特別研究報告

題目

ソースコード中の識別子情報にもとづくコミット分類の提案と評価

指導教員
楠本 真二 教授

報告者
山内 健二

平成 26 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 25 年度 特別研究報告

ソースコード中の識別子情報にもとづくコミット分類の提案と評価

山内 健二

内容梗概

ソフトウェアの開発において，これまでに記述されたソースコードの内容や，それにより実現されたソフトウェアの挙動について整理や検討が必要な場面がある．このような整理や検討は，バージョン管理システムやタスク管理システムを利用し，各コミットにおけるソースコードへの変更内容を，タスクという開発作業の単位で把握することで実現できる．

このとき，コミットをタスク単位で分類するためには，タスクとコミットの対応関係の特定が必要となる．この特定を行うための手段として，コミットコメントやコミットを行った開発者の名前を利用することが考えられるが，コミットコメントの内容不備などから，対応関係の特定が困難な場合がある．

そこで，本論文では，各コミットにおけるソースコードの差分に出現する識別子情報を利用してタスク単位でのコミット分類を行い，タスク単位での変更内容の把握を支援する手法を提案する．識別子情報を用いることで，抽象構文木などの構文情報からは得ることのできない，そのソースコードが書かれた目的などの情報を得ることができる．それぞれのソースコードが書かれた目的は実装内容と強く関連するため，識別子情報を用いてコミットの分類を行うことで，タスク単位でのコミット分類を実現することができると期待できる．

また，提案手法の評価として，いくつかのオープンソース・ソフトウェアに対して，提案手法によるコミットの分類を行い，その結果，コミットコメントや開発者名を用いた場合は適切にコミットの分類ができない事例に対し，適切にコミットの分類を行えたことを確認した．

主な用語

バージョン管理システム
コミット分類
タスク管理システム

目 次

1　まえがき

ソフトウェアの開発において，これまでに記述されたソースコードの内容や，それにより実現されたソフトウェアの挙動 (以下 **実装内容** と呼ぶ) について整理や検討が必要な場面がある．このような整理や検討を効率的に行うには，機能の追加や変更など，開発の過程で設定される目標ごとに実装内容を把握することが有用である．本研究においては，このような開発工程での目標について，それを達成する作業群を，それぞれタスクと呼ぶ．タスク単位での実装内容の把握が有用な場面として，コードレビュー，リリースノートの作成，タスク管理システムとの同期などが挙げられる．

このようなタスク単位での実装内容の把握は，バージョン管理システム (Version Control System，VCS) を利用することで，より効率的に行うことができる．バージョン管理システムからは，ソースファイルに対して行われた変更の内容を，コミットという単位で取得できる．それぞれのコミットがどのタスクに関係しているのかを特定し，コミットをタスクごとに分類することで，タスク単位での実装内容の把握が容易となる．これまでに，コミットコメントやコミットを行った開発者の名前を用いて，コミットとタスクの対応関係を特定する手法が提案されている [1] ．しかし，これらの手法には，適切なコミットコメントが記述されていない場合や，開発者とタスクが多対多で対応している場合に，コミットとタスクを適切に関連付けることができないという課題がある．

コミットコメントや開発者の名前を用いる手法以外のコミットの分類手法には，ソースコードの差分情報を利用する方法がある．ソースコードの差分を用いることで，コミットコメントの記述が不十分な場合や，開発者とタスクが多対多で対応している場合でも，コミットの分類を行うことができる．これまでにソースコードの差分を用いたコミットの分類手法が提案されているが [2] ，それらはタスク単位でのコミット分類を対象としていない．また，ソースコード中の構文構造のみを利用しているため，分類にそのソースコードが書かれた目的などを反映できないという課題がある．

そこで，本研究では，ソースコードの差分に含まれる識別子情報に基づいてコミットの分類を行う手法を提案する．識別子情報を用いることで，抽象構文木などの構文情報からは得ることのできない，そのソースコードが書かれた目的などの情報を得ることができる．それぞれのソースコードが書かれた目的は実装内容と強く関連するため，識別子情報を用いてコミットの分類を行うことで，タスク単位でのコミット分類を実現することができると期待できる．オープンソースソフトウェア (OSS) に対して提案手法を適用し，提案手法がタスク単位でのコミット分類を行うことができているかを評価した．その結果，コミットコメントや開発者名を用いた場合は適切にコミットの分類ができない事例に対し，適切にコミットの分類を行えたことを確認した．

以降，まず2節では，本研究で用いるバージョン管理システム，クラスタリング，タスクに関する説明を行う．次に，3節で提案手法の説明をした後，続く4節，5節でそれぞれ提案手法に対する評価実験の結果とそれに対する考察を述べる．最後に，7節で本研究のまとめと今後の課題について述べる．

2　背景

2.1　バージョン管理システム (Version Control System，VCS)

バージョン管理システム (Version Control System，VCS) は，開発で利用されるソースファイルやその他のリソースについて，開発者間で共有や変更履歴の管理を目的として利用されるシステムである．VCS では，リポジトリと呼ばれる，管理対象となるデータや，それらの変更履歴を保存するデータベースを持ち，各種操作はこのリポジトリに対して行われる．リポジトリに含まれる情報を抽出並びに解析することで，VCS で管理されるソフトウェアに混入したバグや，開発者がそのソフトウェアの開発に携わる際に必要な専門知識など，開発者にとって有益な情報の提供が可能である [3, 4].

以下，VCS で用いられるコミットという用語について，本研究で用いる Git[1] という VCS における表現に準拠して説明を行い，さらにコミットを分類対象とする研究について述べる．

2.1.1　コミット

Git では，リポジトリで管理されているデータに対して変更を反映させることを，コミットという．コミットはファイルに対する変更の内容，変更日時，コミットを実行した人物の情報，変更に対する注釈 (コミットコメントという) といった情報で構成されている．コミットには，一意なハッシュ値が割り当てられ，これを **コミット ID** と呼ぶ．

2.1.2　コミット分類

VCS のリポジトリに含まれているコミットを，その特徴によって分類することが，開発支援や調査の目的から行われている．例えば，Hindle ら，及び Hattori と Lanza はそれぞれコミットコメントに対して自然言語処理を適用して分類を行うことで，各コミットがソフトウェアの挙動に与える影響を調査している [5, 6].

