修士学位論文

題目

ベクトル表現と LSH アルゴリズムを用いた インクリメンタルコードクローン検出法

指導教員
井上 克郎 教授

報告者
本田 紘貴

令和 2 年 2 月 5 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

令和元年度 修士学位論文

ベクトル表現と LSH アルゴリズムを用いたインクリメンタルコードクローン検出法

本田 紘貴

内容梗概

ソフトウェア保守を困難にする大きな要因としてコードクローンが指摘されている．コードクローンとは，ソースコード中に存在する互いに一致，または類似したコード片を指し，主に既存のコード片のコピーアンドペーストによって生成される．大規模なソースコード中のコードクローンを目視で見つけることは困難であるため，これまでにコードクローンを自動で検出する様々な手法やツールが提案されている．

コードクローン検出を用いた応用の 1 つに，複数コミット間のコードクローンの変更履歴を追跡することが挙げられる．複数コミットに渡って，コードクローンの変更履歴を追跡することで，一貫した修正がされていないコードクローンに対しては同時修正を検討することができ，新たに発生したコードクローンに対しては集約を検討することができる．また，変更があったコードクローンを重点的に確認することで確認コストを削減することができる．これらの応用をより活かすために，複数コミットのソースコードから，コードクローンを正確に素早く検出できるツールが求められる．

横井らは，情報検索技術に基づいてコードブロック単位でコードクローンを検出するツール CCVolti を提案した．CCVolti は，情報検索技術で用いられる TF-IDF (Term Frequency-Inverse Document Frequency) を利用し，コードブロックを特徴ベクトルとして表現することで，構文的だけでなく，意味的に類似したコードクローンを検出可能にする．

しかし，CCVolti を用いて，複数コミットのソースコードからコードクローンを検出する際に、2 つの問題がある． 1 つ目の問題は，検出結果の非一貫性である．複数コミットからコードクローンを検出する場合、CCVolti でコードクローン検出のために使用する TF-IDF の IDF 値はコミットごとに変わるため，同一コード片でも特徴ベクトルがコミットごとに変化する．このため，旧コミットではコードクローンとして検出されたコード片が新コミットでは検出されない検出結果の非一貫性が発生する．2 つ目の問題は，複数コミットのソースコードからコードクローンを検出するにはスケーラビリティが不十分である．開発者が複数コミットのソースコードからコードクローンを検出する場合，コードクローン検出ツールのスケーラビリティは重要である．. CCVolti は 1 つのコミットのみを対象に検出を行うよ

うに設計されているため，複数コミットからコードクローンを検出する場合，コミット間の差分が小さい場合でも，各コミットのソースコード全体に対して，コードクローン検出を行う．したがって，大規模ソフトウェアの複数コミットをコードクローン検出の対象とする場合に，実用的な時間でコードクローンを検出することが困難である．

これらの問題を解決するために、本研究では，CCVolti をインクリメンタルコードクローン検出ができるように拡張する．インクリメンタルコードクローン検出とは，1 番目で検出したコードクローンの情報を保存して，2 番目以降の検出では，2 つのコミット間の差分を対象にコードクローンを検出し，保存したコードクローンの情報を更新することである．このため，以前のコミットで検出されたコードクローンが新コミットでもコードクローンとして検出可能であり，検出結果の非一貫性を解決することができる．また，2 つのコミットの差分のみを対象にコードクローンを検出するため，検出時間を短縮し，スケーラビリティを向上させることができる．

評価実験では，2 つの C 言語プロジェクトと 2 つの Java プロジェクトに対して CCVolti と本手法を適用し，コードクローンの検出精度と検出時間の観点で比較評価を行った．その結果，コードクローンの検出精度においては，本手法は CCVolti の検出するクローンペアの内，約 92%から 100%を検出するできる上に，より多くのクローンペアを高い適合率で検出した．また，本手法が CCVolti より，約 2.7〜7.1 倍高速に複数コミットからコードクローンを検出することも確認できた．

主な用語

ソフトウェア保守
インクリメンタルコードクローン検出
コードクローン進化

目 次

1　まえがき

ソフトウェア保守を困難にする大きな要因としてコードクローンが指摘されている [10]. コードクローンとは，ソースコード中に存在する互いに一致，または類似したコード片を指し，主に既存のコード片のコピーアンドペーストによって生成される．コードクローンに対する様々な保守や管理の方法が提案されているが，ソースコードの規模が大きくなるとソースコード中に含まれるコードクローンも膨大な量となり，目視でコードクローンを見つけることは困難である．そこで，コードクローンをソースコードから自動的に検出する様々な手法やツールが提案されている [20][9][14][6][11][7].

コードクローン検出を用いた応用の 1 つに，複数コミット間のコードクローンの変更履歴を追跡することが挙げられる．複数コミットに渡って，コードクローンの変更履歴を追跡することで，一貫した修正がされていないコードクローンに対しては同時修正を検討することができ，新たに発生したコードクローンに対しては集約を検討することができる．また，変更があったコードクローンを重点的に確認することで確認コストを削減することができる．これらの応用をより活かすために，複数コミットのソースコードから，コードクローンを正確に素早く検出できるツールが求められる．

横井らは，情報検索技術を利用することによって，意味的に処理が類似したコードブロック単位でコードクローンを検出するツール CCVolti[16] を提案した．CCVolti は，情報検索技術の 1 つである TF-IDF (Term Frequency-Inverse Document Frequency) 法 [3] を用いて，ソースコード中の識別子や予約語に利用される単語に対して重み付けを行う．そして，重み付けに基づいて各コードブロックを特徴ベクトルに変換し，特徴ベクトル間の類似度を計算することによって，コードクローンを検出する．また，近似最近傍探索アルゴリズムの一種である LSH（Locality-Sensitive Hashing）アルゴリズム [1] を用いて，特徴ベクトルをクラスタリングすることにより，高速にコードクローンを検出する．

しかし，CCVolti を用いて，複数コミットのソースコードからコードクローンを検出する際に、2 つの問題がある 1 つ目の問題は，検出結果の非一貫性である．複数コミットからコードクローンを検出する場合、CCVolti でコードクローン検出のために使用する TF-IDF の IDF 値はコミットごとに変わるため，同一コード片でも特徴ベクトルがコミットごとに変化する．このため，旧コミットではコードクローンとして検出されたコード片が新コミットでは検出されない検出結果の非一貫性が発生する．2 つ目の問題は，複数コミットのソースコードからコードクローンを検出するにはスケーラビリティが不十分である．開発者が複数コミットのソースコードからコードクローンを検出する場合，コードクローン検出ツールのスケーラビリティは重要である．CCVolti は 1 つのコミットのみを対象に検出を行うように設計されているため，複数コミットからコードクローンを検出する場合，コミット間の

差分が小さい場合でも，各コミットのソースコード全体に対して，コードクローン検出を行う．したがって，大規模ソフトウェアの複数コミットをコードクローン検出の対象とする場合に，実用的な時間でコードクローンを検出することが困難である．

これらの問題を解決するために、本研究では，CCVolti をインクリメンタルコードクローン検出ができるように拡張する．インクリメンタルコードクローン検出とは，1 番目のコミットで検出したコードクローンの情報を保存して，2 番目以降のコミットのコードクローン検出では，2 つのコミット間の差分を対象にコードクローンを検出し，保存したコードクローンの情報を更新することである．このため，以前のコミットで検出されたコードクローンが新コミットでもコードクローンとして検出可能であり，検出結果の非一貫性を解決することができる．また，2 つのコミットの差分のみを対象にコードクローンを検出するため，検出時間を短縮し，スケーラビリティを向上させることができる．

評価実験では，2 つの C 言語プロジェクトと 2 つの Java プロジェクトに対して CCVolti と本手法を適用し，コードクローンの検出精度と検出時間の観点で比較評価を行った．その結果，コードクローンの検出精度においては，本手法は CCVolti の検出するクローンペアの内，約 92%から 100%を検出するできる上に，より多くのクローンペアを高い適合率で検出した．また，本手法が CCVolti より，約 2.7〜7.1 倍高速に複数コミットからコードクローンを検出することも確認できた．

以降，2 章では，コードクローンの検出技術，コードクローン管理支援ツール，インクリメンタルコードクローン検出，およびコードクローン検出ツール CCVolti とその問題点について述べる．3 章では，提案するインクリメンタルコードクローン検出手法について述べる．4 章では，本手法の評価実験について述べる．最後に 5 章では，まとめと今後の課題について述べる．

2　コードクローン

コードクローンとは，ソースコード中に存在する互いに一致，または類似したコード片を指し，主に既存のコード片のコピーアンドペーストによって生成される．一般的に，互いにコードクローンとなるコード片はクローンペアと呼ばれ，クローンペアにおいて推移関係が成り立つコードクローンの集合はクローンセットと呼ばれる．

コードクローンの存在は，ソフトウェア保守を困難にしている大きな要因の 1 つとして挙げられている [10]．あるコード片に欠陥が見つかった場合，そのコード片のコードクローンにも欠陥が含まれる可能性が高い．このため，開発者はそのコードクローンと同一クローンセットに含まれている全てのコードクローンに対して同様の修正を行うか検討する必要があり，欠陥を修正するために大きなコストが必要となる．このため，開発者がコードクローンの位置などの情報を認識しておく必要がある．しかし，大規模なソフトウェアにおいては，目視ですべてのコードクローンを見つけることは非現実的である．そこで，コードクローンをソースコードから自動的に検出する様々な手法やツールが提案されている [20].

コードクローンには普遍的定義が存在しない．本論文では，コードクローンの差異の度合いに基づいて以下の 4 つのタイプの分類を用いる [12].

タイプ 1

空白，タブ，改行，コメント，およびコーディングスタイルなどの違いを除いて完全に一致するコードクローン

タイプ 2

タイプ 1 に加えて識別子，リテラル，および型の違いを除いて一致するコードクローン

タイプ 3

タイプ 2 に加えて，文の変更・挿入・削除などの違いを除いて一致するコードクローン

タイプ 4

構文上の実装は異なるが，同様の処理を行うコードクローン
タイプ 4 のコードクローンとして以下のものが挙げられる．

- 条件分岐処理や繰り返し処理などの制御構造の実装が異なるもの
- 中間媒介変数の利用の有無があるもの
- 文が並び替えられているもの

2.1 コードクローンの検出技術

ソースコードの規模が大きくなるとソースコード中に含まれるコードクローンも膨大な量となり，目視でコードクローンを見つけることが困難である．そこで，コードクローンをソースコードから自動的に検出することを目的とした様々なコードクローン検出手法が提案されている． コードクローン検出するための代表的な技術は以下の通りである

トークンベースの検出

トークン単位の検出では，字句解析で入力ソースコードをトークン列に変換して，トークン列の類似度が閾値以上であるものをコードクローンとして検出する．この技術は，プログラミング言語の文法規則を無視したコードクローンを検出してしまうことがある．トークンベースの検出を行う代表的なコードクローン検出ツールとして，神谷らが開発した CCFinder[8] がある．CCFinder は字句解析時に変数名や関数名などの識別子をある 1 つのトークンに置き換えることでタイプ 2 までのコードクローンを検出することができる．または，Sajnani らが開発した SourcererCC[14] は，大規模なソフトウェアに対して，構文的に類似度が高いタイプ 3 までのコードクローンを高速に検出することができる．

ベクトルベースの検出

ベクトルベースの検出では，コード片をベクトル表現に変換し，ベクトル表現の類似度が閾値以上であるものをコードクローンとして検出する．ベクトルベースで検出することで，トークンや構文単位で類似していないコード片でもベクトル空間上で距離が近ければコードクローンとして検出できる．ベクトルベースの検出を行う代表的なコードクローン検出ツールとして，山中らが開発した関数クローン検出ツール [17] がある．関数クローン検出ツールは，情報検索技術の 1 つである TF-IDF を用いて，関数単位のコード片を特徴ベクトルに変換し，LSH を用いてクラスタリングすることでコードクローンを検出する．TF-IDF を用いることでトークン列や構文の類似度に依存せずに検出を行えるため，タイプ 4 までのコードクローンを検出することができる．また，横井らが開発したコードブロック単位のコードクローン検出ツール CCVolti[16] がある．CCVolti は，関数クローン検出ツールを基に開発され，検出粒度をより小さくしたコードブロック単位でコードクローンを検出する．ここで，コードブロックとは，関数と，関数内部の if，for，while 文などの中括弧で囲まれた部分を指す．CCVolti は検出粒度をコードブロック単位にすることで，より多くのコードクローンを検出することができる．

インクリメンタルな検出

インクリメンタルな検出では，複数のコミットからコードクローンを検出する際に，1 番目ののコミットで検出したコードクローンの情報を保存して，2 番目以降のコミットの検出では，2 つのコミット間の差分を対象にコードクローン検出を行い，保存したコードクローンの情報を更新することで，複数のコミットからコードクローンを検出する．つまり，この技術は，2 つのコミット間で変更がないコード片に対してコードクローンの再検出を行なわず，編集，追加，削除されたコード片を対象にコードクローン検出を行う．インクリメンタルな検出は，コミット間の差分のみにコードクローン検出をすることで，複数のコミットから高速にコードクローンを検出できる．インクリメンタルな検出を行う代表的なコードクローン検出ツールとして，Göde らが開発した iClones[4] がある．iClones では，サフィックス木を用いて，コミット間のコード片の編集，追加，削除を調査して，コミット間のコード片を対応づける．そして，編集，追加，削除されたコード片のみを対象にコードクローン検出を行い，サフィックス木の類似度が閾値以上であるものをコードクローンとして検出する．iClones は，サフィックス木を用いることで，タイプ 2 までのコードクローンを検出することができる．

2.2 コードクローン管理支援ツール

コードクローンに対する主な管理作業として以下の作業が挙げられる．

同時修正

クローンセット中のコード片を一貫して修正をすること．クローンセット中のコード片に対して一貫していない修正を行う場合，新たな欠陥が混入される可能性がある．

集約

クローンセット中のコード片と同様の処理を実装するサブルーチンを作り，各コード片をそのサブルーチンの呼び出し文に置き換えること．集約を行うことで，コードクローンの数とソースコード量を削減することができる．

これらのコードクローンに対する管理作業を支援する代表的なツールとして，以下の 2 つのツールが挙げられる．

コードクローン変更管理システム Clone Notifier

山中らが開発したコードクローン変更管理システム Clone Notifier[18] は，2 バージョン間で行われたコードクローンの追加，編集，削除といった変更情報に基づき，クローンセットを分類し，その変更履歴情報を開発者に提供する．Clone Notifier は，2 バージョン間のクローンセットを以下の 4 つに分類する．

Stable クローンセット

2 バージョン間に存在し，変更がなかったクローンセット．

New クローンセット

新バージョンのみに存在するクローンセット．新バージョンで新たなコードクローンが追加されたことを示す．コードクローンの数を削減するために，New クローンセットは集約を検討する必要がある．

Deleted クローンセット

旧バージョンのみに存在するクローンセット．新バージョンで集約などによって削除されたクローンセットを意味する．開発者は Deleted クローンセットに分類されたクローンセットを確認することによって，集約が行われたことを確認することができる．

Changed クローンセット

2 バージョン間に存在し，変更されたクローンセット．そのうち，一貫した修正がされていない Changed クローンセットに対しては同時修正を検討する必要がある．

Clone Notifier は，このようなクローンセットの変更履歴情報を開発者に提供することで，管理作業の対象となるコードクローンの確認コストを削減することができる．また，Clone Notifier では，コードクローン検出ツールとして，2.1 節で述べた，CCFinder，SourcererCC，関数クローン検出ツール，および CCVolti が用いられている．

コードクローン変更履歴可視化システム CCEvovis

上記で述べた Clone Notifier は，2 バージョン間のみのコードクローンの変更履歴情報の提供するため，初心者の開発者が使うには困難である可能性がある．例えば，Clone Notifier にどの 2 バージョンを入力すれば良いか判断することが困難であることや，2 バージョン間のみのコードクローンの変更履歴情報の提供では，重要なコードクローンの変更履歴情報を見落としてしまうという問題が発生する恐れがある．この問題を軽減するために，本田らはコードクローン変更履歴可視化システム CCEvovis[5] を開発した．CCEVovis は，複数コミット間で行われたコードクローンの追加，編集，削除といったコードクローンの変更履歴に基づき，クローンセットを分類し，その変更履歴情報を開発者に提供する．また，ダッシュボードにより，クローンセットの変更履歴を可視化し，開発者が注目すべき 2 バージョン間の変更履歴情報を直観的に把握することを可能にした．CCEvovis を用いることで，開発者は，保守作業に必要なコードクローンの変更情報を効率的に把握することができる．

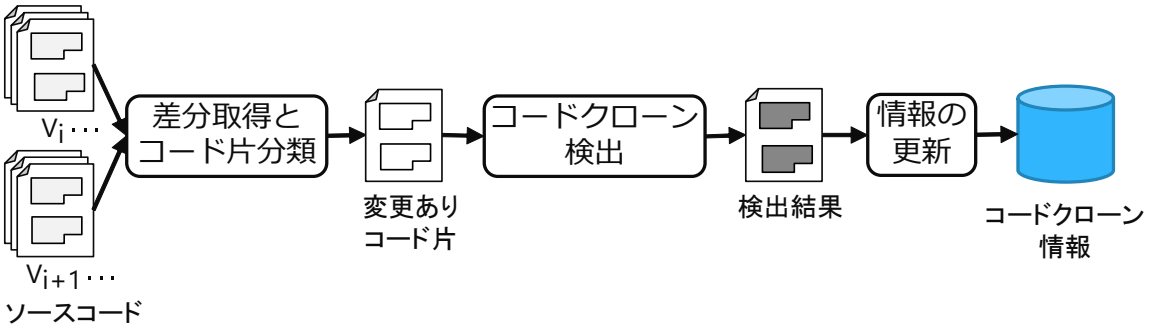


図 1: インクリメンタルコードクローン検出の概要

2.3 インクリメンタルコードクローン検出

2.1 節で説明したように，インクリメンタルコードクローン検出とは，1 番目のコミットで検出したコードクローンの情報を保存し，2 番目以降のコミットの検出では，2 つのコミット間の差分のみを対象にコードクローンを検出して，保存したコードクローンの情報を更新することで，複数のコミットからコードクローンを高速に検出する．図 1 に，インクリメンタルコードクローン検出の概要図を示す．

