実験を行った．以下ではまず，実験に用いたデータセットと実験設定を説明し，次に各観点での実験結果と考察を述べる．

**RQ1**　提案手法のチャンク間関係の判定精度はどの程度か？

**RQ2**　チャンク対の分類は判定精度に影響を与えるか？

**RQ3**　提案手法のコミット分割の精度はどの程度か？

### 4.1 データセット

本研究では，Herzig らの初期の研究 [13] で使用された 5 つのオープンソース Java プロジェクトのデータセットを利用する．各プロジェクトの詳細を表 1 に示す．これらのプロジェクトは，50 ヶ月以上に渡る開発期間と，10 人以上のアクティブな開発者によって開発されている．各プロジェクトのコード行数に着目すると小規模なものから大規模なものまで存在しており，またコミット数やバグ修正の数も多岐にわたっている．

このデータセットには，以下の手順で選ばれた TLC の情報が含まれる．これらの TLC は，Herzig らの研究に限らず，Muylaert らの研究でも使用されている．

（1） 解決済みバグレポートに紐づいているコミットを選択

（2） 複数のタスクに紐づいていることがコミットメッセージより明らかなものを除去

（3） 変更内容からバグ修正のみと確認できるものを選択

コミット分割の評価実験のためには，入力データとして CC とその正しい分割方法が必要であるが，CC の正しい分割方法の決定には，そのプロジェクトに精通した開発者の判断が必要である．このため本研究では，Herzig らが定義した人工的に CC を生成する方法 [4] を採用することとした．

人工的な CC は TLC を組み合わせて生成する．ランダムに組み合わせてしまうと現実世界の CC を再現できない可能性があるため，Herzig らは 1) コミット日時の間隔が 14 日以内であり, 2-a) コミットによって変更されたファイル同士の距離がサブパッケージ 2 つ以内である，もしくは 2-b) 過去に 3 回以上共変更が起こっているような TLC のペアを CC としている．2-b) の条件は合致する TLC ペアが少ないこと, 73%の CC は 2 つの TLC に分解できることが報告 [4] されているため，本研究では，1) と 2-a) のみを条件として採用し，組み合わせる TLC の数は 2 としている．Herzig らが実際に使用した TLC の組み合わせは公開されていなかったため，TLC を組み合わせる処理は著者が実装したものを利用した．

表 2　学習用データセットのサイズ

| | Intra-Same | Intra-Opp. | Inter-Same | Inter-Opp. |
|---|---|---|---|---|
| ArgoUML | 202 | 206 | 948 | 872 |
| | (2,301) | (2,449) | (2,962) | (2,852) |
| GWT | 40 | 40 | 470 | 458 |
| | (1,025) | (1,052) | (1,279) | (1,134) |
| Jaxen | 0 | 0 | 136 | 128 |
| | (3,520) | (3,647) | (1,829) | (1,824) |
| JRuby | 2,864 | 2,810 | 18,150 | 17,862 |
| | (7,970) | (8,285) | (53,027) | (50,895) |
| XStream | 140 | 158 | 1,696 | 1,170 |
| | (529) | (513) | (1,891) | (1,507) |
| 合計 | **3,246** | **3,214** | **21,536** | **20,618** |
| | (15,345) | (15,946) | (60,988) | (58,212) |

表 3　検証用データセットのサイズ

| | Intra-Same | Intra-Opp | Inter-Same | Inter-Opp |
|---|---|---|---|---|
| ArgoUML | 212 | 235 | 342 | 331 |
| GWT | 108 | 124 | 223 | 233 |
| Jaxen | 27 | 37 | 39 | 39 |
| JRuby | 192 | 209 | 2,172 | 1,727 |
| XStream | 304 | 344 | 1,101 | 1,031 |
| 合計 | **843** | **949** | **3,877** | **3,361** |
| 結合合計 | 823 | 928 | 936 | 815 |
| 分離合計 | 20 | 21 | 2,941 | 2,546 |
| Imbalance Ratio | 0.024 | 0.023 | 3.132 | 3.124 |