2.2　クラスタリング

クラスタリングとは，データ群をいくつかのグループに分類する手法のことをいい，統計的機械学習の技術の 1 つである．統計的機械学習とは，現存するデータを統計的に解析することで，未知の問題に対して解を求める技術である．

クラスタリングでは，まず，現存するデータをスカラ値やベクトルなどの数値的な表現 (**特徴量** という) に変換する．次に，各特徴量間の関係を数値化し，その値を元に統計的手法を用いて，分類を実現する．分類が行われたあとの各グループをクラスタと呼ぶ．

[1] http://git-scm.com/

クラスタリングを行うアルゴリズムは，クラスタ数をあらかじめ指定する必要があるものとそうでないものがある．事前にクラスタ数を指定する必要があるアルゴリズムには k-means 法や階層的，指定の必要がないものには Affinitiy Propagation [7] ，Repeated Bisection(CLUTO [8] というクラスタリングツールで利用されている) などがある．

以降，本研究の提案手法で用いるクラスタリングのアルゴリズムである Repeated Bisection と，クラスタリングの精度評価に用いる指標について述べる．

2.2.1 Repeated Bisection

Repeated Bisection は，データ集合とあるしきい値 eps を入力とし，クラスタリング結果を出力する．このアルゴリズムでは，データ集合に対する再帰的な 2 分割を繰り返すことによってクラスタリングを実行する．繰り返しの際，各クラスタには重心と呼ばれるベクトルが計算され，各クラスタの特徴を表す指標となる．

以降，Repeated Bisection の実行手順を述べる．ただし，あるクラスタ C を考えた時， C のまとまり具合を数値化する評価関数 $F(C)$ を定義する [9] .

STEP1 データ集合の全要素を 1 つのクラスタに分類しておく．

STEP2 各クラスタについて，評価関数の値と eps の差を計算し， eps より評価関数の値が大きく，差が最も大きいクラスタ C_t を選択する．

STEP3 C_t の要素群をランダムに二分割した 2 つのグループ C'_{t1}、C'_{t2} を作成し，これらを新たなクラスタとする．

STEP4 C'_{t1}、C'_{t2} それぞれに対する評価関数の値 $F(C'_{t1})$、$F(C'_{t2})$ と， eps との差が小さくなるように両クラスタ間で要素を移動させる．

STEP5 STEP2 から STEP4 を繰り返し，全てのクラスタに対する評価関数の値が eps を下回ったら終了する．

以下，仮想的なデータ集合を考えて，具体的に実行手順を述べる．今，5 つの要素からなるデータ集合 $A = (a_1, a_2, \cdots, a_5)$ があったとし， A としきい値 eps $= 0.2$ を Repeated Bisection のアルゴリズムに与えたとする．

まず，STEP1 で，クラスタ C に対して A の要素を全て所属させる．次に，STEP2 で，$F(C) = 0.5$ の場合，$F(C) > 0.2$ であるから，C_t として C を選択する．さらに，STEP3 で C'_{t1} ， C'_{t2} としてそれぞれ A の要素 a_1、a_3、a_5 および a_2、a_4 がそれぞれランダムに配属されたとする．すなわち， $C'_{t1} = \{a_1, a_2, a_5\}, C'_{t2} = \{a_2, a_4\}$ である．そして，STEP4 では $F(C'_{t1}) = 0.3$、$F(C'_{t2}) = 0.15$ となったとする．

ここで，両方共の評価関数の値が eps より小さいわけではないので，再度 C'_{t1}、C'_{t2} 両クラスタ間で要素を移動させ，両クラスタに対する評価関数と eps との差が縮まるようにする．そして，$C'_{t1} = \{a_1, a_2, a_4\}, C'_{t2} = \{a_3, a_5\}$ となった時に $F(C'_{t1}) = 0.23$、$F(C'_{t2}) = 0.16$ となり，両クラスタに対する評価関数と eps との差が一番小さくなったとする．

このとき，STEP5 でまだ全てのクラスタ C'_{t1}、C'_{t2} に対する評価関数の値が eps より小さくなっていないため，STEP2 に戻る．2 回目の STEP2 において，評価関数の値が eps より大きいのは C'_{t1} であるから，これを選択し，STEP3 で C'_{t11}、C'_{t12} へ分割する．以降，1 度目の処理と同様に STEP3 と STEP4 を行い，最終的に $C'_{t2} = \{a_3, a_5\}$、$C'_{t11} = \{a_2\}$、$C'_{t12} = \{a_1, a_4\}$ となった時，それぞれに対する評価関数の値が $F(C't2) = 0.16$、$F(C'_{t11}) = 0.10$、$F(C'_{t12}) = 0.18$ となったとすれば，全ての値が eps より小さいため，STEP5 でアルゴリズムは終了する．よって，クラスタリング結果は，要素の所属がそれぞれ $C_1 = \{a_3, a_5\}$、$C_2 = \{a_2\}$、$C_3 = \{a_1, a_4\}$ である 3 つのクラスタ C_1、C_2、C_3 となる．

2.2.2 評価指標

本節では，2 種類のクラスタリング結果を比較する指標である Adjusted Rand Index(ARI) [10] ，Homogeneity，Completeness [11] の 3 つの指標について，説明を述べる．

ARI ARI は，同一の要素群に対する 2 種類のクラスタリング結果が与えられた時，両者がどの程度一致しているかを表す指標である．0 から 1 の実数値をとり，0 の場合 2 つの結果は全く一致しておらず，1 であれば完全に一致していることをそれぞれ示す．すなわち，値が 1 に近ければ近いほどクラスタリング結果が一致している度合いが高い．

以下，ARI の定義を示す．まず，分類対象の要素が n 個あるとし，各要素を $o_k(1 \leq k \leq n)$ ，その集合を $S = \{o_1, o_2, ...o_n\}$ とする．また，2 種類のクラスタリング結果を，それぞれ $U = \{u_1, u_2, ...u_R\}$ ， $V = \{v_1, v_2, ...v_C\}$(R, C は自然数) とする．ただし u_k及び $v_l(1 \leq k \leq R, 1 \leq l \leq C)$ はそれぞれ集合であり， それぞれ重複する要素を持たず，各集合の合計が S となる．すなわち， $\cup_{i=1}^{R} u_i = S = \cup_{j=1}^{C} v_j$ かつ $u_i \cap u_{i'} = \emptyset = v_j \cap v_{j'}(1 \leq i \neq i' \leq R, 1 \leq j \neq j' \leq C)$ である．