インクリメンタルコードクローン検出では，2 番目以降のコミットのコードクローン検出は，主に以下のように行なわれる．

1. 2 コミット間の差分取得とコード片の分類
   2 コミット間の差分を取得し，ソースコードの追加，編集，削除情報に基づき，コード片を分類する．

2. 変更されたコード片のみを対象コードクローン検出
   2 コミット間で追加，編集されたコード片のみを対象にコードクローン検出を行う．

3. コードクローン情報の更新
   以前のコミットから検出されたコードクローン情報のうち，2 コミット間で，追加，編集，削除されたコードクローンのみの情報を更新する．

このように，インクリメンタルコードクローン検出を用いて，2 番目以降のコミットのコードクローン検出では 2 コミット間の差分を対象にコードクローン検出をすることで，以前のコミットで検出されたコードクローンが新コミットで変更されない場合は，新コミットでもコードクローンとして検出可能なため，検出結果の非一貫性を解決することができる．また，差分を対象にコードクローンを検出するため，大規模なソフトウェアの複数コミットに対しても実用的な時間でコードクローンを検出することができる．

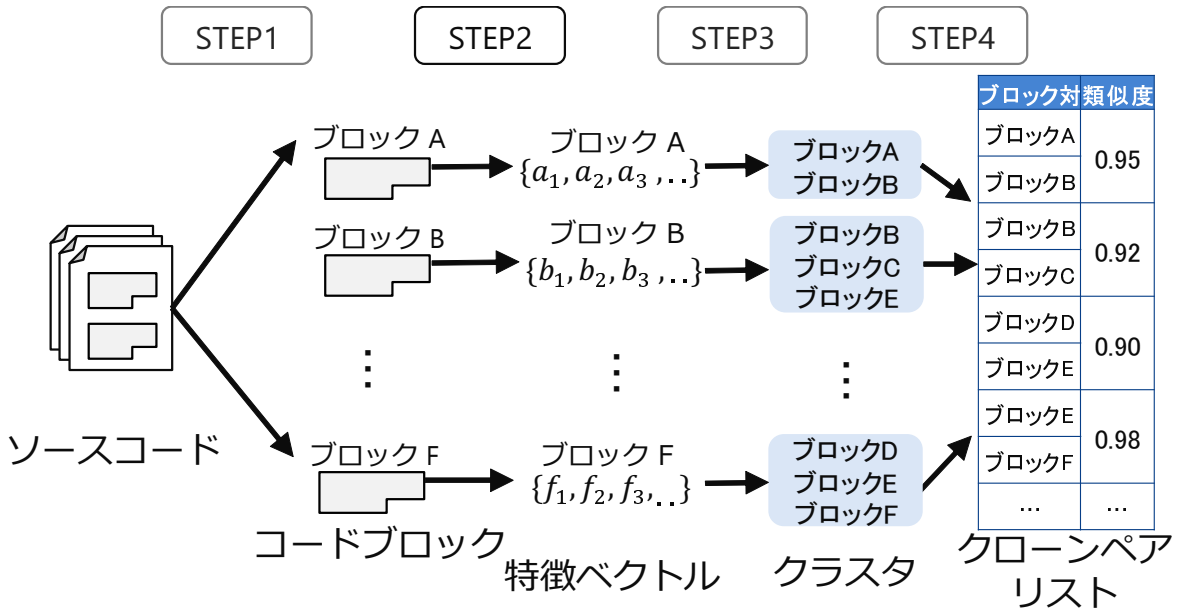


図 2: CCVolti のアルゴリズム

2.4 コードブロック単位のコードクローン検出ツール CCVolti

本節では，2.1 節で述べた，コードブロック単位のコードクローン検出ツール CCVolti のアルゴリズムと，その問題について述べる．

CCVolti のアルゴリズムについて，図 2 に示す．CCVolti のアルゴリズムでは，以下の STEP で入力のソースコードからコードクローンを検出する．

STEP1 コードブロックの取得

STEP2 特徴ベクトルの計算

STEP3 LSH アルゴリズムを用いた特徴ベクトルのクラスタリング

STEP4 コードクローンの検出

最初に，STEP1 で，構文解析を行い，抽象構文木を作成し，コードブロックを取得する．STEP2 では，コードブロックからワードを抽出し，そのワードに対して TF-IDF を計算し，コードブロックを特徴ベクトルに変換する．STEP3 では，各コードブロックの特徴ベクトルをクラスタリングする．最後に，STEP4 では，クラスタ内の特徴ベクトル間の類似度を計算することでクローンペアを検出する．

CCVolti は，特徴ベクトルの変換に TF-IDF を利用している．TF-IDF とは，文書中の単語に関する重み付けの手法で，TF 値と IDF 値の積で表される．

CCVolti では，コードブロック Y 内におけるワード X の TF 値と IDF 値は以下のように計算する．

$$\mathrm{TF}_{X,Y} = \frac{\text{コードブロック } Y \text{ 内におけるワード } X \text{ の出現回数}}{\text{コードブロック } Y \text{ 内における全ワードの出現回数の和}} \tag{1}$$

$$\mathrm{IDF}_X = \log \frac{\text{全関数の数}}{\text{ワード } X \text{ が出現する関数の数}} \tag{2}$$

上記の式のように，CCVolti では，コードブロック中のワードの出現頻度を TF 値，ソースコード全体の単語の希少さを IDF 値として計算している．CCVolti では，この 2 つの値をかけて求められた値，つまり各単語の重みを特徴量として，各コードブロックを特徴ベクトルに変換する．上記の式のように，TF 値は，コードブロック Y 内におけるワード X の出現回数と，コードブロック Y 内における全ワードの出現回数の和によって値が変化するため，コードブロック内の変更の影響を受けて値が変化する．また，IDF 値は，ソースコード全体の全関数の数とワード X が出現する関数の数によって値が変化するため，ソースコード全体の変更の影響を受けて値が変化する．また，CCVolti が計算する特徴ベクトルの次元数は，全関数内で n 回以上（デフォルトは n=2）出現するワードの種類数（以降，出現するワードの種類数）としている．

単一バージョンのソースコードに対してコードクローンを検出する場合，CCVolti は既存のコードクローン検出ツールより高速度かつ高精度でコードクローンを検出する．しかし，2.2 節で説明したように，コードクローンの変更履歴の確認や保守対象のコードクローンを特定するため，開発者は複数のコミットからコードクローンを検出し，その検出情報を用いてコードクローンを保守する必要がある．したがって，CCEvovis などで複数のコミットからコードクローンを検出するために CCVolti を使用した場合，以下の 2 つの問題が発生する可能性がある．

（問題 1）検出結果の非一貫性

CCVolti において，IDF 値は，ソースコード全体の全関数の数とワード X が出現する関数の数によって値が変化するため，ソースコード全体の変更の影響を受けて値が変化する．したがって，ソースコード全体でどこか一部が変更されると IDF 値は変化するため，2 コミット間でコード片が変更されていない場合でも変換される特徴ベクトルが変化する．このため，以前のコミットではコードクローンと検出されたコード片が新コミットでは検出されないという検出結果の非一貫性が発生する．非一貫性のあるコードクローンの検出結果を用いて，コードクローンの変更履歴を確認する場合，コミット間でコードクローンが変更されていない場合でも，新コミットではコードクローンが削除されたと間違った情報を開発者に提供してしまう可能性がある．

（問題 2）スケーラビリティが不十分

CCEvovis など，複数コミットから検出したコードクローンの情報を必要とするツールにとって，コードクローン検出ツールのスケーラビリティは重要である．しかし，CCVolti は，単一のコミットを対象にコードクローンを検出することを前提に開発されている．このため，複数コミットをコードクローンの検出対象とする場合に，コミット間の差分が小さいにも関わらず，各コミットのソースコード全体に対して，コードクローン検出を行う．したがって，大規模なソフトウェアの複数コミットからコードクローンを検出する場合，実用的な時間でコードクローン検出をすることが困難である．

これらの（問題 1）と（問題 2）を解決して，複数のコミットから素早く正確なコードクローン検出し，効率的にコードクローン管理支援を行うためには，CCVolti の改善が必要である．そこで本研究では，CCVolti をインクリメンタルコードクローン検出ができるように拡張することで，これらの問題を解決し，コードクローン検出結果の非一貫性の解決とスケーラビリティの向上を目指す．

CCVolti をインクリメンタルコードクローン検出ができるように拡張するには，1 番目のコミットで検出したコードクローンの情報として，ソースコード全体に存在するコードブロックの情報（位置情報や特徴ベクトルなど）と，互いにコードクローンと判定されたクローンペアのリストを保存し，2 番目のコミット以降の検出では，2 コミット間の差分を対象にコードクローン検出を行い，随時このコードクローン情報を更新する必要がある．

しかし，CCVolti をインクリメンタルコードクローン検出ができるように拡張するためには以下の問題を解決する必要がある．

（問題 3）TF-IDF はコミットごとに特徴ベクトルが変化

複数のコミットのソースコードからコードクローンを検出する場合，TF-IDF の IDF 値はコミットごと異なる．このため，コミット間で編集されていない同じコードブロックであっても，コードブロックの特徴ベクトルがコミットごと異なる．このため，2 つのコミット間で変更されたコードブロックのみを対象に特徴ベクトルに変換することはできず，コミットごとにすべてのコードクローン情報の再計算が必要となり，インクリメンタルにコードクローンを検出することはできない．したがって，CCVolti をインクリメンタルコードクローン検出ができるように拡張するためには，コードブロックが変更されない限り，コミットが変わっても特徴ベクトルは不変である必要がある．

3　提案手法

本節では，CCVolti をインクリメンタルコードクローン検出ができるように拡張する時に発生する問題の解決策とその解決策に基づいたインクリメンタルコードクローン検出手法を提案する．2.4 節では，以下の問題があることを述べた．

（問題 3）TF-IDF はコミットごとに特徴ベクトルが変化

（問題 3）を解決して，CCVolti をインクリメンタルコードクローン検出ができるように拡張するためには，2 つのコミット間で変更されないコードブロックの特徴ベクトルを変化させない必要がある．このため，以下の 2 つの解決策のどちらかでこの問題に対処する．

（解決策 1）IDF 値の固定　（以降，IDF 固定）

コミット間の差分が小さい場合は，IDF 値に大きな変化はない．このため，2 つのコミットの間で変更が少ない場合、固定した IDF 値を用いることで，インクリメンタルなコードクローン検出を行う．そして，IDF 値を固定したコミットから，出現するワードの種類数が設定した閾値以上増減した場合，IDF 値を再計算して，全コードブロックの特徴ベクトルを更新する．

（解決策 2）BoW (Bag of Words)　に変更　（以降，BoW 採用）

BoW[13] とは，ワードの出現回数に基づいたベクトル表現である．つまり，TF-IDF の TF 値のみを用いて，コード片を特徴ベクトルに変換する．既存研究 [15] で，CCVolti で TF-IDF を用いるよりも，BoW を用いた方が高い再現率が得られることが確認できた．また，2 つのコミット間で変更されていないコードブロックであれば，変換される特徴ベクトルは，コミットによらず不変にすることができる．このため，2 つのコミット間で変更されていないコードブロックの特徴ベクトルを再計算する必要がないため，以前のコミットで検出したコードクローンの情報を新コミットの検出で利用することができる．

3.1　検出手法の概要

本研究では，2.4 節で説明した CCVolti を拡張したインクリメンタルコードクローン検出手法を提案する．

本手法は，効率的にコードクローンを検出するために，1 番目のコミットで検出したコードクローン情報を 2 番目以降のコミットの検出で利用する．このため，1 番目のコミットの検出と 2 番目以降のコミットの検出で使用するコードクローン検出アルゴリズムが異なる．

1 番目のコミットと 2 番目以降のコミットの検出で使用するコードクローン検出アルゴリズムの概要をそれぞれ以下で説明する．

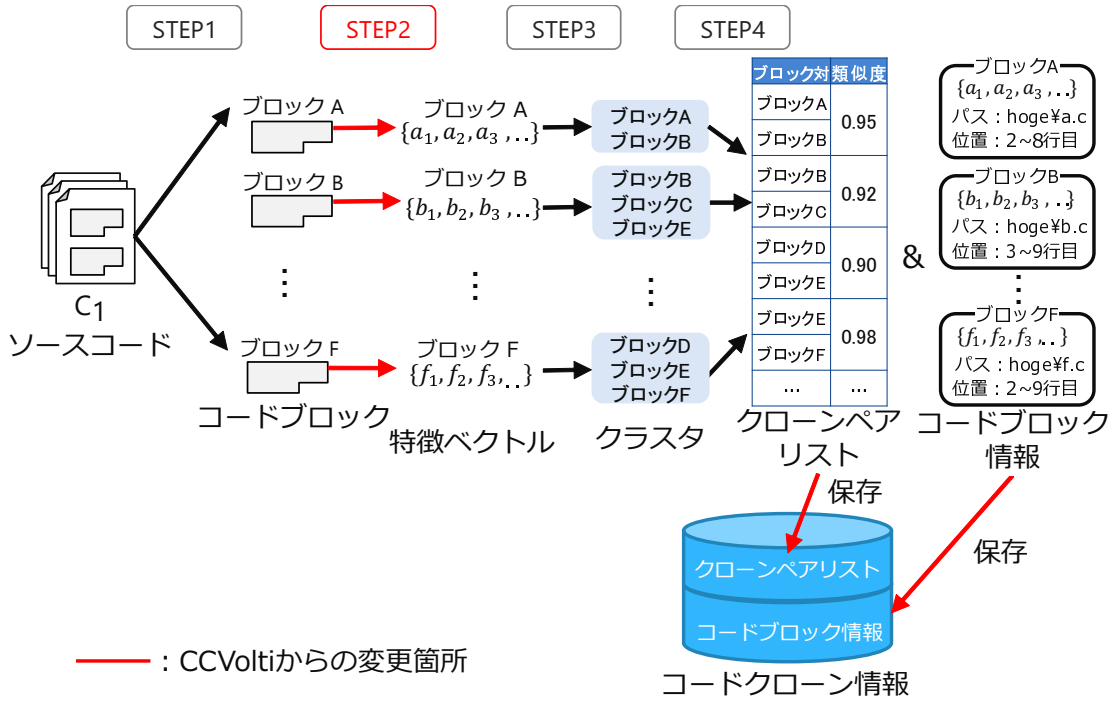


図 3: 1 番目のコミットで使用するコードクローン検出アルゴリズム

1 番目のコミットで使用するコードクローン検出アルゴリズム

1 番目のコミットで使用するコードクローン検出アルゴリズムを図 3 に示す．1 番目のコミットで使用するコードクローン検出アルゴリズムは，既存の CCVolti とほとんど同様のアルゴリズムを使用し，コードクローンを検出する．ただし，既存の CCVolti と違う点は大きく 2 つある．1 つ目は，2.4 節で説明した CCVolti のコードクローン検出アルゴリズムの STEP3 で使用されるベクトル表現手法が異なる．つまり、コードブロックを特徴ベクトルに変換する時に既存の TF-IDF ではなく，固定した IDF を用いた TF-IDF か BoW のどちらかを利用する．2 つ目は，ソースコード中の全コードブロックの情報（以降，コードブロック情報）と互いにクローンペアとなっているコードブロックの対のリスト情報（以降，クローンペアリスト）をコードクローン情報として保存することである．コードブロック情報に保存される内容は以下である．

- **ID** コードブロックを識別するための番号
- **位置情報** ファイル名，開始行，および終了行
- **特徴ベクトル** コードブロックをベクトル表現したもの
- **分類** 変更履歴による分類，1 コミット目の検出は NULL (詳細は 3.3 節に記述)