前述の方法で得られた TLC と CC に対して，3.1 で述べた方法で結合と分離の正解ラベルを含むデータを生成する．これらのデータを学習用と検証用に分割する．学習用データセットに対して，過学習を防ぐために，結合ラベルデータと分離ラベルデータが同数となるように整形する．表 2 に，学習用データの整形前のデータ数と整形後のデータ数をそれぞれ示す．少数のラベルデータに合わせて削減するため整形によって大幅にデータ数が減る結果となった．表 3 に検証用データのデータ数を示す．検証用データでは整形は行わないため Imbalance Ratio(分離/結合) は 1 ではない不均衡データである点に注意されたい．

### 4.2 実 験 設 定

JavaParser(注1)によって AST ノード抽出部分を，TensorFlow(注2)によってニューラルネットワークモデル部分をそれぞれ実装し，提案手法のパラメータである，AST ノード列の長さ

(注1)：https://github.com/javaparser/javaparser

(注2)：https://www.tensorflow.org/

表 4　分類ごとの判定精度

| | Intra-Same | Intra-Opp | Inter-Same | Inter-Opp |
|---|---|---|---|---|
| **Accuracy** | **0.534** | **0.542** | **0.693** | **0.722** |
| Prec(結合) | 0.991 | 0.996 | 0.354 | 0.385 |
| Recall(結合) | 0.527 | 0.533 | 0.332 | 0.248 |
| **F1(結合)** | **0.688** | **0.694** | **0.343** | **0.302** |
| Prec(分離) | 0.040 | 0.042 | 0.792 | 0.784 |
| Recall(分離) | 0.800 | 0.905 | 0.807 | 0.873 |
| **F1(分離)** | **0.076** | **0.080** | **0.799** | **0.826** |

表 5　チャンク間関係の判定精度

| | 提案手法 (分類あり) | 提案手法 (分類なし) |
|---|---|---|
| Accuracy | **0.673** | 0.632 |
| Prec(結合) | **0.617** | 0.546 |
| Recall(結合) | **0.412** | 0.299 |
| F1(結合) | **0.494** | 0.386 |
| Prec(分離) | **0.692** | 0.655 |
| Recall(分離) | 0.838 | **0.843** |
| F1(分離) | **0.758** | 0.737 |

$L$ を 200, ノード埋め込みの次元数 $d$ を 128，分割停止コミット数を 2 として実験を行った．実験には Intel Xeon E5 の 2way 構成でメモリを 128GB，GPU に GeForce RTX2060 を搭載した計算機を利用した．

チャンク間の関係推定については, 出力の値が 0.5 以上の場合は結合，0.5 未満の場合は分離として，Accuracy，Precision, Recall, F1-score の値をそれぞれ評価する．チャンク対の分類ごとにデータセットを学習用と検証用に分割し，バッチ数 256，エポック数 20 として学習を行い，エポックごとに検証用データを用いて検証を行う．20 エポックの学習が終了した時点での検証結果を評価する．また，チャンク対分類の効果を確認するために，4 分類を全て混ぜ合わせたデータセットに対しても同様に学習，検証を行う．コミット分割の精度については，Herzig が定義した以下の分割成功率 (Success Rate) [13] を用いる．

$$\text{分割成功率} = \frac{\text{正解 TLC に分割できたチャンク数}}{\text{コミット内の全チャンク数}}$$

既存手法である Herzig ら手法の再実装は困難であったため，彼らが報告した精度を参考値として比較することとした．文献 [13] では分割成功率のみが記載されているため，RQ1,RQ2 の観点では比較ができなかった．

### 4.3　実験結果 (RQ1)：チャンク間関係の判定精度

データを分類ごとに分けた時のモデルの判定精度を表 4 に示す．変更種別が同種 (Same-Type) と異種 (Opposite-Type) の間には大きな差は見られなかったが，変更箇所の分類では判定精度に異なる特徴が確認できた．

変更箇所が異なるファイル間のチャンク対 (Inter) については，7 割前後の Accuracy であった．ファイルが別であるならば別コミットに分離すべき確率が高くなるため，分離の F1-score が 0.8 前後と高精度なだけでなく，結合についても 0.30〜0.34 と一定の精度で判定できていることは，SAN を用いた特徴学習が有用であったものと考える．