ここで，2 つの要素の組 o_k及び $o_{k'}(1 \leq k \neq k' \leq n)$ を全通り考え，以下の a, b, c, d の値を計算する．

a 2 つの要素が U でも V でも同じクラスタに属している

b 2 つの要素が U では同じクラスタに， V では別のクラスタに属している

c 2 つの要素が U では別のクラスタに， V では同じクラスタに属している

d 2つの要素が U でも V でも別のクラスタに属している

ここから，ARI は式 1 で得られる．$\binom{n}{2}$ は $\frac{n(n-1)}{2}$ である．

$$ARI = \frac{\binom{n}{2}(a+d) - [(a+b)(a+c)+(c+d)(b+d)]}{\binom{n}{2}^2 - [(a+b)(a+c)+(c+d)(b+d)]} \tag{1}$$

本実験においては，11 個のクラスタについて，提案手法による分類の結果 (上記の U に当たる) で含まれているコミット全体を S，被験者による手動分類の結果を V として考えることで，式 1 から ARI を各被験者ごとにそれぞれ求める．

Homogeneity と Completeness　Homogeneity は，比較対象となるクラスタリング結果におけるあるクラスタが，もう片方の正解となるクラスタリング結果におけるあるクラスタに属する要素のみを含んでいるかどうかの指標である．他方，Completeness はあるクラスタリング結果におけるあるクラスタの要素全てが，比較対象となるクラスタリング結果において同じクラスタに属しているかどうかを示している．ARI が 2 つのクラスタリング結果が厳密にどの程度一致しているかを検証しやすい分，Homogeneity や Completeness は，間違ったクラスタリング結果にも高い値が出てしまう可能性があるものの，1 つのクラスタに属すべき要素群がどの程度同じクラスタに分類されやすいかを検証しやすいという特徴を持つ．

両者とも ARI と同様に 0 から 1 の実数をとり，1 に近づくほど比較対象となるクラスタリング結果がよいことを示す．また，互いに相反する指標であり，一般的に Homogeneity が高くなると Completeness が低くなり，逆に Completeness が高くなると Homogeneity が低くなる．

以下，両指標に対する定義を述べる．まず，比較対象となるクラスタリング結果が与えられた時の，正解となるクラスタリング結果に対する条件付きエントロピー $H(C|K)$ と，比較対象となるクラスタリング結果に対してのエントロピー $H(C)$ を，それぞれ式 2 と式 3 のように求める．

ただし，n，n_c，n_k はそれぞれクラスタリング対象となった要素の数，正解となるクラスタリング結果のあるクラスタ c に含まれる要素の数，比較対象となるクラスタリング結果のあるクラスタ k に含まれる要素の数を表す．また，$n_{c,k}$ は正解となるクラスタリング結果のあるクラスタ c にも，比較対象となるクラスタリング結果のあるクラスタ k にも含まれる要素の数である．

$$H(C|K) = -\sum_{c=1}^{|C|}\sum_{k=1}^{|K|}\frac{n_{c,k}}{n}\log\left(\frac{n_{c,k}}{n_k}\right) \tag{2}$$

$$H(C) = -\sum_{c=1}^{|C|} \frac{n_c}{n} \log\left(\frac{n_c}{n}\right) \tag{3}$$

このとき，Homogeneity h は式 4 のように定義される．

$$h = 1 - \frac{H(C|K)}{H(C)} \tag{4}$$

式 2 と式 3 と同様に，正解となるクラスタリング結果が与えられた時の比較対象となるクラスタリング結果に対する条件付きエントロピー $H(K|C)$ と，比較対象となるクラスタリング結果に対するエントロピー $H(C)$ を，求めることで Completeness c が式 5 のように与えられる．

$$c = 1 - \frac{H(K|C)}{H(K)} \tag{5}$$

また，Homogeneity と Completeness はこれらの調和平均を取ることで，V-Measure と呼ばれる指標を得られる．この V-Measure 単体で，片方のクラスタの全要素を含むようなもう片方のクラスタがどの程度存在するかの指標となる．

2.3 文書クラスタリング

文書クラスタリングは，自然言語処理において，自然言語において記述された文書を，クラスタリングの手法を適用してスポーツ，経済など，あらかじめ規定された種別に分類することである [12]．本研究では，コミットを文書とみなして，クラスタリングを行う．

以下，本研究で利用する，文書クラスタリングに関する手法についての説明を行う．

2.3.1 Bag-Of-Words

Bag-Of-Words とは，文書を単語の出現回数を各要素としたベクトルとして数値化することである [13] ．このベクトルはクラスタリングの際，特徴量となる．

このとき，分類精度向上のため，ストップワードと呼ばれる語はベクトルの要素の対象から外される場合がある．ストップワードは，英語であれば助動詞，前置詞など，どの文書にもよく出現するが，その文書を特徴づける要因とはならないために，分類の基準として不要とみなせる語のことである．

2.3.2 tf-idf 法

tf-idf 法とは，文書クラスタリングで分類精度向上のために用いられる手法の 1 つである [14] ．各文章において登場する単語が，どの程度その文章を特徴付けるかを考慮して，2.3.1 節で挙げた，文書を表現するベクトルを正規化する．

各文章において，出現頻度が多い単語に対して重み付けを大きくする tf 法と，重み付けにその単語が出現する文章の数の逆数をとる，すなわち出現する文章が少ないほどその単語の重み付けが大きくなる idf 法を組み合わせる．tf 法は，その文章において出現頻度が多い単語が重要であるという考えに，idf 法は，出現する文書数が少ない単語ほど，その単語の出現が文章を特定させる要因となるために，文章の特徴を示しうるという考えにそれぞれ依拠する．

具体的な正規化の計算方法を述べる．まず， 2.3.1 節で挙げた，各文書から生成されたベクトルそれぞれについて，tf 法，idf 法でそれぞれで正規化した結果 tf_v、idf_v を求める．このとき， i 番目の要素の新しい値は，tf 法では式 6 の tf_i ，idf 法では式 7 の idf_i となる．また，式 6，7 において， N は文章全体での単語の数， n_i は BOW ベクトルの i 番目の要素の元の値 $(i \geq 1)$， D は総文書数，d_i は BOW ベクトルの i 番目の要素の値が対応する単語が出現した文書数を示す．

$$\mathrm{tf}_i = \frac{n_i}{\sum_j^N n_j} \tag{6}$$

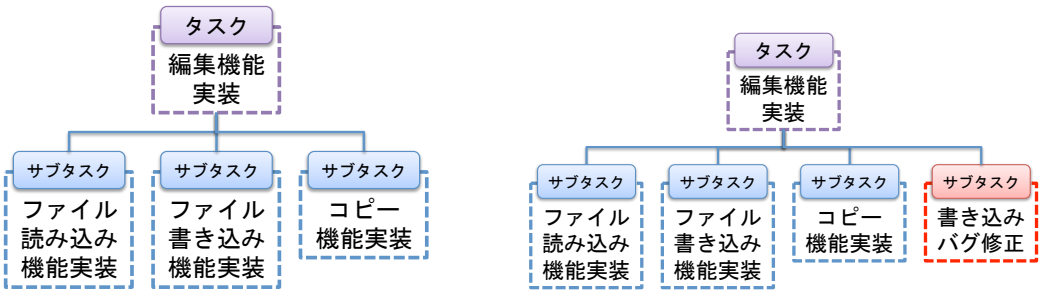
$$\mathrm{idf}_i = \log \frac{|D|}{|d_i|} \tag{7}$$