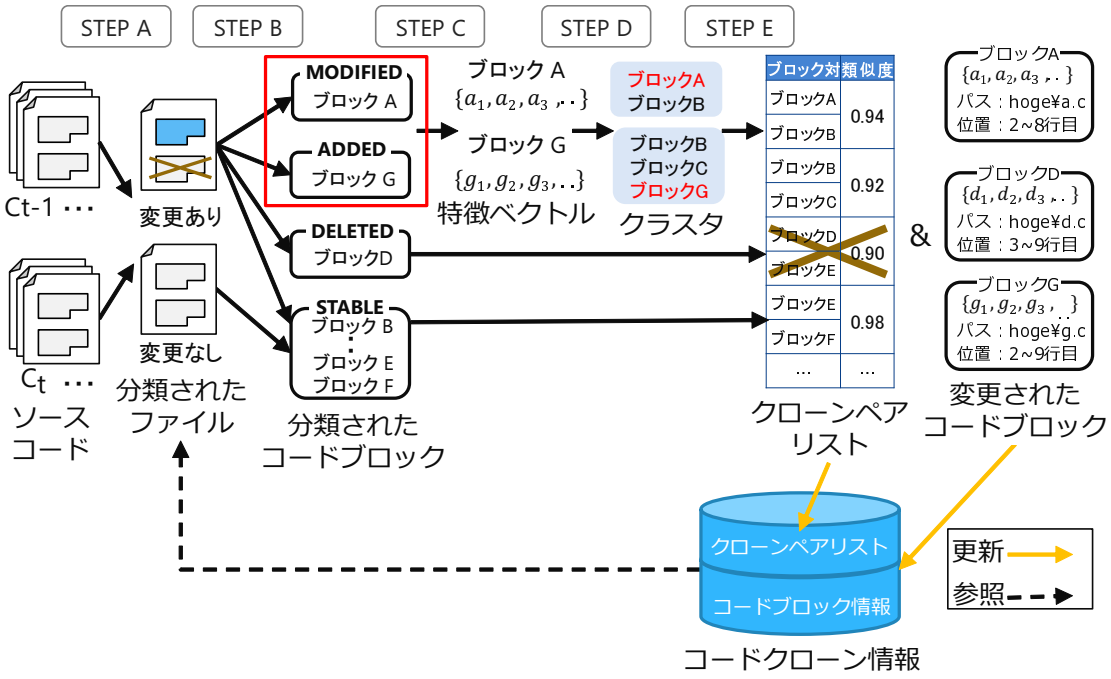


図 4: 2 番目以降のコミットで使用するコードクローン検出アルゴリズム

2 番目のコミット以降で使用するコードクローン検出アルゴリズム

2 番目以降のコミットで使用するコードクローン検出アルゴリズムを図 4 に示す．このコードクローン検出アルゴリズムは，主に以下の 5 つの STEP から構成される．

STEP A：差分の取得とソースファイルの分類

STEP B：コードブロックの対応付けと分類

STEP C：特徴ベクトルの計算

STEP D：LSH アルゴリズムを用いた特徴ベクトルのクラスタリング

STEP E：コードクローンの検出と情報の更新

最初に，STEP A で，2 コミット間のソースコードの差分を抽出し，ソースファイルの追加，編集，削除に基づき，ソースファイルを分類する．STEP B では，C_{t-1} のコミットのコードクローン検出で保存されているコードブロック情報とクローンペアリストを参照する．そして，編集または追加があったソースファイルに対して構文解析を行い，コードブロックを取得し，差分情報をもとに 2 つのコミット間のコードブロックを対応付ける．その後，追加，編集，削除といった変更履歴に基づきコードブロックを分類する．STEP C では，固定した IDF を用いた TF-IDF または BoW で追加，編集されたコードブロックのみを特徴ベクトルに変換する．STEP D では，LSH

アルゴリズムを用いた特徴ベクトルのクラスタリングによって，全コードブロックの特徴ベクトルの集合から，追加，編集されたコードブロックの特徴ベクトルと近似した特徴ベクトルの集合のクラスタを取得する．STEP E では，クラスタ内の特徴ベクトル間の類似度を計算することでクローンペアを検出し，C_t のコミットの検出で保存されたクローンペアリストを更新する．このとき，2 つコミット間で互いに変更されなかった 2 つのコードブロックが C_{t-1} のコミットの検出でも互いにコードクローンとして検出されているものは，C_t のコミットでもコードクローンとして検出する．また，2 つのコミット間で削除されたコードブロックを含むクローンペアがある場合は，C_t のコミットの検出で，そのクローンペアをクローンペアリストから削除する．そして，最後にコードクローン情報に保存されているクローンペアリストを C_t のコミットの検出結果をもとに更新し，コードブロック情報を 2 つのコミット間で変更されたコードブロックの情報をもとに更新する．

以降の節で，2 番目以降のコミットで使用するコードクローン検出アルゴリズムにおけるそれぞれの STEP の詳細について説明する．

3.2 STEP A：差分の取得とソースファイルの分類

2 番目以降のコミットで使用するコードクローン検出アルゴリズムの STEP A では，2 コミット間のソースコードの差分を抽出し，ソースファイルの追加，編集，削除に基づき，ソースファイルを分類する．本手法では，GNU diff[1] を用いて，2 コミット間のソースコードの差分を取得し，その差分情報に基づき，ソースファイルを以下の 4 つに分類する．なお，以下のソースファイルの分類は Clone Notifier[18] の分類方法を参考にした．

Stable ソースファイル

2 つのコミット間に渡って存在し，変更がないソースファイル

Added ソースファイル

新コミットのみに存在し，新コミットで新たに作成されたソースファイル

Modified ソースファイル

2 つのコミット間に渡って存在し，編集されたソースファイル

Deleted ソースファイル

旧コミットのみに存在し，新コミットで削除されたソースファイル

[1] http://www.gnu.org/software/diffutils/

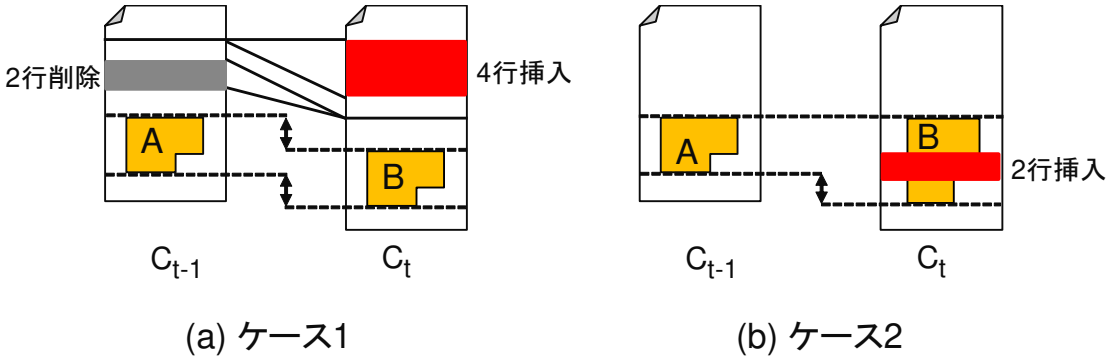


図 5: コードブロックの親子関係

3.3 STEP B：コードブロックの対応付けと分類

2 番目以降のコミットで使用するコードクローン検出アルゴリズムの STEP B では，C_{t-1} のコミットのコードクローン検出で保存されているコードブロック情報とクローンペアリストを参照する．そして，STEP A で分類された Added と Modified ソースファイルに対して構文解析を行い，コードブロックを取得し，差分情報をもとに 2 つのコミット間のコードブロックを対応付ける．その後，追加，編集，削除といった変更履歴に基づきコードブロックを分類する．

本手法は，文献 [19] で利用されている，親子関係に基づいたコードクローンの対応付け手法を用いて，2 つのコミット間の各コードブロックの対応付けを行う．本手法では，コードブロックの開始行と終了行に基づき，2 つのコミット間のコードブロックの対応付けを行う．

旧コミットのコードブロックの集合を CB_{t-1}，新コミットコードブロックの集合を CB_t とする．あるコードブロック $A \in CB_{t-1}$ に対応するコードブロックが $B \in CB_t$ である場合，コードブロック B をコードブロック A の子ブロック，コードクローン A をコードクローン B の親ブロックと定義する．図 5 にコードブロックの親子関係の例を示す．図 5(a) では，コードブロック $A \in CB_{t-1}$ の前で 4 行の挿入と 2 行の削除が行われている．このため A に対応するコードブロック B の開始行番号と終了行番号は，それぞれ A の開始行番号と終了行番号に 2 行追加した値となる．そして，B が CB_t に含まれる場合，コードブロック A の子ブロックはコードブロック B となる．また，図 5(b) では，コードブロック A の前で編集は行われていない．このため，A に対応するコードブロック B の開始行番号は A と同じである．しかし，A 中に 2 行の挿入が行われている．このために，B の終了行番号は A の終了行番号に 2 行追加した値となる．そして，B が CB_t に含まれる場合，コードブロック A の子ブロックはコードブロック B となる．このように，あるコードブロック $A \in CB_{t-1}$ に対

応するコードブロックをその開始行番号と終了行番号の対応に基づいて求め，そのコードブロック B が対応するコードブロックとなっている場合，コードブロック A とコードブロック B の間に親子関係を定義する．

2 つのコミット間の各コードブロックの対応付けを行なったあと，STEP A で取得した，追加，編集，削除といった変更履歴に基づき以下の 4 つにコードブロックを分類する．なお，以下のコードブロックの分類は Clone Notifier[18] のコードクローンの分類方法を参考にした．

Stable コードブロック

2 つのコミット間に渡って存在し，変更がないコードブロック．Stable ソースファイル内に存在するすべてのコードブロックは，自動的に Stable コードブロックに分類される．新コミットで，Stable コードブロックに分類されたものは，新コミットのコードクローン検出の対象にならない．ただし，コードブロックの開始行や終了行が変更された場合は，コードブロック情報を更新する．

Added コードブロック

新コミットのみに存在し，新コミットで新たに作成されたコードブロック．Added ソースファイル内に存在するすべてのコードブロックは，自動的に Added コードブロックに分類される．新コミットで Added　コードブロックに分類されたものは，新コミットのコードクローン検出の対象になる．

Modified コードブロック

2 つのコミット間に渡って存在し，新コミットで編集されたコードブロック．新コミットで Modified コードブロックに分類されたものは，新コミットのコードクローン検出の対象になる．

Deleted コードブロック

旧コミットのみに存在し，新コミットで削除されたコードブロック．Deleted ソースファイル内に存在するすべてのコードブロックは，自動的に Deleted コードブロックに分類される．新コミットで Deleted コードブロックに分類されたものは，新コミットのコードクローン検出の対象にならない．

次以降の STEP からは，Added と Modified コードブロックのみを対象にコードクローン検出が行われる．

3.4 STEP C：特徴ベクトルの計算

2 番目以降のコミットで使用するコードクローン検出アルゴリズムの STEP C では，固定した IDF を用いた TF-IDF または BoW を用いて，STEP B で分類された dded と Modified コードブロックを特徴ベクトルに変換する．

特徴ベクトルを計算する前に，コードブロックからワードを抽出する．本手法では，以下のいずれかを満たすものをワードとして定義する．

- 予約語
- 識別子名を構成する単語

識別子名が複数の単語から構成される場合，以下の方法でワード単位に分割する．

方法 1：ハイフンやアンダースコアなどの区切り記号による分割

方法 2：識別子名中の大文字になっているアルファベットによる分割

また，2 文字以下の識別子は，それらをまとめて同一のメタワードとして扱う．この理由として，例えば，繰り返し処理等で頻繁に利用される i や j といった変数は，意味情報が込められていない変数として扱うためである．さらに，条件分岐に用いられる if や switch，繰り返し処理に用いられる for や while 等の予約語もワードとして扱う．加えて，各ワードの大文字と小文字による区別はつけず，同一のワードとして扱う．

コードブロックからワードを抽出したあと，ワードに対して，IDF 固定または BoW 採用のどちらかを用いて，コードブロックを特徴ベクトルに変換する．以下に IDF 固定と BoW 採用の特徴ベクトルの計算について述べる．

IDF 固定

コードブロックを特徴ベクトルに変換する時に使用する TF-IDF の IDF 値を，異なるコミット間で固定して，TF-IDF 値を計算し，コードブロックを特徴ベクトルに変換する．そして，あるコミットで以下の条件を満たす場合に，IDF 値を再計算して，ソースコード中の全コードブロックの特徴ベクトルを再計算する．

条件：IDF 値を固定した旧コミットに比べて，出現するワードの種類数が W%増減

例えば，上記の条件の W を 10 とし，任意のコミット C_{t-1} で出現するワードの数を 100 とする．C_t 以降のコミットで出現するワード種類数が，90 または，110 になったコミットで，IDF 値を再計算して，ソースコード中の全コードブロックの特徴ベクトルを再計算する．その後は，IDF 値を再計算したコミットを基準として，出現するワードの種類数をカウントしていき，W%増減したコミットで，IDF 値を再計算をして，全コードブロックの特徴ベクトルを計算する．

BoW 採用

BoW[13] とは，ワードの出現回数に基づいたベクトル表現である．つまり，TF-IDF の TF 値のみを用いて，コードブロックを特徴ベクトルに変換する．本手法では，コードブロック Y 内におけるワード X の BoW は以下のように計算する．

$$\mathrm{BoW}_{X,Y} = \frac{\text{コードブロック } Y \text{ 内におけるワード } X \text{ の出現回数}}{\text{コードブロック } Y \text{ 内における全ワードの出現回数の和}} \tag{3}$$

つまり上記の式を用いて，コードブロック中のワードの出現頻度を特徴量として，各コードブロックを特徴ベクトルに変換する．

また，本手法では，ベクトルの次元数を 1 番目のコミットに出現するワードの種類数に加えて余分に m 次元確保する．これは，複数のコミットからコードクローンの検出を行う本手法で，新たに出現したワードの種類を考慮して特徴ベクトルを計算するためである．本手法では，インクリメンタルにコードクローンを検出するために，コードブロックが変更されない限り，コミットが変わっても特徴ベクトルは不変である必要がある．このため，2 番目のコミット以降のコードクローン検出では，ベクトルの次元数を 1 番目のコミットのコードクローン検出で使用したベクトルの次元数からから変更することはできない．そこで，本手法では，1 番目のコミットのコードクローン検出で使用するベクトルの次元数を 1 番目のコミットに出現するワードの種類数に加えて余分に m 次元確保し，余分に確保した次元は 0 とする．そして，2 番目以降ののコミットのコードクローン検出においても，1 番目のコミットのコードクローン検出で使用したベクトルの次元数を使用し，コミットごとに新たなワードが出現しても余分に確保した次元に充てる．本手法は，このように特徴ベクトルを計算することで，2 番目以降のコミットで新たな種類のワードが出現しても対応することができる．

3.5 STEP D：LSH アルゴリズムを用いた特徴ベクトルのクラスタリング

2 番目以降のコミットで使用するコードクローン検出アルゴリズムの STEP D では，特徴ベクトルのクラスタリングによって，C_t 番目のコミットで出現する全コードブロックの特徴ベクトルの集合のうち，C_t 番目のコミットで追加，編集されたコードブロックの特徴ベクトルと近似した特徴ベクトルの集合のクラスタを取得する．これにより，C_t 番目のコミットでクローンペアとなる可能性のある候補を高速に絞り込む．

本手法では，特徴ベクトルのクラスタリングを行う際に，近似最近傍探索アルゴリズムの Cross-Polytope LSH[1] を用いる．Cross-Polytope LSH とは，距離が近いベクトル同士は高い確率で衝突する特徴を持つハッシュ関数を用いて，高次元なベクトル集合をハッシュ化して，近似的に近傍点を求める近似最近傍探索アルゴリズムである局所性鋭敏型ハッシュ(LSH) の一種である．また，ハッシュ関数に対して，2 つのベクトル x, y が同じハッシュ値を取る

ことを衝突という．2 つのベクトル x, y に対して類似度 $S(x, y)$ が定義された d 次元空間上において，x, y のハッシュ値が衝突する確率を衝突確率と呼ぶ．本手法では，Cross-Polytope LSH を利用するために，LSH ライブラリ FALCONN[2][2] を使用した．LSH アルゴリズムは，(p_1, p_2, r, c)-Sensitive Hashing と (r, c)-Approximate Neighbor を用いたクラスタリング手法である．このアルゴリズムを利用することによって，クエリとして 1 つの特徴ベクトルを与えると，特徴ベクトル集合からそのクエリと近似した特徴ベクトル集合のクラスタを取得することができる．(p_1, p_2, r, c)-Sensitive Hashing と (r, c)-Approximate Neighbor の定義を以下に示す．

(p_1, p_2, r, c)-Sensitive Hashing

実数 $p_1, p_2 (0 \leq p_1, p_2 \leq 1)$，実数 $r (0 < r)$，実数 $c (1 < c)$，R^d 空間に対するベクトル集合 V 上の任意の点 v_i, v_j が与えられたとき，

$$\begin{cases} \text{if } \ D(v_i, v_j) < r \quad then \ \ Prob[h(v_i) = h(v_j)] > p_1 \\ \text{if } \ D(v_i, v_j) > cr \ \ then \ \ Prob[h(v_i) = h(v_j)] < p_2 \end{cases}$$

を満たすハッシュ関数 h を (p_1, p_2, r, c)-Sensitive Hashing と呼ぶ．ここで，D は R^d 上の距離関数，$Prob$ は条件式が真となる確率を表している．

(r, c)-Approximate Neighbor

R^d 空間に対するベクトル集合 V 上の任意の点（クエリ）v が与えられたとき，

$$U = \{u \in V | D(v, u) \leq cr\}$$

で定義される V の部分集合 U をクエリ v に対する (r, c)-Approximate Neighbor と呼ぶ．ここで，c と r の定義は (p_1, p_2, r, c)-Sensitive Hashing で利用した定義と共通である．

R^d 空間に対する特徴ベクトル集合 V が与えられ，(p_1, p_2, r, c)-Sensitive Hashing を用いた関数の族 $h_{l,k} (1 \leq l \leq L, 1 \leq k \leq K)$ からなる関数 H_l が以下の式で与えられたとき，

$$H_l = (h_{l,1}(v), h_{l,2}(v), ..., h_{l,K}(v)) \in R^K$$

V 上の任意の点 v に対して L 個の K 次元のベクトルが得られる．LSH アルゴリズムでは，これらのベクトルをそれぞれハッシュテーブルの鍵とすることで高速に近傍点を求めることができる．