一方で，変更箇所が同一ファイル内のチャンク対 (Intra) の判定については，Accuracy が 0.54 前後と判定精度が低い結果となった．Intra の 2 分類は，学習が 20 エポック付近まで進んでも精度が 5%から 10%ほどの幅で上下しており収束しなかった．学習用と検証用ともにデータ数が少なく，データ数の不足が原因と考える．判定結果別でみると，分離の F1-score が 0.08 前後と低いことが分かる．これは recall は 8 割を超えているが，precision が 0.04 前後と極めて低い値となっていることから，Intra に対しても多くを分離と判定し，その多くが誤検出となっていることが原因である．このことから，特に分離の学習用データが不足しているものと考える．

### 4.4　実験結果 (RQ2)：チャンク対分類の効果

分類ごとの判定結果を統合しデータセット全体での判定精度と，分類をせずに学習した場合の結果を足し合わせたものを表 5 に示す．結合と分離の双方の F1-score が優れており Accuracy も高いことから，提案手法の特徴であるチャンクペアの分類は有効であることが確認できる．特に，分類なしでは結合の recall が 0.299 と低い結果となっている．これは，Inter-File のデータ数は Intra-File のデータ数の約 7 倍であり，分類をしない場合はこの影響で Intra-File の特徴量が学習しづらくなり結合すべきチャンクペアを正しく判定できないことが原因と考える．

また，分類をした場合でも，結合の F1-score は 0.494 と分離の F1-score と比べて低いことから，モデルが全体的に分離判定を出しやすくなっていることが分かる．本実験で利用したデータセットについて，結合のデータに関しては同時にコミットされたという条件のもと集められているが，一方分離のデータは 4.1 で述べた条件に従った人工的な CC を用いてる．この人工的な CC を構成する条件が不十分であり，幅広い特徴量を持った CC が生成されたことで，分離と判定する範囲が広くなっているものと考える．

### 4.5　実験結果 (RQ3)：コミット分割の精度

表 6 に提案手法と提案手法をチャンク対の分類せずに適用した場合の平均分割成功率を示す．また，学習・評価に利用したデータが異なることの影響があるため直接比較はできないが，既存手法の精度の参考情報として文献 [13] での 2 つの TLC で構成された CC に対する平均分割成功率も含めている．

3.3 で述べた方法でコミットを分割するため，全てのチャンク対関係の推定が正しくなくても正しくコミット分割される場合があるが，ArgoUML 以外では分類しない場合と比べて高い分割成功率であり, 全体で 74% のチャンクを正しい TLC に割り当てることができている．このことから，学習時のチャンク対分類は，コミット分割に重要となるチャンク間関係学習に有効であるものと考える．

また，Herzig らの結果 [13] と比べ, Jaxen, Jruby では分割成功率が低い結果となっているが，全体としては同程度かつより安定した分割ができていると考える．

## 5.　議　　論

前節の評価結果を踏まえて，改善すべき点について議論する．

表 6　コミット分割の平均分割成功率の比較

| | Herzig [13] | 提案手法 (分類あり) | 提案手法 (分類なし) |
|---|---|---|---|
| ArgoUML | 0.60 | 0.69 | **0.72** |
| GWT | 0.77 | **0.80** | 0.77 |
| Jaxen | **0.91** | 0.67 | 0.66 |
| JRuby | **0.84** | 0.74 | 0.71 |
| XStream | 0.75 | **0.78** | 0.70 |
| 全体 | 0.73 | **0.74** | 0.71 |

まず，評価をするためのデータセットの規模，選択方法を改善が必要である．本稿でのデータセットは全体として約 50,000 のチャンク対を用いたが，それらは 400 ほどのコミットから生成されたものであった．Intra-File のチャンク対を増やすだけでなく，より多くのコミットを対象としてデータセットを拡充する必要がある．

また，人工的に CC を生成する手法についても，可能な限り現実の CC を反映できるような基準を採用したが，現実世界の CC の特徴を網羅できているとは限らない．より正確な評価のためには，多数の現実の CC とそれらの正しい分割結果が必要となるが，特に正しい分割結果の入手は難しい．分割の正しさについては，Muylaert らの研究 [17] のように，開発者の反応を集計することで評価する方法があるが，学習に必要なデータ量を集めることは困難である．