次に，正規化後のベクトルの内積，すなわち $\mathrm{tf}_v \cdot \mathrm{idf}_v$ を最終的な正規化の結果とする．

2.4 タスク

タスクとは，本来 Work Breakdown Structure(WBS) によるソフトウェア開発工程で用いられる用語で，開発におけるモジュール，パッケージ，機能など，特定の成果物を完成させるための作業内容である [15]．WBS での定義では，1 つの成果物に対してタスクは 1 つ以上対応する．また，タスク自体も，開発者の判断でさらに複数のサブタスクへと細分化される場合がある．このような成果物，タスク，サブタスクの関係は，木構造で表現することができる．元のタスクの分割回数に応じて木構造は高さが深くなっていく．

本研究ではタスクという用語を，WBS で定義されたものから拡張して用いる．まず，機能追加や挙動の変更などの，開発の工程で最終的に達成すべき目標もタスクとして考え，これが前述した木構造の根となるものとする．また，機能の実装途中で発生したバグの修正など，実装前には予定されていなかった作業内容についても，その作業内容が関連するタスク



(a) タスクとサブタスク　　(b) サブタスクの追加

図 1: タスクの例

に対するサブタスクとする．さらに，タスクあるいはサブタスクについて，前述した木構造における高さを **粒度** と呼ぶことにする．より粒度が粗いタスクほど根に近く，細かいほど葉に近い．

タスクの例として，図 1 に示すように，エディタの開発を想定し，ファイルの編集機能を作成するタスクを考える．初期状態で編集機能作成タスクに対しては図 1a のように，3 つのサブタスクが存在していたと仮定する．この状況において，3 つのサブタスクをそれぞれ完了させることで，編集機能作成タスクは完了する．しかし，サブタスクを進めていく過程で新たなタスクが発生する場合がある．図 1a で考えた時，書き込み機能実装のサブタスクを実装中にバグが発生したと仮定する．このとき，バグ修正は，図 1b のように関連するタスクのサブタスクになるとする．

このように，ソフトウェア開発における作業内容は，階層的なタスクの構造を考えることで，整理できる．タスクは開発者の判断で粒度を細かくできない段階まで細分化するため，開発での作業内容をこれらタスクの集合とみなすことで，開発者が把握しやすい単位で開発工程の管理をできるようになる．

2.4.1 タスク管理システム

タスク単位による開発工程の管理を支援するために，Redmine[2] や JIRA[3] などの，プロジェクト管理システムや課題追跡システムと呼ばれるシステム (以下 **タスク管理システム** と呼ぶ) が存在する．また，タスク管理システムでは，各タスクへそれぞれに一意な ID(以下 **タスク ID** と呼ぶ) を割り当て，各タスクにおける作業の内容や進行状況などを個別に管理できるようにしている．

[2] http://www.redmine.org/
[3] https://www.atlassian.com/software/jira

2.4.2 タスク単位での実装内容の把握

ソフトウェアの開発工程において，実装内容について整理や検証が必要な場面がある．このような整理や検証は，実装内容をタスク単位で把握することで効率的に行うことができる．以下，タスク単位での実装内容の把握が有効である具体的な場面を挙げる．

リリースノートの作成　新しいバージョンのソフトウェアをリリースする際，リリースノートを作成する場合がある．リリースノートには通常前回のリリース時から実現されたソフトウェアの挙動やバグ修正を列挙する必要がある．ここで，これら列挙すべき項目はそれぞれ1つのタスクに相当するため，記述されたコードで実現されたソフトウェアの挙動について，タスク単位で把握することが必要である．

タスク管理システムとの同期　2.4.1 節 で述べた，タスク管理システムを利用している場合，開発工程で発生する各タスクについては，全てタスク管理システムへ登録し，開発の進行との同期を行うことが理想であるが，実際の開発工程では完全に同期されないことも多い [1]．そのため，タスク登録の漏れを補正する必要があるが，この場合，前回同期を行った時点からの実装内容を，タスクごとに理解，整理する必要がある．

コードレビュー　ソフトウェア開発では，あらかじめ期日を設け，それまでに達成しなければならない実装内容をタスク単位で決定した上で開発を行うことが多い．例えば，OSS である Eclipse[4] や FireFox[5] などは次のメジャーリリースまでに実装すべきタスクと，それらの達成度を提示している．このような場合，リリース前の特定の段階でレビューを行うことで，設定したタスクが達成されているかの検証を行うことができる．検証すべき内容の例としては，実装されていないタスクが存在していないか，あるいは実装されたタスクについても仕様を満たすようになっているかといったものが挙げられる．このような検証を行うためには，コードレビューの際に，ある期間での実装内容がどのタスクと対応しているかを把握する必要がある．

2.5 タスク単位でのコミット分類

2.4.2 節で触れたタスク単位での実装内容の把握を支援する手段の 1 つとして，タスク単位でのコミット分類が挙げられる．VCS を利用することで，リポジトリから，実装内容をコミットごとにソースコードの差分として取得することができる．これを関連するタスクご

[4] http://www.eclipse.org/
[5] https://www.mozilla.org/firefox/

とに分類することで，それぞれのタスクを実現するためにソースコードに対してどのような修正が加えられたのかを把握することができる．

タスク単位でコミットを分類する場合，分類の基準の候補としてコミットコメント，コミットを行った人物の名前，ソースコードの差分情報という 3 つの情報が挙げられる．しかし，差分情報以外の情報である前者 2 つの情報は，タスク単位でのコミット分類に利用するには不十分である．以下，差分情報以外の情報にもとづく分類とその問題点を挙げた後，差分情報による分類手法，特に本研究にて利用する差分中に出現する識別子を利用した分類について述べる．