[2]https://falconn-lib.org/

本手法では，LSH アルゴリズムを利用することで，クエリとして新コミットで追加，編集されたコードブロックの特徴ベクトルを与えると，新コミットのソースコード中に存在する全コードブロックの特徴ベクトル集合から，新コミットで追加，編集されたコードブロックの特徴ベクトルと近似した特徴ベクトルの集合のクラスタを取得することができる．

3.6 STEP E：コードクローンの検出と情報の更新

2 番目以降のコミットで使用するコードクローン検出アルゴリズムの STEP E では，STEP D で取得したクラスタ内の特徴ベクトル間の類似度を計算することでクローンペアを検出し，以前のコミットで保存されたクローンペアリストを更新する．

本手法では，コサイン類似度が閾値以上のコードブロックのペアをクローンペアとして検出する．コサイン類似度は，多次元ベクトルの類似度を表す尺度である．次元が D である 2 つの特徴ベクトル $\vec{a}, \vec{b}$ 間の類似度は，以下の式で与えられる．

$$\mathrm{sim}(\vec{a}, \vec{b}) = \frac{\sum_{i=1}^{|D|} a_i b_i}{\sqrt{\sum_{i=1}^{|D|} a_i^2}\sqrt{\sum_{i=1}^{|D|} b_i^2}} \tag{4}$$

コードブロックを特徴ベクトルに変換するとき，ベクトル表現には，IDF 固定と BoW 採用を使用するため，特徴量は常に正の値になるため，コサイン類似度は 0 から 1 の範囲である．コードブロックのペアのコサイン類似度が閾値以上であれば，それらをクローンペアとして検出し，以前のコミットのコードクローン検出で保存されたクローンペアリストを更新する．

また，2 つコミット間で互いに変更されなかった 2 つのコードブロックが C_{t-1} のコミットの検出でも互いにコードクローンとして検出されているものは，C_t のコミットでもコードクローンとして検出する．さらに，2 つのコミット間で削除されたコードブロックを含むクローンペアがある場合は，C_t のコミットの検出で，そのクローンペアをクローンペアリストから削除する．

4　評価実験

本章では，本研究で提案したインクリメンタルコードクローン検出手法の評価実験について述べる．評価実験では，本手法の有用性を確認するために，本手法を含めた 6 つのコードクローン検出ツールを，2 つの C 言語プロジェクトと 2 つの Java プロジェクトに対して適用し，検出精度と検出時間の観点で比較評価を行った．

評価実験の方法を以下に示す．

評価対象のコードクローン検出ツール

評価実験では，TF-IDF を用いたコードクローン検出ツール 3 つと，BoW を用いたコードクローン検出ツール 3 つ，合計 6 つのツールを評価対象とした．各コードクローン検出ツールについて，表 1 に示す．

EX DIM はベクトルの次元数を余分に確保したコードクローン検出ツールである．EX DIM を評価対象のコードクローン検出ツールに選択したのは，ベクトルの次元数を余分に確保することによる検出結果の違いを確認するためである．EX DIM のベクトルの次元数は以下の式の通りに確保した．

$$\text{EX DIM のベクトルの次元数} = \text{1 番目のコミットに出現するワードの種類数} + 15000 \tag{5}$$

本手法も 1 番目のコミットのコードクローン検出では，上記の式の通りにベクトルの次元数を確保し，2 番目以降のコミットでも 1 番目のコミットで確保したベクトルの次元数でコードブロックを特徴ベクトルに変換する．このため，1 番目のコミットのコードクローン検出では，CCVolti(TF-IDF EX DIM) と本手法（IDF 固定），CCVolti(BoW EX DIM) と本手法（BoW 採用）のそれぞれのコードクローン検出ツールで計算される特徴ベクトルは完全に同じである．したがって，CCVolti(TF-IDF EX DIM) と本

表 1: 対象のコードクローン検出ツール

ツール名	説明
CCVolti (TF-IDF)	既存の CCVolti でベクトル表現は TF-IDF を利用
CCVolti (TF-IDF EX DIM)	TF-IDF を利用かつベクトルの次元数を余分に確保
本手法 (IDF 固定)	IDF 値を固定して本手法を実現 詳細は 3 章に記述
CCVolti (BoW)	既存の CCVolti でベクトル表現は BoW を利用
CCVolti (BoW EX DIM)	BoW を利用かつベクトルの次元数を余分に確保
本手法 (BoW 採用)	BoW を採用して本手法を実現 詳細は 3 章に記述

手法（IDF 固定），CCVolti(BoW EX DIM) と本手法（BoW 採用）のそれぞれコードクローン検出ツールが検出するクローンペアは完全に同じであると考えられる．

また，6 つのコードクローン検出ツールのパラメータは以下の通り設定した．

- コサイン類似度 0.9 以上（TF-IDF を用いた既存研究 [15] で高い精度）
- 最少トークン数 50

なお，本手法 (IDF 固定) の IDF 値の再計算条件は，IDF 値を固定した旧コミットに比べて，出現するワードの種類数が 5%増減する場合，そのコミット以降で使用する IDF 値を再計算するように設定した．

適合率の算出

適合率（以降，図では略語として P を用いる）とは，検出結果の中にどの程度正解が含まれるかその割合のことである．本実験では，コードクローンの検出結果からランダムに任意の個数のクローンペアを抽出して，検出されたクローンペアがコードクローンであるか目視で判別する．

評価実験対象プロジェクト

本実験で評価実験対象に選択したプロジェクトの一覧を表 2 に示す．この表からわかるように，評価実験の対象のプロジェクトとして，ソースコードの規模が大小異なる C 言語の 2 つのプロジェクトと，Java の 2 つプロジェクトを選択した．また，評価実験対象プロジェクトのメインブランチにあるコミットの内，任意の日時のコミットから，1 日間隔で対象言語のソースファイルに差分がある 1000 個のコミットを抽出し，そのコミットを対象にコードクローンを検出をした．

実行環境

以下の実行環境で評価実験を行った．

CPU Xeon E5-2690 2.90GHz 8 コア× 2

ヒープサイズ 2GB

以降，各コードクローン検出ツールの検出結果と検出時間を比較した結果について説明する．

4.1 検出結果の比較評価

本節では，各コードクローン検出ツールが出力する検出結果を比較した結果について述べる．以下に，本手法と比較対象のコードクローン検出ツールが評価実験対象プロジェクトの 1000 個のコミットからコードクローンを検出した結果をプロジェクトごとに説明する。

表 2: 検出対象プロジェクト

プロジェクト名	期間と検出コミット数	規模と出現するワードの種類数	
		1 番目のコミット	1000 番目のコミット
Redis （C 言語）	2014/01/01〜2019/10/19 1 日間隔で 1000 コミット	LOC：9 万行 ファイル数：236 ワード数：2449	LOC：18 万行 ファイル数：476 ワード数：3598
PosgreSQL （C 言語）	2014/01/01〜2017/02/10 1 日間隔で 1000 コミット	LOC：101 万行 ファイル数：1678 ワード数：10211	LOC：120 万行 ファイル数：1920 ワード数：11565
Apache Ant （Java）	2001/01/01〜2005/04/08 1 日間隔で 1000 コミット	LOC：8 万行 ファイル数：449 ワード数：1014	LOC：24 万行 ファイル数：1168 ワード数：2162
WildFly （Java）	2015/01/01〜2018/08/15 1 日間隔で 1000 コミット	LOC：68 万行 ファイル数：6387 ワード数：3376	LOC：88 万行 ファイル数：8431 ワード数：3768

Redis

Redis プロジェクトの 1 番目と 1000 番目のそれぞれのコミットから，各コードクローン検出ツールが出力する検出結果まとめたものを表 3 に示す．また，TF-IDF を用いたツールと BoW を用いたツールが 1 番目のコミットと 1000 番目のコミットで検出するクローンペアの関係を示したベン図をそれぞれ図 6 と図 7 に示す．

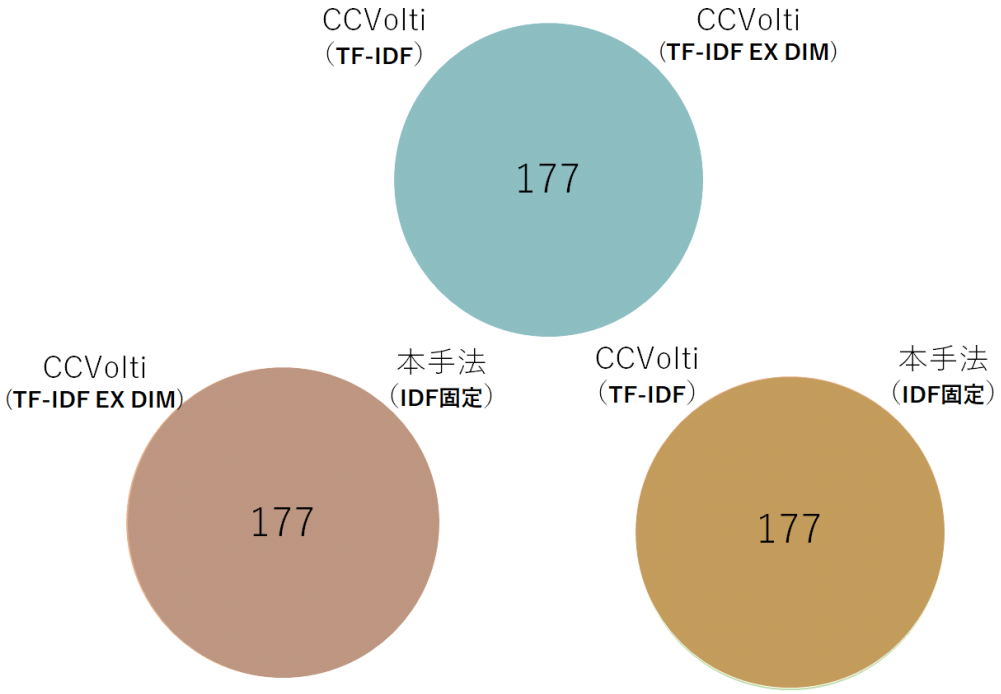
表 3 と図 6 からわかるように，1 番目のコミットのコードクローン検出において，TF-IDF を用いてコードブロックから特徴ベクトルを変換する 3 つのコードクローン検出ツール（CCVolti(TF-IDF), CCVoitl(TF-IDF EX DIM, 本手法（IDF 固定））が検出するクローンペアに違いはなかった．また、BoW を用いるコードクローン検出ツール（CCVolti(BoW), CCVoitl(BoW EX DIM), 本手法（BoW 採用））が検出するクローンペアに違いはなかった． したがって，Redis プロジェクトのコードクローン検出では，ベクトルの次元数を余分に確保することによって検出するクローンペアに違いがあった．

表 3 と図 7 からわかるように，1000 番目のコミットのコードクローン検出においては，本手法（IDF 固定）は，同じく特徴ベクトル変換に TF-IDF を用いる CCVolti(TF-IDF) と CCVolti(TF-IDF EX DIM) とは，検出するクローンペアが異なり，本手法（IDF 固定）は，CCVolti（TF-IDF）と CCVolti（TF-IDF EX DIM）の検出するク

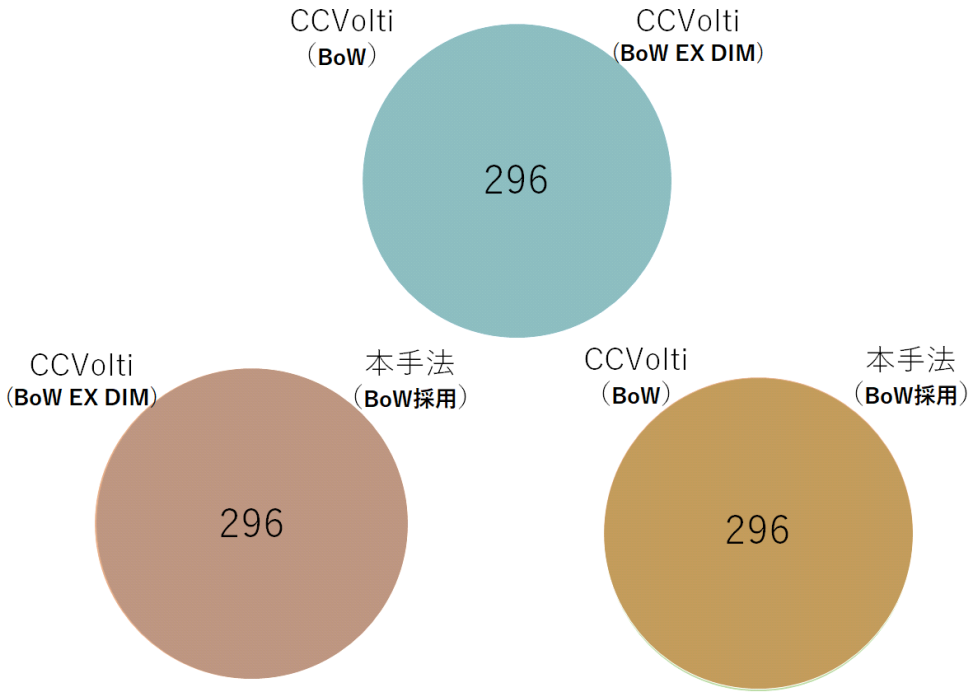
ローンペアの約 92.3%を検出できる上に，より多くのクローンペアを高い適合率で検出した．また，本手法（BoW 採用）は，同じく特徴ベクトル変換に BoW を用いる CCVolti(BoW) と CCVolti(BoW EX DIM) とは，検出するクローンペアが異なり，本手法（BoW 採用）は，CCVolti（BoW）と CCVolti（BoW EX DIM）の検出するクローンペアの 100.0%を検出できる上に，より多くのクローンペアを高い適合率で検出した．

表 3: Redis から各コードクローン検出ツールが検出した結果

	1 番目のコミットの検出結果		1000 番目のコミットの検出結果	
	クローンペア数	適合率	クローンペア数	適合率
CCVolti (TF-IDF)	177	0.97 (172/177)	498	0.99 (296/300)
CCVolti (TF-IDF EX DIM)	177	0.97 (172/177)	498	0.99 (296/300)
本手法 (IDF 固定)	177	0.97 (172/177)	522	0.98 (294/300)
CCVolti (BoW)	296	0.96 (284/296)	710	0.99 (297/300)
CCVolti (BoW EX DIM)	296	0.96 (284/296)	710	0.99 (297/300)
本手法 (BoW 採用)	296	0.96 (284/296)	740	0.99 (298/300)