我々は，効率良く CC とその正しい分割を得る手段として，Dias らの提案手法 [11] のように IDE 等への操作履歴を記録し，提案した分割の仕方が正しいかどうかを開発者が確認・修正した結果を利用する方法が有効であると考えている．そこで，本研究でも提案手法をツール化し，既存コミットの分割結果だけでなく，コミットを行う前の Staging の段階でチャンクを識別・クラスタリングすることで，複数の TLC に分解してコミットする方法を提案し，開発者の評価を受ける方法を検討している．

また，ソースコードの Embedding 領域に関しては，様々な手法が提案され続けており [8]，これらを応用することでチャンクの Embedding 方法について改善ができると考える．例えば，Zhang らの研究 [10] で提案された AST の Embedding 手法は，ソースコード分類やコードクローン検知の領域で優れた結果を出している．本手法では，チャンク単位での抽出が必要であり，彼らの手法をそのまま取り入れることはできなかったが，彼らの手法を拡張することによって，ソースコードの断片，特に，それ単体では構文が作れないような断片についても Embedding が可能になれば精度が向上すると考える．

本稿では，チャンク間関係の学習と，チャンクペア分類の効果を中心に議論したため，グラフ分割については分割停止数を固定した方法を採用したが，実際に適用する際には分割するべき TLC の数は一定ではない．このため，関係の強度に応じてグラフ分割を停止するように変更する必要がある．また，分割戦略についても，RQ1 と RQ2 の結果を考慮し，統合後の関係強度の決定方法について詳細な研究が必要であると考える．

## 6. おわりに

本研究では，コミットを構成するチャンク間の複雑な関係をニューラルネットワークによって学習・推定することで，複合コミット (CC) を自動分割する手法を提案した．先行研究と同様のデータセットを用意して実験を行い，提案手法はチャンクペアの結合・分離の判定を Accuracy 0.67 と一定の精度で判定できること，チャンクペアの分類が有効であることを示した．また，人工的に生成した CC を用いた複合コミット分割の実験の結果，チャンクペア分類はコミット分割の精度向上に寄与しており 69〜80%のチャンクを正しく割り当てることができ，先行研究と同程度の精度であることを示した．

**謝辞**：本研究の一部は科研費（#18H03221）の助成を受けた．

### 文　献

[1] Y. Kamei, et al. Studying just-in-time defect prediction using cross-project models. *Empir. Softw. Eng.*, Vol. 21, No. 5, 2016.
[2] 森, A. Hagward, 小林. 改版履歴の分析に基づく変更支援手法における時間的近接性の考慮と同一作業コミットの統合による影響. 情報処理学会論文誌, Vol. 58, No. 4, 2017.
[3] S. P. Berczuk and B. Appleton. *Software Configuration Management Patterns*. Addison-Wesley Professional, 2002.
[4] K. Herzig. *Mining and untangling change genealogies*. PhD thesis, Universität des Saarlandes, 2012.
[5] H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *Proc. ISSRE*, 2013.
[6] Y. Tao and S. Kim. Partitioning composite code changes to facilitate code review. In *Proc. MSR*, 2015.
[7] D. Kawrykow. *Enabling precise interpretations of software change data*. PhD thesis, McGill University, 2011.
[8] Z. Chen and M. Monperrus. A literature study of embeddings on source code. arXiv:1904.03061, Apr. 2019.
[9] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proc. POPL*, 2019.
[10] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. A novel neural source code representation based on abstract syntax tree. In *Proc. ICSE*, 2019.
[11] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, , and S. Ducasse. Untangling fine-grained code changes. In *Proc. SANER*, 2015.
[12] Z. Lin, M. Feng, et al. A structured self-attentive sentence embedding. In *Proc. ICLR*, 2017.
[13] K. Herzig and A. Zeller. Untangling changes. September 2011.
[14] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Proc. SC*, 1995.
[15] T. Zimmermann, et al. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, Vol. 31, No. 6, 2005.
[16] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, Vol. 33, No. 11, 2007.
[17] W. Muylaert and C. D. Roover. Untangling composite commits using program slicing. In *Proc. SCAM*, 2018.
[18] A. Vaswani, et al. Attention is all you need. In *Proc. NIPS*, 2017.
[19] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Proc. ICLR*, 2015.