2.5.1 差分情報以外の情報にもとづく分類とその問題点

2.4.1 節 で述べたタスク管理システムを利用していれば，各タスクには一意なタスク ID が割り当てられる．よって，このコミットコメントで記述していれば，タスクとコミットとの関連性が判断できる [1]．そのため，全てのコミットについて，コミットコメントに関連するタスクのタスク ID が記述されていれば，そこからコミットをタスク単位で分類できる．しかしながら，単純にコミットコメントを参照するだけでは，タスク単位での実装内容の把握が困難な場合がある．なぜなら，タスクに割り当てられているタスク ID がコミットコメントに含まれていない，あるいはそもそもコミットコメント自体が記述されていないなどの不備が存在しうるからである．このような場合，それぞれのコミットを個別に確認していくだけでは，各コミットがどのタスクと関連しているかを判断することが困難である．特に，1 つのタスクに対する実装が複数のコミットで行われている場合，これらのコミットについてコミットコメントの不備があり，しかも時系列的に連続していない場合，各コミットとタスクとの関連性を判断することがより困難となる．さらに，あるコミットと対応するタスクがタスク管理システムに登録されていないという状況も考えられ，この場合はコミットとタスクの関連性の判断がそもそも不可能である．

次に，コミットを行った人物の名前を分類基準として利用することを考えられる．1 つのタスクに対して，1 人の開発者のみが割り当てられているならば，コミットを行った人物の名前から関連するタスクを判定可能である．しかしこれも，開発者とタスクが多対多で割り当てられている可能性があるため，コミットと関連のあるタスクを一意に判断できるとは限らないという問題がある．

2.5.2 差分情報にもとづく分類

2.5.1 節で述べたように，コミットコメントやコミットを行った人物の名前には，コミットとタスクを 1 対で関連付けるための情報が欠落している場合がある．特に，不特定多数の

開発者が断続的に開発に関わるOSSでは，このような欠落は頻繁に起きうると考えられる．そのため，このような欠落がある場合にも対処するため，ソースコードの差分情報を利用したコミット分類を考える．ソースコードの差分情報を用いたコミットの分類では，各コミットでソースコードに対してどのような修正が加えられたのか，という情報にもとづきコミットの分類を行う．分類基準としてソースコードの差分情報を利用することには，各コミットに必ず含まれ，しかもタスク管理システムや仕様書などリポジトリ以外のデータベースに含まれる情報へ依存する必要が無いという利点がある．

例えばDraganらは，ステレオタイプと呼ばれる典型的な設計種別を利用してコミットの分類を行っている [2]．しかしこの研究の手法では，タスク単位での分類を目的としておらず，文法構造上でどのような変化がなされたかという情報でしか分類できない．

そこで本研究では，ソースコードの差分情報中に出現する識別子を利用したコミット分類を考える．識別子は抽象構文木などの文法構造だけからは得られない，実装内容についての情報を得ることができる [16]．仕様書など，ソースファイル以外の情報を除くと，ソフトウェアの保守において最も用いられている情報は，ソースファイル中の識別子であることも報告されている [17]．

ここで，抽象構文木のような文法構造では得られない実装内容についての情報が，識別子から得られることを，具体的に例を挙げて述べる．Javaで記述されたメソッドについて `isLeapYear(int year)` というシグネチャがあるとする．このシグネチャからは，抽象構文木のような文法構造の情報からは，int型の引数をひとつ取るなんらかのメソッドであるということしかわからない．しかし， `isLeapYear` と `year` というメソッド名および引数名の識別子を見ることで，このメソッドが引数として西暦年 (`year`) を取り，うるう年であるか (`isLeapYear`) を判別する処理を行うことが理解できる．すなわち， `year` からはそのint型の変数が西暦年を表現することを意図しており，`isLeapYear` からは，そのメソッドがうるう年の判定を意図して記述されたことを把握できる．このように，識別子にはそのソースコードが書かれた目的が反映される．

また，タスクは特定の成果物に対する作業内容という点で，このような目的と関連しやすいと考えられる．例えば先程の `isLeapYear(int year)` のメソッドであれば，そのメソッド自体が成果物である「与えられた西暦年がうるう年であるかどうかを判定する機能を実装する」というタスクを考えることができる．

このように，ソースコード中の識別子はソースコードが書かれた目的を通して，タスクとの関連が強いといえる．

ここで，識別子情報を用いたタスク単位でのコミット分類には，トピックモデリングを利用することも考えられる．トピックモデリングとは，自然言語処理で利用されている手法で，単語の出現傾向から，ある文書の特徴づける分野や話題 (トピックという) を特定する手法の

ことである [18] ．この手法をソフトウェア工学に応用し，ソースコード中に出現する識別子を単語とみなして，トピックモデリングによりリポジトリを解析することが行われている．

例えば，Thomas らはリポジトリ中の各リビジョンにおけるソースファイル群に対してトピックモデリングを行い，ソフトウェア開発の流れを可視化している [19] ．このため，トピックをタスクとみなし，各コミットでの変更内容に対してトピックを付与することで，タスク単位でのコミット分類を実現する手法も考えられる．しかしながら，ソフトウェア工学におけるトピックは，ソフトウェアの開発工程やソースコードの全体像を把握するために用いられるため，設定されるトピックの数は少ない [16]．そのため，開発における具体的な実装内容の単位であるタスクと比べて粒度が粗すぎるために，タスク単位での分類へ応用するのは困難である．

そこで，本研究ではこれらの制限を改善するため，差分中に含まれる識別子の出現回数を用いたクラスタリングによるコミット分類を考える．トピックマイニングでは，生成するトピックの数を限定していたが，これを限定せずに行うことで，より粒度の細かい分類，すなわちタスク単位での分類を行うことを目指す．手法の詳細は 3 節 にて述べる．

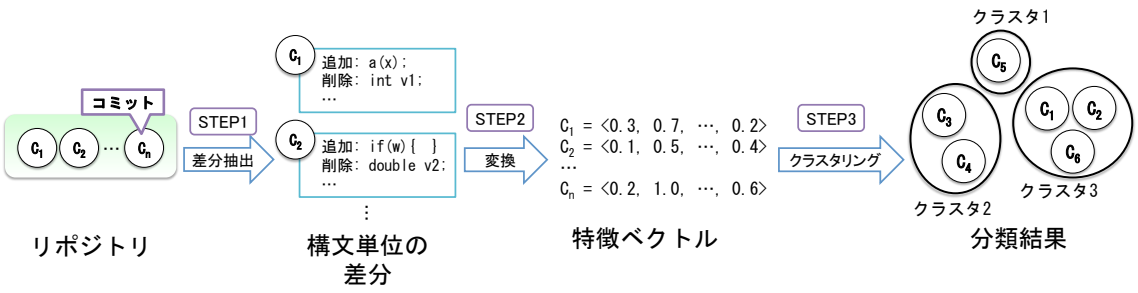


図 2: コミット分類

3 提案手法

本節では， タスク単位でのコミット分類を実現するための提案手法について述べる．

提案手法では，バージョン管理システムのリポジトリを入力とし，そこに蓄積されたコミットをタスク単位で分類した結果を出力する．このとき，分類は各コミットで行われたソースコードへの変更内容，すなわちリビジョン間でのソースコードの差分に含まれる識別子の出現回数にもとづいて行われる．