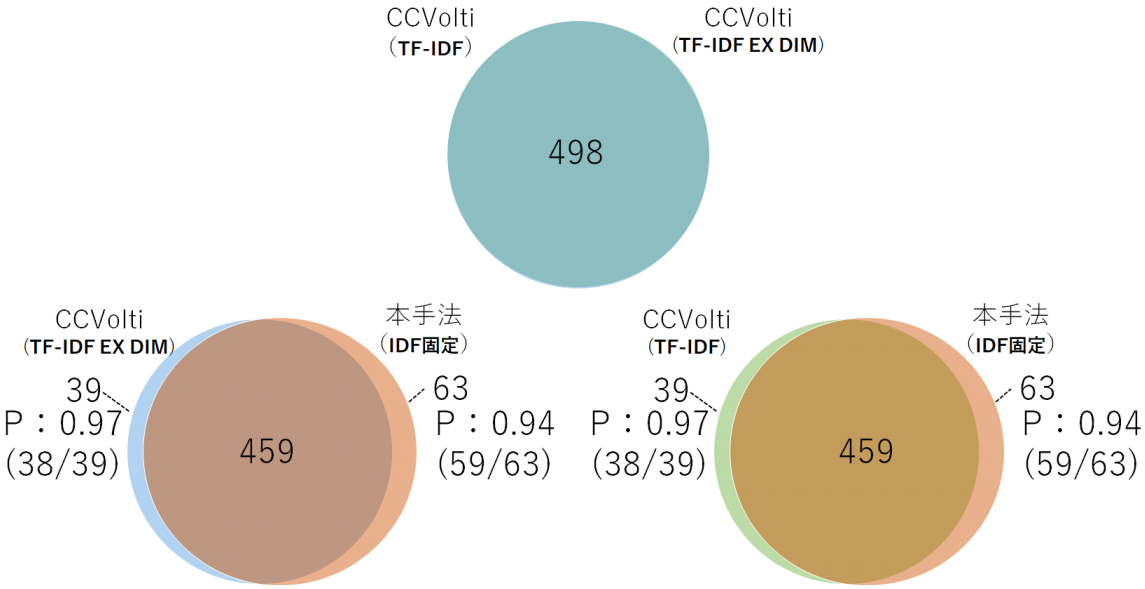


(a) TF-IDF を用いたツール

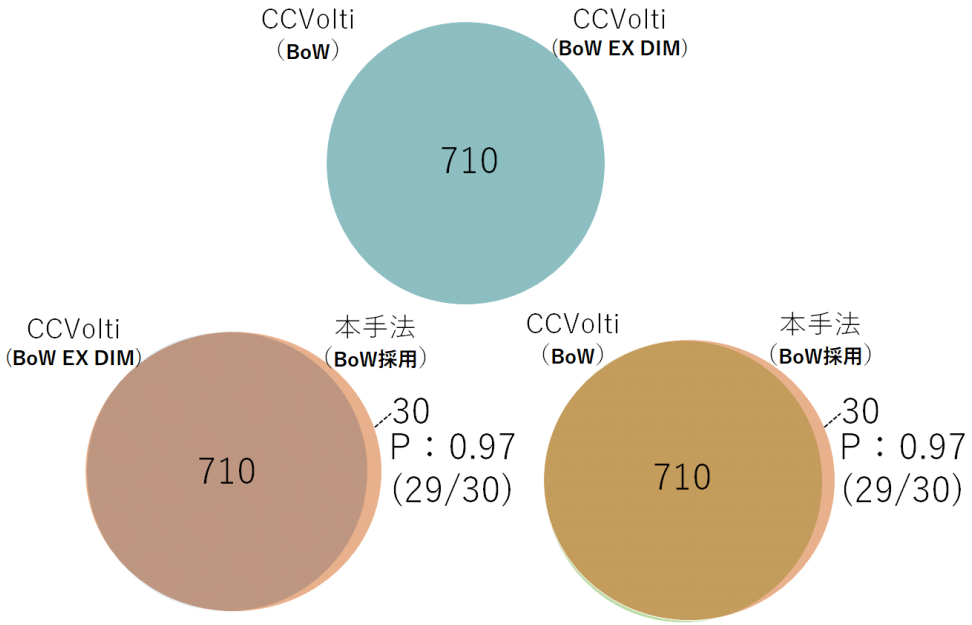


(b) BoW を用いたツール

図 6: 1 番目のコミットの検出で各ツールが出力するクローンペアの関係（Redis）



(a) TF-IDF を用いたツール



(b) BoW を用いたツール

図 7: 1000 番目のコミットの検出で各ツールが出力するクローンペアの関係（Redis）

PostgreSQL

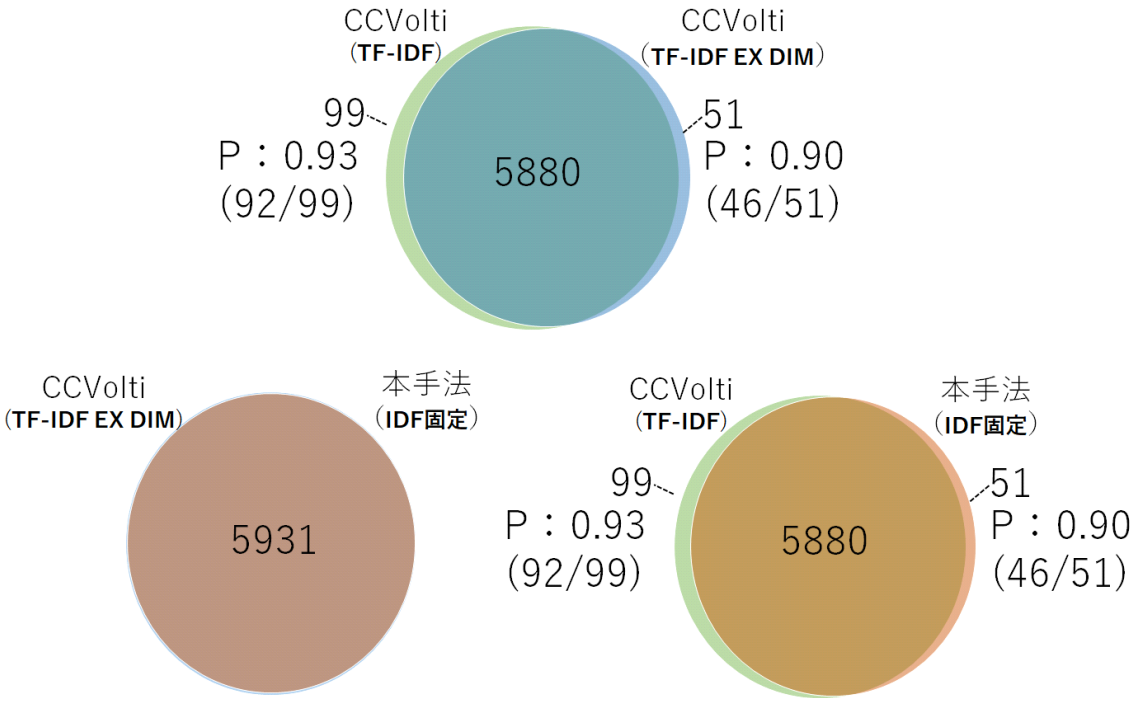
PostgreSQL プロジェクトの 1 番目と 1000 番目のそれぞれのコミットから，各コードクローン検出ツールが出力する検出結果まとめたものを表 4 に示す．また，TF-IDF を用いたツールと BoW を用いたツールが 1 番目のコミットと 1000 番目のコミットで検出するクローンペアの関係を示したベン図をそれぞれ図 8 と図 9 に示す．

表 4 と図 8 からわかるように，1 番目のコミットのコードクローン検出において，TF-IDF を用いるコードクローン検出ツール CCVolti（TF-IDF）と CCVolti（TF-IDF EX DIM）が検出するクローンペアに違いがあった．また、BoW を用いるコードクローン検出ツール CCVolti（BoW）と CCVolti（BoW EX DIM）が検出するクローンペアに違いがあった． したがって，PostgreSQL プロジェクトのコードクローン検出では，ベクトルの次元数を余分に確保することによって検出するクローンペアに違いがあった．

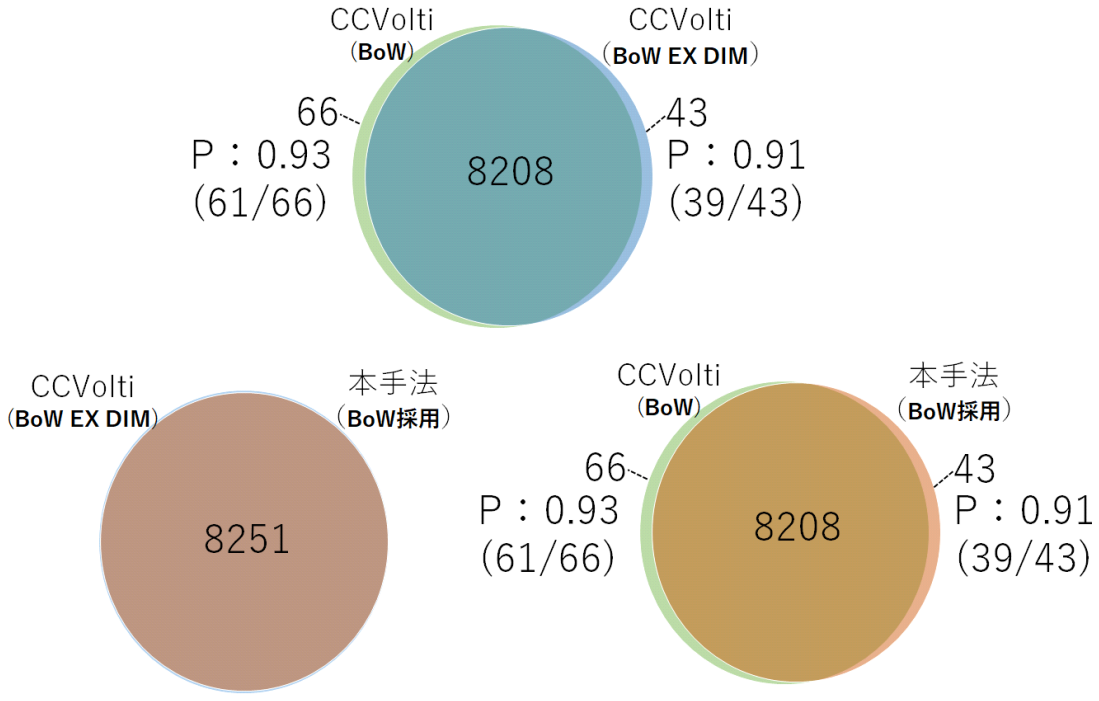
表 4 と図 9 からわかるように，1000 番目のコミットのコードクローン検出においては，本手法（IDF 固定）は，同じく特徴ベクトル変換に TF-IDF を用いる CCVolti(TF-IDF) とは，検出するクロ―ンペアが異なり，本手法（IDF 固定）は，CCVolti（TF-IDF）の検出するクローンペアの約 97.0%を検出できる上に，より多くのクローンペアを高い適合率で検出した．また，本手法（BoW 採用）は，同じく特徴ベクトル変換に BoW を用いる CCVolti(BoW) とは，検出するクローンペアが異なり，本手法（BoW 採用）は，CCVolti（BoW）との検出するクローンペアの約 98.8%を検出できる上に，より多くのクローンペアを高い適合率で検出した．

表 4: PosgreSQL の検出において各ツールが出力する検出結果

	1 コミット目		1000 コミット目	
	クローンペア数	適合率	クローンペア数	適合率
CCVolti (TF-IDF)	5979	0.97 (290/300)	6424	0.97 (291/300)
CCVolti (TF-IDF EX DIM)	5931	0.97 (290/300)	6421	0.97 (290/300)
本手法 (IDF 固定)	5931	0.97 (290/300)	6666	0.98 (293/300)
CCVolti (BoW)	8274	0.97 (292/300)	8943	0.97 (290/300)
CCVolti (BoW EX DIM)	8251	0.98 (293/300)	8897	0.97 (290/300)
本手法 (BoW 採用)	8251	0.98 (293/300)	9274	0.98 (293/300)

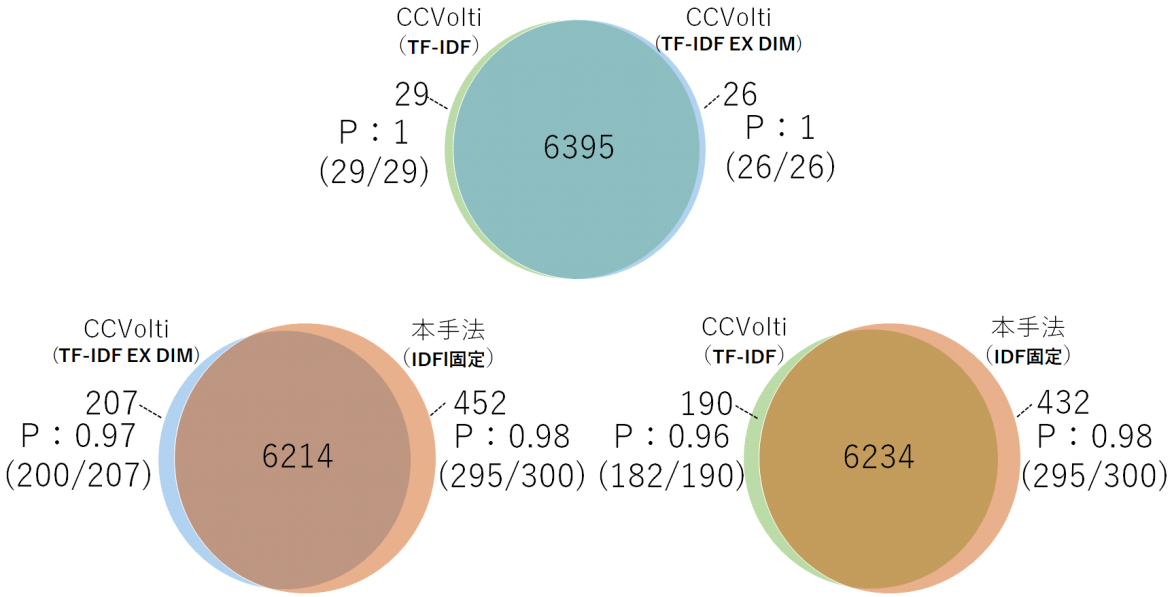


(a) TF-IDF を用いたツール

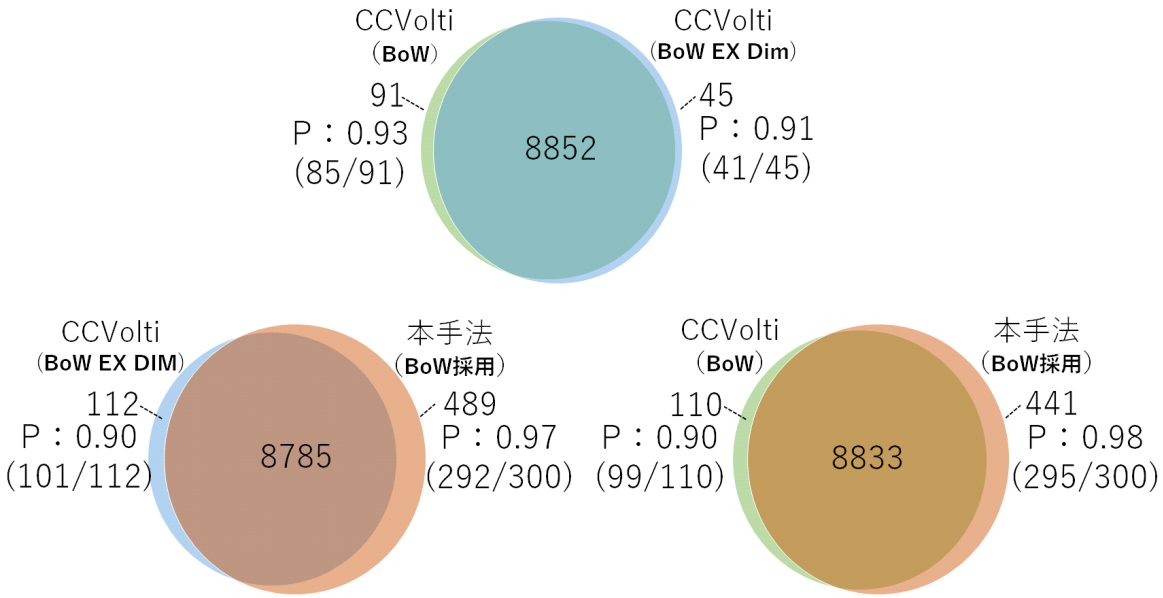


(b) BoW を用いたツール

図 8: 1 番目のコミットの検出で各ツールが出力するクローンペアの関係（PostgreSQL）



(a) TF-IDF を用いたツール



(b) BoW を用いたツール

図 9: 1000 番目のコミットの検出で各ツールが出力するクローンペアの関係（PostgreSQL）

Apache Ant

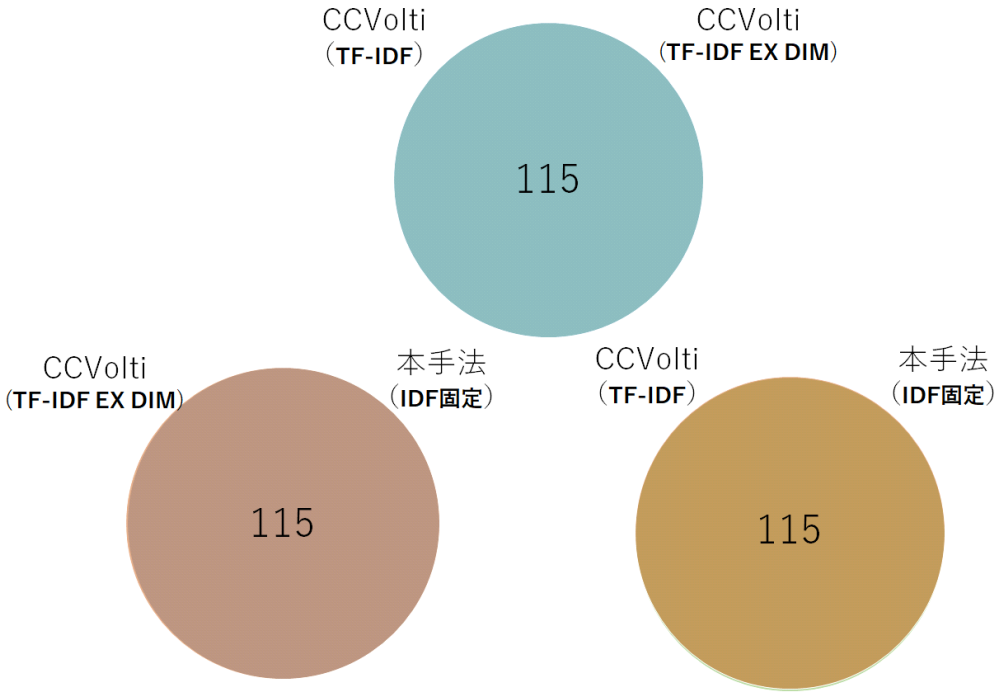
Apache Ant プロジェクトの 1 番目と 1000 番目のそれぞれのコミットから，各コードクローン検出ツールが出力する検出結果まとめたものを表 5 に示す．また，TF-IDF を用いたツールと BoW を用いたツールが 1 番目のコミットと 1000 番目のコミットで検出するクローンペアの関係を示したベン図をそれぞれ図 10 と図 11 に示す．

表 5 と図 10 からわかるように，1 番目のコミットのコードクローン検出において，TF-IDF を用いてコードブロックから特徴ベクトルを変換する 3 つのコードクローン検出ツール（CCVolti(TF-IDF), CCVoitl(TF-IDF EX DIM, 本手法（IDF 固定)）が検出するクローンペアに違いはなかった．また、BoW を用いるコードクローン検出ツール（CCVolti(BoW), CCVoitl(BoW EX DIM), 本手法（BoW 採用)）が検出するクローンペアに違いはなかった． したがって，Redis プロジェクトのコードクローン検出では，ベクトルの次元数を余分に確保することによって検出するクローンペアに違いがあった．