提案手法の処理の流れを図 2 に示す．この図に示すように，提案手法によるコミットの分類は以下の 3 ステップからなる．

STEP1 リポジトリから抽出した各コミットでのソースコードに対する変更内容を，単純な行単位ではなく，構文として意味のある単位 (以後 **構文単位の差分** と呼ぶ) で解釈した上で取得する．

STEP2 STEP1 で得られた構文単位の差分それぞれについて，含まれる識別子を全て取得し，それぞれ英単語として意味のある単位に分割した上で，分割結果の単語の出現回数を元にした **特徴ベクトル** というベクトルの生成を行う．

STEP3 生成した特徴ベクトルをコミットの特徴量としてクラスタリングを実行し，これをタスク単位での分類を行う．すなわち，1 つのクラスタには，1 つのタスクを実現したコミット群が含まれているとみなす．

以降，全てのソースファイルに対する操作を，ファイルに対する記述内容の変更として考える．すなわち，ファイルの追加と削除に関しては，それぞれ，空のファイルに対してファイルの全内容を追加する変更，ファイルの全内容を削除して空の内容のファイルにする変更とみなす．また，ファイル名の変更は，ファイルの内容に対しての変更を行わない操作とみなす．これにより，コミットにおけるソースコードに対する操作を，ファイルの追加と削除を個別に考慮することなく，変更という処理として一元的に行うことが可能となる．

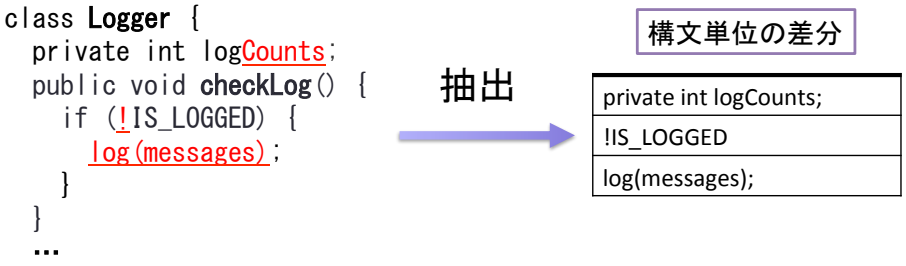


図 3: ソースコード変更例

以降の小節で，各手順についての詳細を述べる．

3.1 (STEP1) 識別子群の抽出

本手法ではまず，各コミットについて，構文単位の差分を抽出し，さらにそれらに含まれている識別子を抽出する．

3.1.1 抽出の流れ

まず，各コミットにおいて，そのコミットで変更されたソースファイルを特定し，それぞれのソースファイルについて，Change Distilling [20] と呼ばれる手法を用いることで，リビジョン間での構文単位の差分を求める．

次に，それら構文単位の差分に含まれる識別子を抽出する．このとき，その差分に含まれる識別子と合わせて，その差分を含むメソッド名およびクラス名の抽出も行う．変更の対象がフィールドかメソッドの宣言ならば，それらが含まれるクラスの識別子のみ抽出する．

以上のようにして，最終的に全ての変更に対してコミットごとで識別子群を得る．

3.1.2 抽出の例

ここでは，図 3 に示す例を用いて，識別子群の抽出方法を述べる．図 3 は，Java で記述されたあるプロジェクトにおいて，あるコミットで行われた，A というクラスに対する変更内容を示している．図中の赤字部分はそのコミットで削除された内容を表している．

まず，メソッド “ `checkLog` ” における，メソッド呼び出し “ `log(messages);` ” の削除を考えると，これは文の削除である．そのため，そのまま Change Distilling [21] において文の削除 (Statement Delete) という種別の構文単位の差分として解釈される．よって，差分に含まれる識別子として “log” 及び “messages” が抽出される．さらに，抽出された識別子が含まれていた差分 “ `log(messages)` ” がクラス “Logger” のメソッド “checkLog” 内に存在することより，“Logger” と “checkLog” も同時に識別子として抽出される．すなわち，

“ `log(messages);` ” の削除に対応して抽出される識別子は “log”，“messages”，“Logger”，“checkLog” である．

また，同様にメソッド checkLog において，条件式が修正され “ `if(IS_LOGGED)` ” となった箇所について考える．Change Distilling では，このような変更は条件式変更 (Condition Expression Change) として解釈され，“ `if(IS_LOGGED)` ” が構文単位の差分として抽出される．この差分に含まれる識別子は “IS_LOGGED” であり，また if 文はクラス Logger のメソッド checkLog 内に存在する．よってこの変更に対応して抽出される識別子は “IS_LOGGED”，“Logger”，“checkLog” である．

最後に，フィールド名の一部削除，すなわち記述 “ `private int logCounts;` ” の変更について考える．Change Distilling において，この変更に対する構文単位の差分は属性名変更 (Attribute Renaming) となり， `int v1;` そのものが構文単位の差分として抽出される．ここに含まれる識別子は “logCounts” のみで，フィールド宣言はメソッド内に存在しないが，クラス Logger の内部には存在しているため，変更に対応して抽出される識別子は “logCounts”，およびクラス名の “Logger” である．

最終的に，各変更に含まれる識別子を整理し，結果の識別子群とする．図 3 に示したクラス A に対する変更の場合，“log”，“messages”，“Logger”，“checkLog”，“IS_LOGGED”，“Logger”，“checkLog”，“logCounts”，“Logger” である．

3.2 (STEP2) 特徴語の抽出と特徴ベクトルの生成

次に，3.1 節 で抽出した識別子群から，特徴ベクトルを生成する．この生成は以下の 4 サブステップからなる．

STEP2A 識別子を英単語として意味のある単位で分割する．

STEP2B STEP2A で分割されたあとの単語をそれぞれ見出し語化する．見出し語化した後の単語を **特徴語** と呼ぶ．

STEP2C STEP2C で得られた特徴語それぞれについて，出現回数を集計する．

STEP2D STEP2D での集計結果を，特徴ベクトルへ変換する．

(STEP2A) 単語分割 抽出した識別子それぞれについて英単語として意味のある単位に分割する．このとき，識別子それぞれがキャメルケースあるいはスネークケースで記述されていることを前提とする．ここで，キャメルケースとは getYourName のような，単語の区切りとなる英字を大文字に，他を小文字とする記述法で，スネークケースは get_your_name のような，単語の区切りにアンダースコア (_) を用いる記述法である．また，各分割後の単語