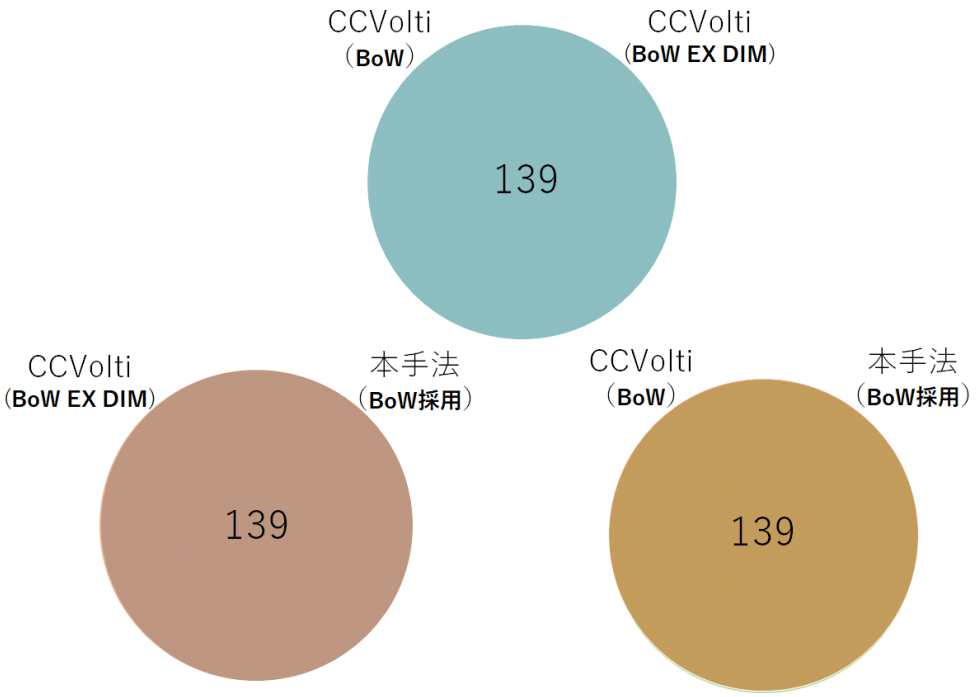
表 5 と図 11 からわかるように，1000 番目のコミットのコードクローン検出においては，本手法（IDF 固定）は，同じく特徴ベクトル変換に TF-IDF を用いる CCVolti(TF-IDF) と CCVolti(TF-IDF EX DIM) とは，検出するクロ―ンペアが異なり，本手法（IDF 固定）は，CCVolti（TF-IDF）と CCVolti（TF-IDF EX DIM）の検出するクローンペアの約 94.0%を検出できる上に，より多くのクローンペアを高い適合率で検出した．また，本手法（BoW 採用）は，同じく特徴ベクトル変換に BoW を用いる CCVolti(BoW) と CCVolti(BoW EX DIM) とは，検出するクローンペアが異なり，本手法（BoW 採用）は，CCVolti（BoW）と CCVolti（BoW EX DIM）の検出するクローンペアの約 99.7%を検出できる上に，より多くのクローンペアを高い適合率で検出した．

表 5: Apache Ant の検出において各ツールが出力する検出結果

	1 コミット目		1000 コミット目	
	クローンペア数	適合率	クローンペア数	適合率
CCVolti (TF-IDF)	115	1 (115/115)	1169	1 (300/300)
CCVolti (TF-IDF EX DIM)	115	1 (115/115))	1169	1 (300/300)
本手法 (IDF 固定)	115	1 (115/115)	1234	1 (300/300)
CCVolti (BoW)	139	1 (139/139)	1355	1 (300/300)
CCVolti (BoW EX DIM)	139	1 (139/139)	1355	1 (300/300)
本手法 (BoW 採用)	139	1 (139/139)	1480	1 (300/300)

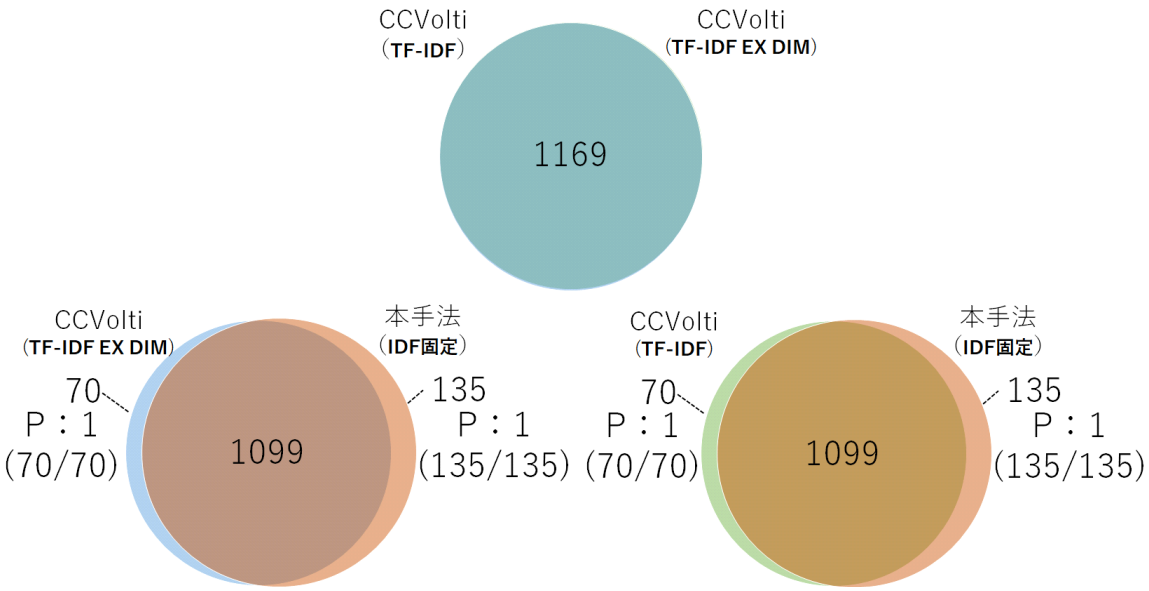


(a) TF-IDF を用いたツール

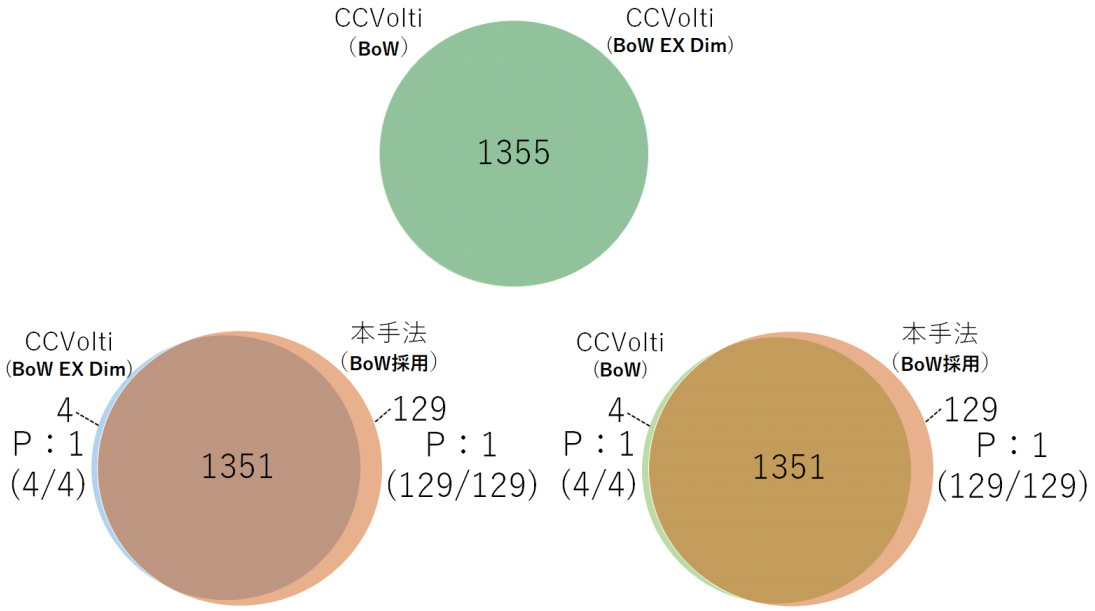


(b) BoW を用いたツール

図 10: 1 番目のコミットの検出で各ツールが出力するクローンペアの関係（Apache Ant）



(a) TF-IDF を用いたツール



(b) BoW を用いたツール

図 11: 1000 番目のコミットの検出で各ツールが出力するクローンペアの関係（Apache Ant）

WildFly

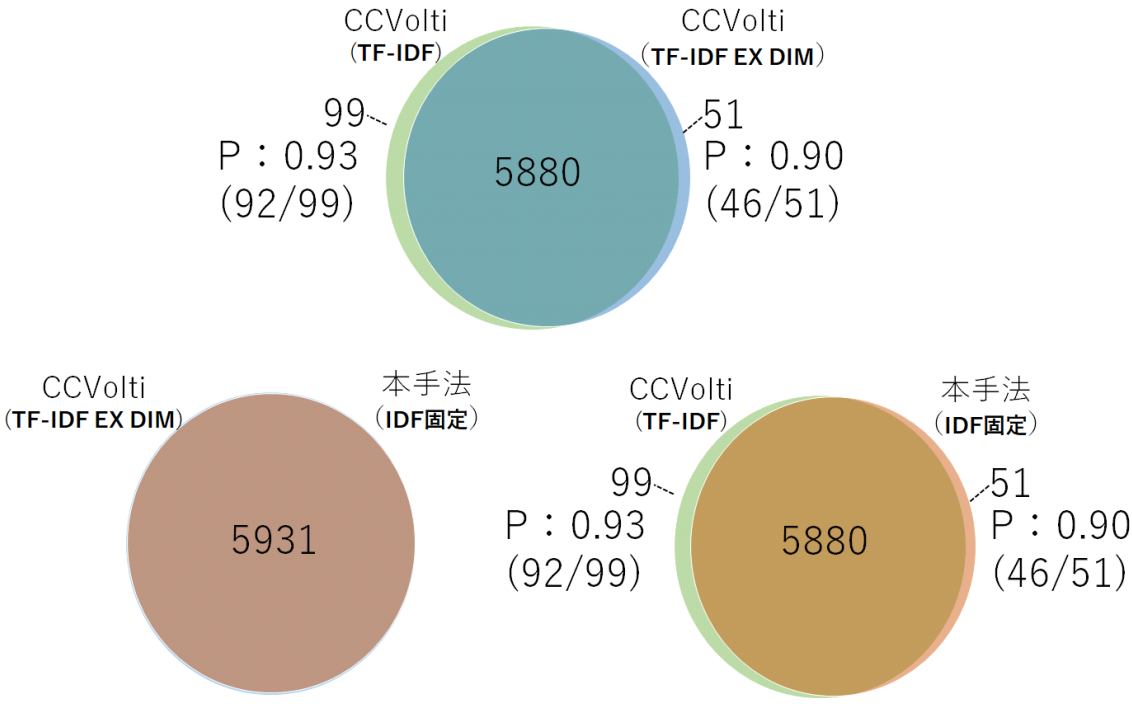
WildFly プロジェクトの 1 番目と 1000 番目のそれぞれのコミットから，各コードクローン検出ツールが出力する検出結果まとめたものを表 6 に示す．また，TF-IDF を用いたツールと BoW を用いたツールが 1 番目のコミットと 1000 番目のコミットで検出するクローンペアの関係を示したベン図をそれぞれ図 12 と図 13 に示す．

表 6 と図 12 からわかるように，1 番目のコミットのコードクローン検出において，TF-IDF を用いるコードクローン検出ツール CCVolti（TF-IDF）と CCVolti（TF-IDF EX DIM）が検出するクローンペアに違いがあった．また、BoW を用いるコードクローン検出ツール CCVolti（BoW）と CCVolti（BoW EX DIM）が検出するクローンペアに違いがあった． したがって，PostgreSQL プロジェクトのコードクローン検出では，ベクトルの次元数を余分に確保することによって検出するクローンペアに違いがあった．

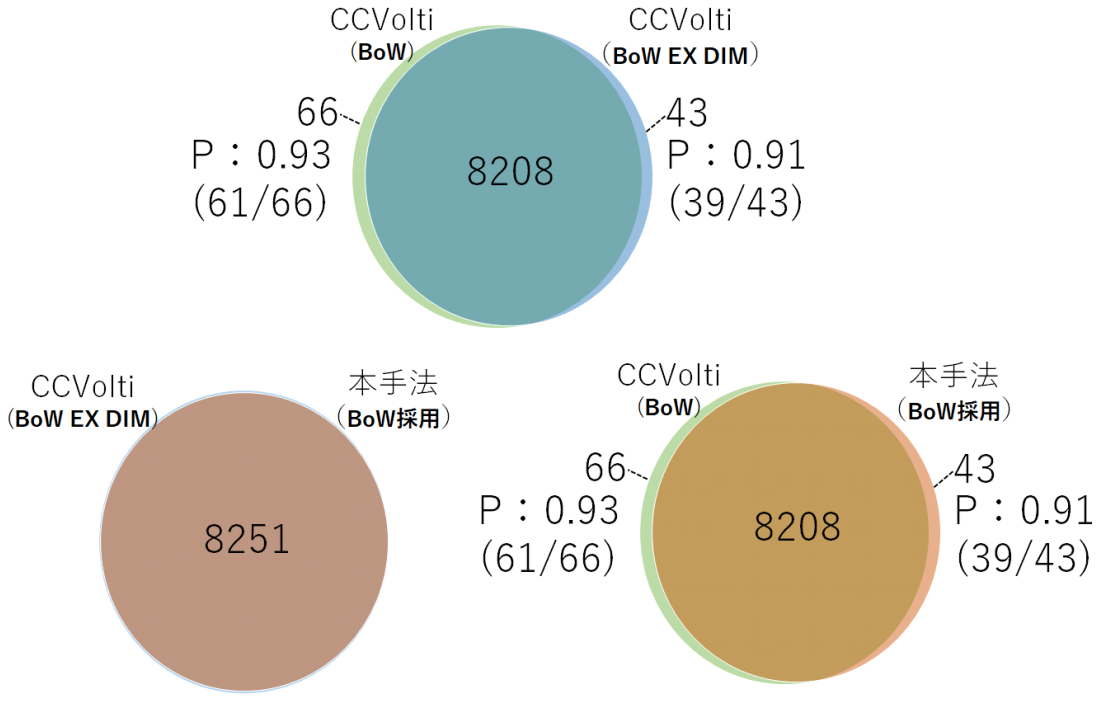
表 6 と図 13 からわかるように，1000 番目のコミットのコードクローン検出においては，本手法（IDF 固定）は，同じく特徴ベクトル変換に TF-IDF を用いる CCVolti(TF-IDF) とは，検出するクロ―ンペアが異なり，本手法（IDF 固定）は，CCVolti（TF-IDF）の検出するクローンペアの約 96.8%を検出できる上に，より多くのクローンペアを高い適合率で検出した．また，本手法（BoW 採用）は，同じく特徴ベクトル変換に BoW を用いる CCVolti(BoW) とは，検出するクローンペアが異なり，本手法（BoW 採用）は，CCVolti（BoW）との検出するクローンペアの約 97.5%を検出できる上に，より多くのクローンペアを高い適合率で検出した．

表 6: WildFly の検出において各ツールが出力する検出結果

	1 コミット目		1000 コミット目	
	クローンペア数	適合率	クローンペア数	適合率
CCVolti (TF-IDF)	6787	0.99 (299/300)	8922	0.99 (299/300
CCVolti (TF-IDF EX DIM)	6775	0.99 (299/300)	8934	0.99 (299/300)
本手法 (IDF 固定)	6775	0.99 (299/300)	9384	0.99 (299/300)
CCVolti (BoW)	8825	0.99 (298/300)	12817	0.99 (299/300)
CCVolti (BoW EX DIM)	8772	0.99 (299/300)	12789	1 (300/300)
本手法 (BoW 採用)	8772	0.99 (299/300)	13092	1 (300/300)