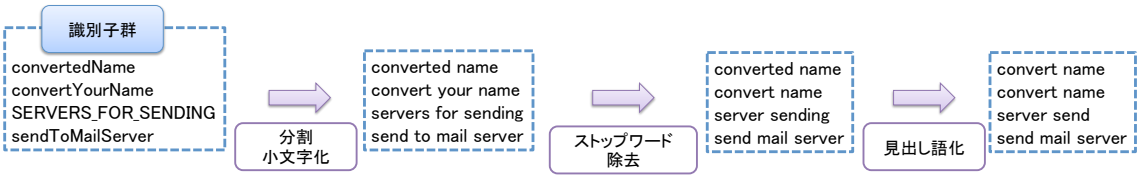


図 4: 識別子の英単語の見出し語化までの過程

については，後述する特徴語の出現回数の集計処理を簡便にするため小文字化する．例えば，“convertedName” と “MAX_COUNTS” という識別子があった場合，それぞれ，converted name と max counts という形で分割および小文字化される．

(STEP2B) 単語の見出し語化　次に，分割後のそれぞれの単語について，動詞や名詞などについては見出し語化を行う．見出し語化とは，動詞の活用や名詞の格変化で語尾の変化がある単語について，基本形に変形することである．先ほどの例の場合，converted と counts がそれぞれ convert と count になる．

ここで，図 4 へ，ここまでの流れの具体例を示す．この例では，“convertedName”，“convertYourName”，“SERVERS_FOR_SENDING”，“sendToMailServer” という 4 つの識別子群が 3.1 節 で示した手法で取得できたとして，特徴語群を抽出するまでを示している．

(STEP2C) 特徴語の出現回数集計　STEP2B で得られた特徴語群について，それぞれの出現回数を集計する．このとき，ストップワードについては集計を行わない．ストップワードとは，自然言語処理における用語で，前置詞や助動詞など，どのような文書でも現れやすいために文書の特徴を示しにくい語群を意味する．これらの単語は，文書分類において，分類の精度へ悪影響を与えるおそれがあるため，あらかじめ集計対象から排除される．よって，差分を文書として扱う本提案手法でも，ストップワードを特徴語群から排除する．

図 4 の例であれば，特徴語群に関して，出現回数は，最終的に convert が 2 回，name が 2 回，server が 2 回，send が 2 回，mail が 1 回という集計結果を得られることになる．

(STEP2D) 集計結果の特徴ベクトルへの変換　最後に，集計結果を，クラスタリングの際に特徴量となる特徴ベクトルへ変換する．

本研究ではコミットにおけるソースコードの差分を文書とみなし，2.3 節で取り上げた文書クラスタリングの手法を用いて，コミットを分類する．すなわち，差分内に含まれる識別子を構成する単語の出現回数を元に，Bag-Of-Words によりベクトルを作成し，このベクトルを特徴量としてクラスタリングを行う．また，分類精度向上のため，ストップワードの除去と，tf-idf 法も利用する．

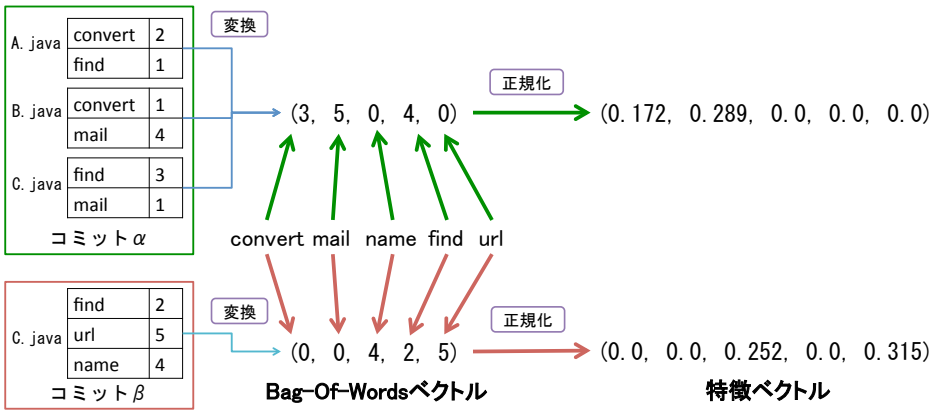


図 5: 特徴ベクトルの生成

それぞれのコミットで，各特徴語の出現回数を各要素とするベクトル (以下 **Bag-Of-Words(BOW) ベクトル** と呼ぶ) へと変換し，これを正規化して特徴ベクトルとする．このとき，BOW ベクトルの次元数は，全てのコミットで出現した特徴語の数と等しくなる．そのコミットで出現しない単語に対応する要素の値は 0 となる．

ここで，図 5 へ，BOW ベクトルの生成について，具体例を示す．図 5 には，リポジトリから 2 つのコミット α と β が得られたとして，それぞれについて BOW ベクトルを生成後，さらに正規化して特徴ベクトルへ変換する流れを示している．各コミットでは，変更があったソースファイルそれぞれについて特徴語の出現回数が集計されたとする．例えば，コミット α では，A.java についての変更では，特徴語 convert が 2 回，特徴語 find が 1 回集計されたことを示す．ここから，各コミットについて BOW ベクトルを生成する．まず，2 つのコミット全体で出現した全特徴語は convert，name，mail，find，url の 5 つであるため，BOW ベクトルの次元数は 5 となる．そして，各コミットについてファイル単位で集計した特徴語の出現数を合計し，BOW ベクトルへ変換する．例えば，コミット α であれば，A.java，B.java，C.java の変更について得られる特徴語の出現回数を全て合計すると，convert が 3 回，mail が 5 回，find が 4 回となり，url と name は出現していない．よって，図 5 に示すように BOW ベクトルは (3,5,0,4,0) となる．ただし，各要素は左から convert，mail，name，find，url の出現回数を表すものとする．また，コミット β についても同様に (0,0,0,2,5) という BOW ベクトルを生成できる．

続いて，生成した BOW ベクトルを正規化し，特徴ベクトルへ変換する過程を，コミット α で説明する．コミット α の BOW ベクトルは (3,5,0,4,0) であり，式 6，7 について，$N = 5$ ， $D = 2$ である．ここからまず tf 法で正規化したベクトルを求める． $\sum_j^N n_j = 3+5+0+4+0 = 12$ なので，例えば $i = 1$ については， $\mathrm{tf}_1 = \frac{3}{12} = 0.25$ である．同様にして他の i についても tf_i を求めて，正規化後のベクトル $\mathrm{tf}_v = (0.25, 0.417, 0.0, 0.333, 0.0)$