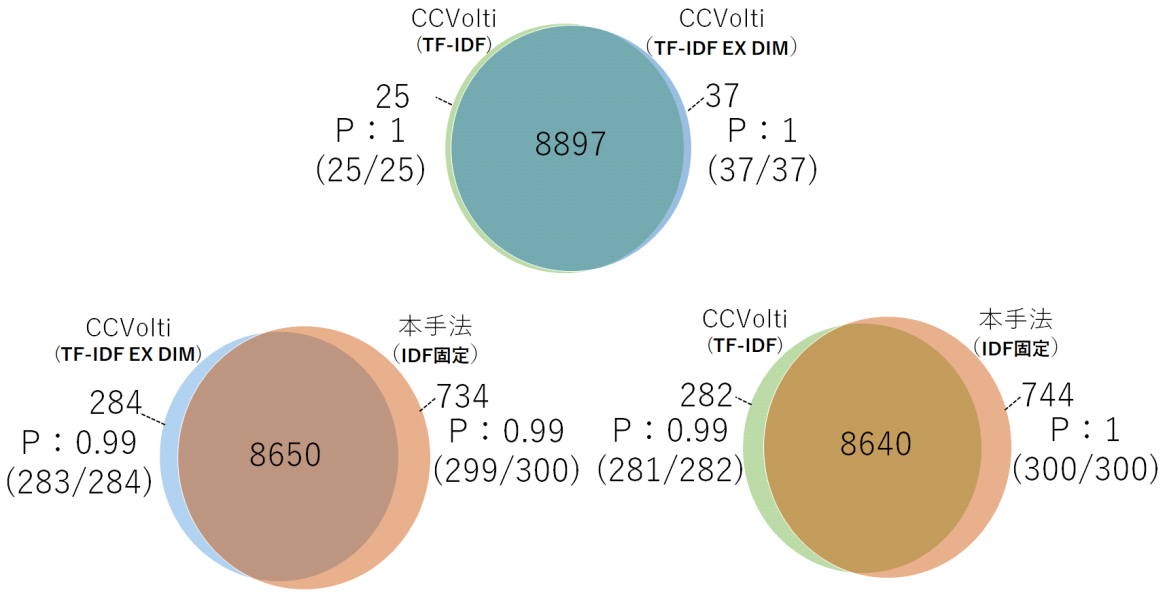


(a) TF-IDF を用いたツール

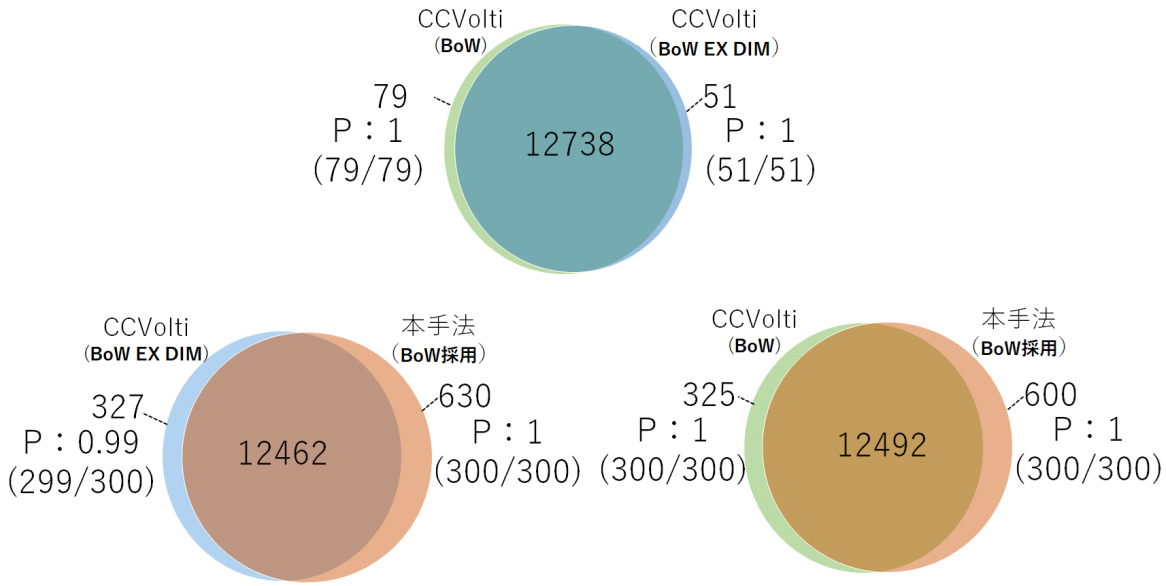


(b) BoW を用いたツール

図 12: 1 番目のコミットの検出で各ツールが出力するクローンペアの関係（WildFly）



(a) TF-IDF を用いたツール



(b) BoW を用いたツール

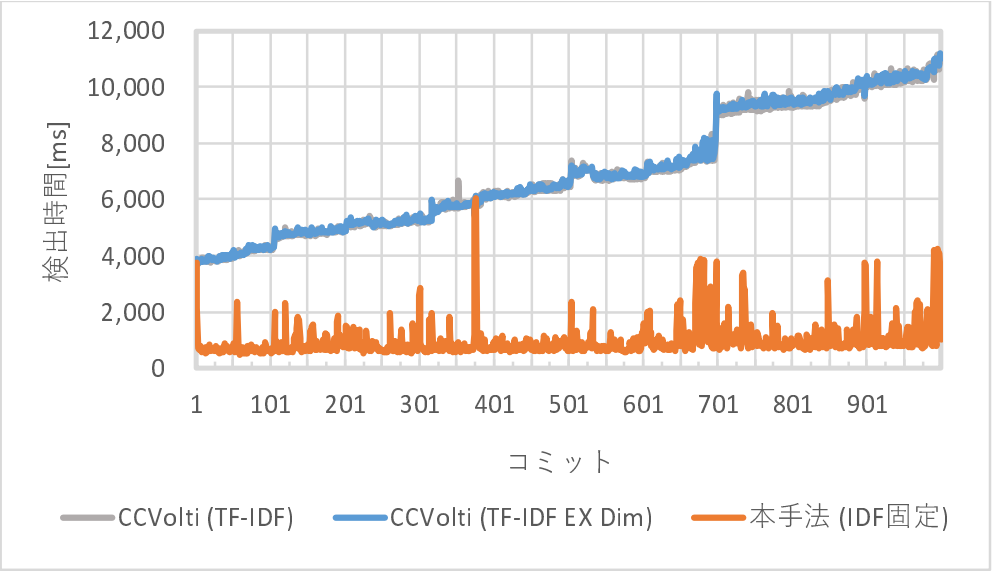
図 13: 1000 番目のコミットの検出で各ツールが出力するクローンペアの関係（WildFly）

4.2 検出時間の比較評価

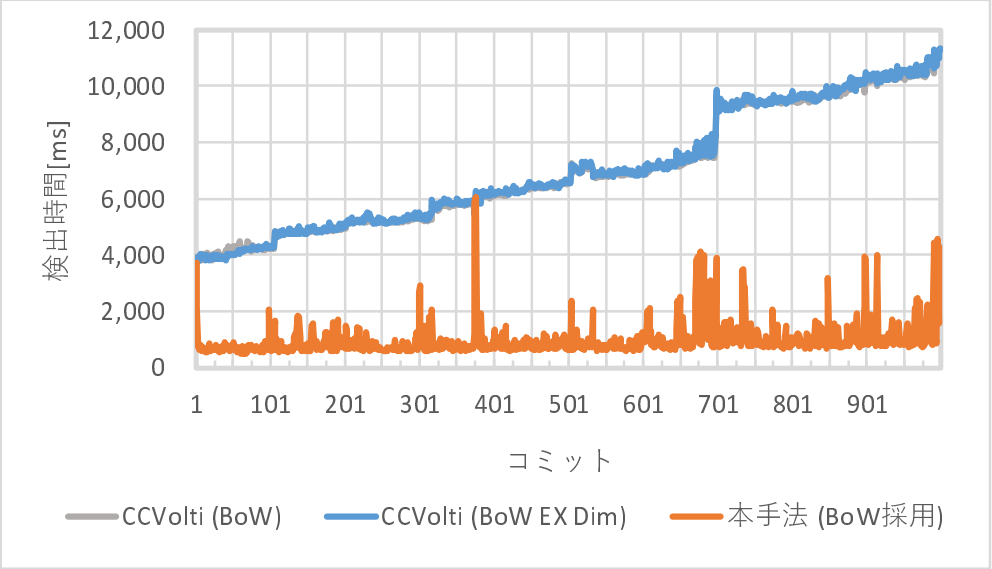
本節では，各コードクローン検出ツールの検出時間について述べる．以下に，本手法と比較対象のツールが評価実験対象プロジェクトからコードクローンを検出するのにかかった検出時間の変化をプロジェクトごとに示し，最後に総検出時間を示す．

Redis

Redis プロジェクトでの TF-IDF を用いたツールと BoW を用いたツールの検出時間の変化を表した図を，図 14 に示す．なお，これらのグラフの要素は，凡例の左の要素から順に重ねられている．(灰色，青色，および橙色の順)



(a) TF-IDF を用いたツール

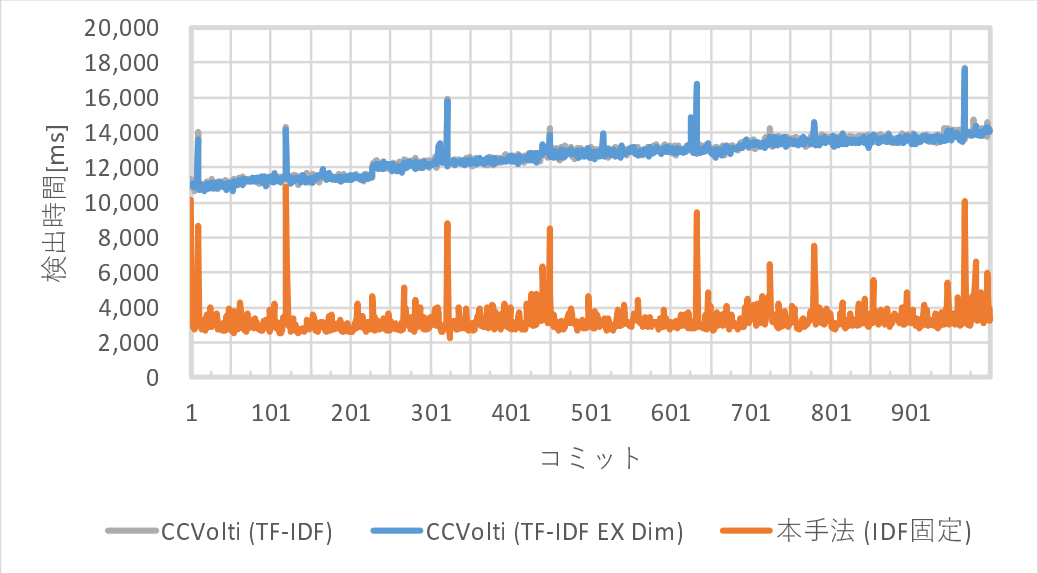


(b) BoW を用いたツール

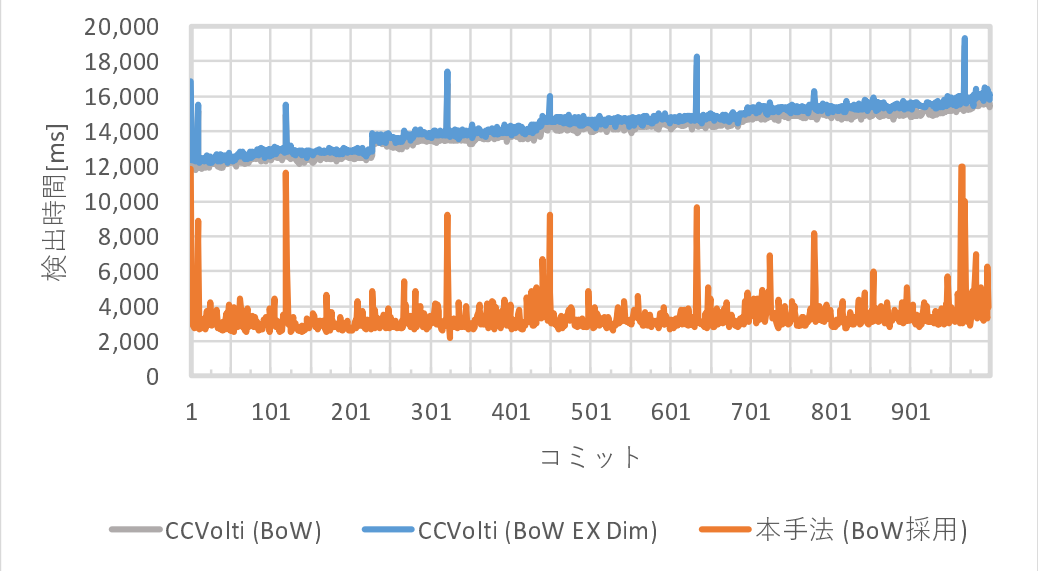
図 14: Redis からコードクローンを検出するのにかかった検出時間の変化

PostgreSQL

PostgreSQL プロジェクトでの TF-IDF を用いたツールと BoW を用いたツールの検出時間の変化を表した図を，図 15 に示す．なお，これらのグラフの要素は，凡例の左の要素から順に重ねられている．(灰色，青色，および橙色の順)



(a) TF-IDF を用いたツール

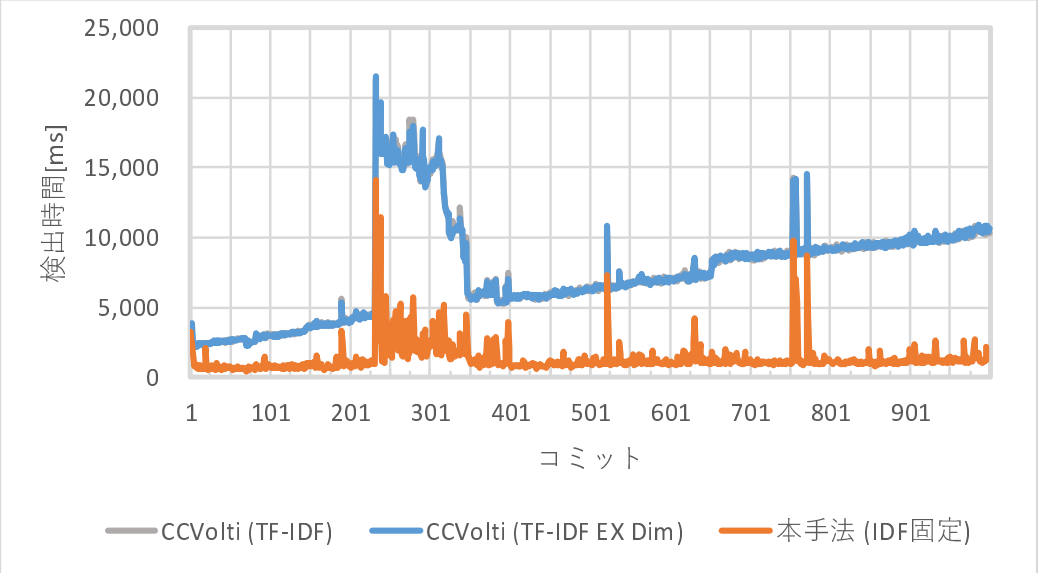


(b) BoW を用いたツール

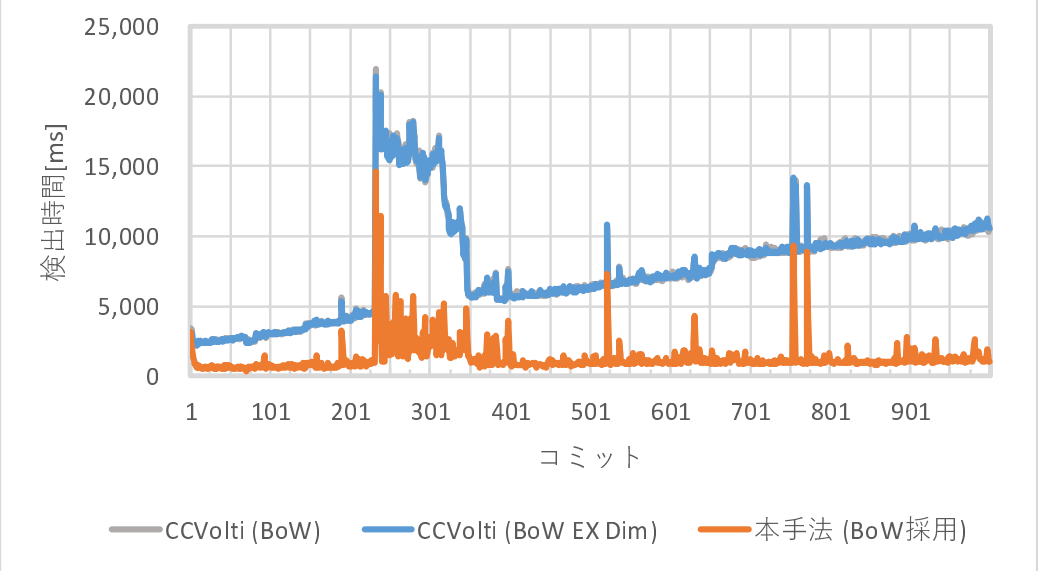
図 15: PostgreSQL からコードクローンを検出するのにかかった検出時間の変化

Apache Ant

Apache Ant プロジェクトでの TF-IDF を用いたツールと BoW を用いたツールの検出時間の変化を表した図を，図 16 に示す．なお，これらのグラフの要素は，凡例の左の要素から順に重ねられている．(灰色，青色，および橙色の順)



(a) TF-IDF を用いたツール

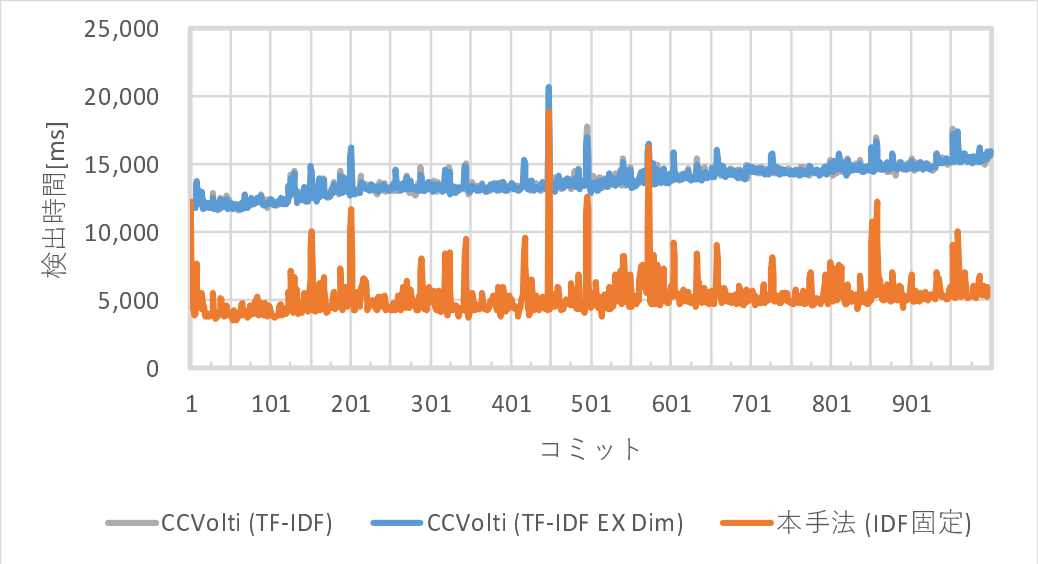


(b) BoW を用いたツール

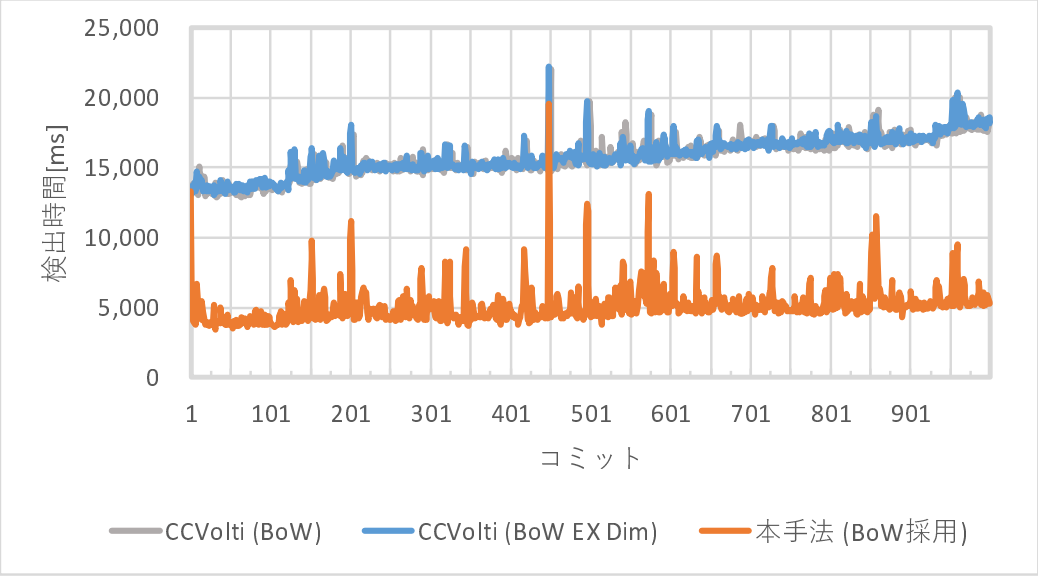
図 16: Apache Ant からコードクローンを検出するのにかかった検出時間の変化

WildFly

WildFly プロジェクトでの TF-IDF を用いたツールと BoW を用いたツールの検出時間の変化を表した図を，図 17 に示す．なお，これらのグラフの要素は，凡例の左の要素から順に重ねられている．(灰色，青色，および橙色の順)



(a) TF-IDF を用いたツール



(b) BoW を用いたツール

図 17: WildFly からコードクローンを検出するのにかかった検出時間の変化

総検出時間

TF-IDF を用いたツールと BoW を用いたツールが評価対象プロジェクトの 1000 個のコミットからコードクローンを検出するのにかかった総検出時間をそれぞれ表 7 と表 8 に示す.

これらの表が示すように，CCVolti（TF-IDF）と本手法（IDF 固定）の総検出時間を比較した結果，Redis では，約 7.1 倍，PosgreSQL では，約 4 倍，Apache Ant では，約 5.8 倍，WildFly では，約 2.7 倍，本手法（IDF 固定）の方が高速にコードクローンを検出することができた.

また，CCVolti（BoW）と本手法（BoW 採用）の総検出時間を比較した結果，Redis では，約 7.1 倍，PosgreSQL では，約 4.1 倍，Apache Ant では，約 6.1 倍，WildFly では，約 3.1 倍，本手法（BoW 採用）の方が高速にコードクローンを検出することができた.

表 7: 総検出時間（TF-IDF を用いたツール）

プロジェクト名	総検出時間		
	CCVolti (TF-IDF)	CCVolti (TF-IDF EX DIM)	本手法 (IDF 固定)
Redis	1h 54m	1h 55m	16m
PostgreSQL	3h 42m	3h 41m	55m
Apache Ant	2h 8m	2h 8m	22m
WildFly	3h 52m	3h 51m	1h 27m

表 8: 総検出時間（BoW を用いたツール）

プロジェクト名	総検出時間		
	CCVolti (BoW)	CCVolti (BoW EX DIM)	本手法 (BoW 採用)
Redis	1h 54m	1h 53m	16m
PostgreSQL	3h 52m	3h 59m	56m
Apache Ant	2h 9m	2h 9m	21m
WildFly	4h 22m	4h 24m	1h 25m

4.3 考察

本節では，検出結果と検出時間の結果の考察を行う．

検出結果の考察

最初に各プロジェクトの 1 番目と 1000 番目のコミットからコードクローンを検出した結果に基づき，ベクトルの次元数を余分に確保することによって検出されるクローンペアの違いの考察を述べる．

各プロジェクトにおいて CCVolti(TF-IDF) と CCVollti(TF-IDF EX DIM) の検出結果を比較し，CCVolti(BoW) と CCVolti(BoW EX DIM) の検出結果を比較することで，ベクトルの次元数を余分に確保することで検出されるクローンペアに違いはあるか確認した．その結果，Redis と Apache Ant では，ベクトルの次元数を余分に確保することによって検出されるクローンペアに違いはなかった．しかし，PostgreSQL と WildFly では，ベクトルの次元数を余分に確保することによって，検出されるクローンペアに違いがあった．この原因として，LSH がランダム性のあるアルゴリズムを利用していることが考えられる．LSH が利用するハッシュ関数は，ランダムに選んだ次元の値を対象のベクトルからサンプリングすることで，ベクトルの次元数が増加しても計算速度が低下しない．また，距離が近いベクトルの衝突確率を上げることができる．このランダム性のある LSH のアルゴリズムにより，CCVolti とベクトルの次元数を余分に確保した EX DIM のツールで，クラスタリングの結果が変わり，検出されるクローンペアに違いが見られたと考えられる．この現象が Redis と Apache Ant には見られず，PostgreSQL と WildFly に見られたのは，PostgreSQL と WildFly の方がソースコードの規模が大きく，検出対象のコードブロック数が多いため，ランダム性の影響が大きく表れたと考えらえる．

次に，本手法（IDF 固定，BoW 採用）が 1000 番目のコミットで他のツールよりも多くのクローンペアを検出した結果の考察を述べる．

1000 番目のコミットのコードクローン検出では，TF-IDF と BoW のどちらを用いた場合でも本手法の方が CCVolti より検出するクローンペア数が多くなった．本手法は，2 つのコミット間で Stable と判定されたコードブロックが旧コミットでも互いにコードクローンとなっているものは，そのまま新コミットでもコードクローンと判定する．このため，検出したコミット数が増えていくと，Stable と判定されたコードクローンが蓄積されていき，CCVolti より本手法の方が検出するクローンペア数が上回ったと考えられる．また，すべてのプロジェクトの 1000 番目のコミットのコードクローン検出の結果から，CCVolti と比べて本手法のみが検出するクローンペアは，TF-IDF と BoW のどちらを用いた場合でも CCVolti とほとんど変わらず高い適合率であった．

次に，本手法の再現率について考察を述べる．

再現率とは，正解集合に対して，実際に検出された割合のことである．CCVolti は既存研究でクローンペアを高い再現率で検出している．本手法では，再現率の測定は行っていないが，代わりに CCVolti が検出するクローンペアの内，本手法はどの程度のクローンペアを検出することができるか確かめた．その結果，評価実験対象の 4 つのプロジェクトから CCVolti(TF-IDF) が検出するクローンペアの内，本手法（IDF 固定）は，約 95.0%を検出し，より多くのクローンペアを検出した．また，評価実験対象の 4 つのプロジェクトから CCVolti(BoW) が検出するクローンペアの内，本手法（BoW 採用）は，約 98.9%を検出し，より多くのクローンペアを検出した．この結果から，本手法は，CCVolti が検出するクローンペアの多くを検出する上に，さらにより多くのクローンペアを検出するため，本手法は CCVolti と同程度以上の再現率を出すことができると考えられる．

最後に，TF-IDF と BoW を用いてコードクローンを検出した結果の違いについて考察を述べる．

すべてのプロジェクトにおいて，TF-IDF と BoW で検出するクローンペアの適合率に大きな違いはなかった．しかし，BoW を用いた場合の方が TF-IDF を用いた場合より多くのクローンペアを検出した．ワードの重要度を考慮する TF-IDF は，予約語より識別子の重要度の方が高くなる傾向がある．タイプ 3 やタイプ 4 のコードクローンは識別子名の変更が行われる場合が多いため，BoW より TF-IDF の方がコサイン類似度が低くなり，検出されるクローンペア数が減ったと考えられる．このため，本手法のコードクローン検出において，コードブロックを特徴ベクトルに変換する時に用いるベクトル表現手法は，TF-IDF より BoW の方が検出するクローンペア数の観点で有効的であると考えられる．

検出時間の考察

各プロジェクトからコードクローンを検出するのにかかった検出時間の変化の考察をする．図 14，図 15，図 16，および図 17 より，すべてのプロジェクトの検出時間の変化の図において，本手法の検出時間は CCCVolti の検出時間をほとんどのコミットで大きく下回ったが，いくつかのコミットで本手法の検出時間と CCVolti の検出時間が近い値となった．これは，そのコミットと 1 つ前のコミットとの間で，ソースコードに対して大きな変更があったためである．例えば，図 15 より，PostgreSQL の検出対象となったコミット内，9，321，633，および 968 番目のコミットの検出で，本手法の検出時間が CCVolti の検出時間と近い値となった．この原因は，Copyright の変更により，プロジェクト内の多くのソースファイルが修正されたためであった．また，

PostgreSQL の検出対象となったコミットの内，120 番目のコミットの検出で，本手法の検出時間が CCVolti の検出時間と近い値となった．この原因は，コーディング規約が変わったことにより，プロジェクト内の多くのソースファイルに対して，インデント整形や空白挿入などのコード整形が行われたためであった．このように本手法では，2 つコミット間のソースコードに対して大きな変更があった場合，検出時間が長くなる．

評価実験の結果と上記の考察より，検出精度に関して，本手法は CCVolti と同程度の適合率と同程度以上の再現率でより多くのコードクローンを検出できることを確認できた．また，検出時間に関して，本手法は CCVolti より，約 2.7 倍〜7.1 倍高速に複数のコミットからコードクローンを検出できることを確認できた．

5　まとめと今後の課題

本研究では，情報検索技術を利用したコードブロック単位のコードクローン検出ツール CCVolti を拡張した，インクリメンタルコードクローン検出手法を提案した．本手法では，1 番目のコミットで検出したコードクローンの情報を保存する．そして，2 番目以降のコミットのコードクローン検出では，2 つのコミット間の差分を抽出し，追加，編集，削除といった変更履歴に基づきコードブロックを分類し，変更されたコードクローン情報を更新する．その後，追加，編集されたコードブロックのみを対象に，特徴ベクトルを計算する．そして，LSH アルゴリズムを用いたクラスリングによって，全コードブロックの特徴ベクトルの集合から，追加，編集，削除されたコードブロックの特徴ベクトルと近似した特徴ベクトルの集合のクラスタを取得する．最後に，クラスタ内の特徴ベクトルの間の類似度を計算することで，クローンペアを検出し，コードクローン情報を更新する．

評価実験では，本手法を含めた 6 つのコードクローン検出ツールを，2 つの C 言語プロジェクトと 2 つの Java プロジェクトに適用し，コードクローンの検出精度と検出時間の観点で比較した．その結果，コードクローンの検出精度においては，本手法は CCVolti の検出するクローンペアの内，約 92%から 100%を検出するできる上に，より多くのクローンペアを高い適合率で検出した．また，検出時間においては，本手法は CCVolti より，約 2.7〜7.1 倍高速に複数コミットからコードクローンを検出した．

今後の課題としては，以下が挙げられる．

- CCEvovis への適用し，保守支援の改善を評価
  本手法は，検出結果の非一貫性の解決とスケーラビリティの向上を実現した．また，本手法は 2 コミット間の差分を対象に効率的にコードクローンを検出するために，コードクローンの変更履歴情報を保持している．このため，本手法を CCEvovis に適用することで，より高速で正確なコードクローン保守支援を開発者に提供できると考えられる．したがって，本手法を CCEvovis に適用し，コードクローンの保守支援の改善を評価する必要がある．

- 多言語対応と他のさまざまなプロジェクトの評価実験
  本手法は，C，C++，C#，Java のプログラミング言語に対応しているが，他の言語にも対応させる必要がある本研究では，2 つの C プロジェクトと 2 つの Java プロジェクトを評価実験対象にコードクローンを検出したが，さらに評価実験対象を増やし，本手法の有用性を確認する必要がある．

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上克郎教授には，研究において大変貴重な御指導および御助言を賜りました．井上教授の御指導および御助言のおかげで本論文を完成させることができました．井上教授に心より深く感謝いたします．

名古屋大学大学院情報学研究科附属組込みシステム研究センター / 情報システム学専攻 吉田則裕准教授には，研究に関する直接の御指導および御助言を賜りました．常に適切な御指導および御助言を頂いたことにより，本論文を完成することができました．吉田准教授に心より深く感謝いたします．

京都工芸繊維大学情報工学・人間科学系 崔恩瀞助教には，研究に関する直接の御指導および御助言を賜りました．常に適切な御指導および御助言を頂いたことにより，本論文を完成することができました．崔助教に心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下誠准教授には，研究の各段階において多くの御助言を賜りました．多くの御指導および御助言を頂いた松下准教授に心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 春名修介特任教授には，研究において御指導及び御助言を賜りました．御指導および御助言を頂いた春名特任教授に心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 神田哲也助教には，研究の各段階において多くの御助言を賜りました．多くの御指導および御助言を頂いた神田助教に心より深く感謝いたします．

株式会社富士通研究所 徳井翔梧氏には，本研究で開発したツールの実装方法において多くの御助言を賜りました．御指導および御助言を頂いた徳井翔梧氏に心より深く感謝いたします．

株式会社エヌ・ティ・ティ・データ 横井一輝氏には，本研究で開発したツールの実装方法において多くの御助言を賜りました．御指導および御助言を頂いた横井一輝氏に心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様には，多くの御指導および御助言を賜りました．井上研究室の皆様のおかげで，有意義な研究室生活を送りながら本論文を完成させることができました．井上研究室の皆様に心より深く感謝いたします．

最後に，学生生活及び研究生活を支えてくださった家族に心より深く感謝いたします．

参考文献

[1] Alexandr Andoni and Indyk Piotr. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Foundations of Computer Science*, Vol. 51, No. 1, pp. 117–122, 2006.

[2] Alexandr Andoni, Indyk Piotr, Laarhoven Thijs, Razenshteyn Ilya, and Schmidt Ludwig. Practical and optimal LSH for angular distance. In *Proc. of NIPS*, pp. 1225–1233, 2015.

[3] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval: The Concepts and Technology behind Search*. Addison-Wesley, 2011.

[4] N. Göde and R. Koschke. Incremental clone detection. In *13th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 219–228, 2009.

[5] Hirotaka Honda, Shogo Tokui, Kazuki Yokoi, Eunjong Choi, Norihiro Yoshida, Katsuro Inoue. Ccevovis: a clone evolution visualization system for software maintenance. In *In the IEEE/ACM 27th International Conference on Program Comprehension (ICPC 2019)*, May 2019.

[6] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pp. 96–105. IEEE Computer Society, 2007.

[7] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[8] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. A token-based code clone detection tool-ccfinder and its empirical evaluation. *Techinal report, Osaka University, Department of Information and Computer Scineces, Inoue Laboratory*, 2000.

[9] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Annual report of Osaka University: academic achievement*, Vol. 2001, pp. 22–25, 2002.

[10] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, Vol. 55, No. 7, pp. 1165–1199, 2013.

[11] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th IEEE International Conference on Program Comprehension*, pp. 172–181, June 2008.

[12] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools. *Science of Computer Programming*, Vol. 74, No. 7, pp. 470–495, 2009.

[13] Zellig s. Harris. Distributional structure. *WORDs*, Vol. 10, No. 2–3, pp. 146–162, 1954.

[14] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: scaling code clone detection to big-code. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pp. 1157–1168. IEEE, 2016.

[15] 横井一輝, 崔恩瀞, 吉田則裕, 井上克郎. コード片のベクトル表現に基づく大規模コードクローン集合の特徴調査. ソフトウェアエンジニアリングシンポジウム 2018 論文集, pp. 192–199, September 2018.

[16] 横井一輝, 崔恩瀞, 吉田則裕, 井上克郎. 情報検索技術に基づく細粒度ブロッククローン検出. コンピュータソフトウェア, Vol. 35, No. 4, pp. 16–36, 2018.

[17] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎. 情報検索技術に基づく高速な関数クローン検出. 情報処理学会論文誌, Vol. 55, No. 10, pp. 2245–2255, 2014.

[18] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎, 佐野建樹. コードクローン変更管理システムの開発と実プロジェクトへの適用. 情報処理学会論文誌, Vol. 54, No. 2, pp. 883–893, 2013.

[19] 川口真司, 松下誠, 井上克郎. 版管理システムを用いたクローン履歴分析手法の提案. 電子情報通信学会論文誌, Vol. J89-D, No. 10, pp. 2279–2287, 2006.

[20] 肥後芳樹, 楠本真二, 井上克郎. コードクローン検出とその関連技術. 電子情報通信学会論文誌 D, Vol. 91, No. 6, pp. 1465–1481, 2008.