を得る．次に，idf 法で正規化したベクトルを求める．例えば $i = 4$ の時， $d_4 = 2$ であるから， $\mathrm{idf}_4 = \log \frac{|2|}{|2|} = 0.0$ となる．ゆえに，他の i についても idf_i を求めて，正規化後のベクトル $\mathrm{idf}_v = (0.693, 0.693, 0.693, 0.0, 0.693)$ を得る．

最後に，それぞれのベクトルの内積を求め，コミット α に対する特徴ベクトル $w = \mathrm{tf}_v \cdot \mathrm{idf}_v = (0.173, 0.289, 0.0, 0.0, 0.0)$ を得る．

3.3 (STEP3) クラスタリング

最後に，3.2 節 で生成した特徴ベクトルを用いて，クラスタリングを行う．アルゴリズムには，以下に述べる 2 つの条件を考慮した上で，2.2.1 節で説明した Repeated Bisection を選択した．

- 分類後のクラスタ数の指定が不要である．本手法では事前にタスク数，すなわち分類後のクラスタ数を特定できない．そのため，事前に分類後のクラスタ数を指定する必要があるクラスタリングは利用できない．Repeated Bisection は，クラスタ数ではなく，クラスタに対する評価関数 (後述) の値に対するしきい値を指定すればよい．
- スケーラビリティが高い． 開発が長期にわたって行われているプロジェクトは膨大なコミット数をリポジトリに蓄積している場合がある．例えば JRuby[6] という OSS であれば，2014 年 1 月 16 日現在で 20,531 のコミットが蓄積されている．したがって，本研究では，アルゴリズムにスケーラビリティが要求される．

分類後のクラスタ数を事前に指定する必要のないクラスタリングのアルゴリズムには，Repeated Bisection 以外にもいくつか存在する．しかし，それらは計算量が大きく，膨大なデータを処理するには大きな計算時間を要する．Repeated Bisection は他のクラスタリング数の指定が不要なクラスタリングアルゴリズムに比べて高速である [8] ことから 1 つ目の条件と 2 つ目の条件を同時に満たす．

[6] http://jruby.org/

4 評価実験

本節では 3 節 で述べた手法に対する評価実験について記述する．具体的には以下の 2 種類の実験を行う．

提案手法による分類結果の確認 いくつかのリポジトリに対して提案手法を適用してコミット分類を行い，どのような結果が得られるかを確認する

タスクとクラスタの対応関係の調査 タスク管理システムで管理されているプロジェクトについて，提案手法で得られる分類結果での各クラスタがタスクとどの程度対応できているかを調査する

各実験でのコミット分類は，各コミットにおいて出現する特徴語の抽出までを筆者が行い，Repeated Bisection によるクラスタリングには Bayon[7] を利用した．また，クラスタに対する評価関数のしきい値 `eps` には 1.0 を指定した．

以降，各実験の説明をそれぞれ行う．

4.1 提案手法による分類結果の確認

本実験では，4 つの OSS のリポジトリに対して提案手法を適用し，生成されるクラスタ数と，分類結果からタスクと対応しているクラスタとそうでないクラスタの具体例を挙げる．

実験対象として，Java で記述され Git で管理されている OSS のうちこれまでに行われたコミット数が多い (5,000 以上) ものから 4 つを選択し，提案手法によりコミットの分類を行った．

ここで，分類対象とならなかったコミットが発生しているが，これはそのようなコミットが以下に列挙する理由によって分類対象から排除されたからである．

表 1: 分類結果

プロジェクト名	コミット数	分類対象となったコミット数	クラスタ数
Lucene/Solr[8]	11,159	7,341	919
Jenkins CI[9]	17,640	8,452	1,102
WildFly[10]	14,280	10,247	1,204
JRuby	20,531	13,142	1,487

[7] `http://code.google.com/p/bayon/`
[8] `https://lucene.apache.org/solr/`
[9] `http://jenkins-ci.org/`
[10] `http://www.wildfly.org/`

- Java のソースファイルに対する変更が含まれていなかった
- Java のソースファイルに対する変更があっても，コメント部分の変更など，識別子を含まないものであった

以下，実験結果のクラスタのうち，タスクと対応していると考えられる例と，そうでない例をそれぞれ説明する．

タスクと対応したクラスタの例　まず，タスクと対応していると考えられるクラスタの例を示す．表 2 は，Lucene/Solr についてコミット群を分類した結果におけるあるクラスタについて，そのクラスタを構成する 4 つのコミットについての情報を示したものである．以下，望ましい分類が行われているとする理由を示す．

ここで，コミット ID が e4a64f5，5adc910，c5a985e のコミットについては，それぞれのコミットコメントから，Lucene/Solr の開発で利用されているタスク管理システムで，SOLR-4275 [15] というタスク ID を割り当てられたタスクと関連しているコミットであるとわかる．

また，コミット ID が a3e95d0 のコミットは差分の内容を確認すると，コミットコメントでも言及されているように TokenTokenizer クラスへ，オフセットの利用を目的として処理を記述したことがわかる．さらに，その記述で生じたバグに対して，コミット ID が c5a985e のコミットで修正を行ったことが分かる．

よって，TokenTokenizer クラスのオフセット機能の実装というタスクを考えた時，a3e95d0 のコミットはコミットコメントでタスク ID の記述がないが，他 3 つのコミット e4a64f5，5adc910，c5a985e と同じタスクの作業であるといえる．

まとめると，このクラスタは 1 つのタスクに関連したコミットのみで構成されているといえ，よってタスクと正しく対応しているといえる．

表 2: 望ましい分類結果の例 (Lucene/Solr)

コミット ID	修正されたソースファイル	コミットコメントに記述されたタスク ID
e4a64f5[11]	TestTrie.java	SOLR-4275
5adc910[12]	TestTrie.java	SOLR-4275
c5a985e[13]	TrieTokenizerFactory.java	SOLR-4275
a3e95d0[14]	TrieTokenizerFactory.java	記述なし

[11] https://github.com/apache/lucene-solr/commit/e4a64f5
[12] https://github.com/apache/lucene-solr/commit/5adc910
[13] https://github.com/apache/lucene-solr/commit/c5a985e
[14] https://github.com/apache/lucene-solr/commit/a3e95d0
[15] http://issues.apache.org/jira/browse/SOLR-4275

タスクと対応していない例　次に，1 つのタスクに対応していないと考えられるクラスタの例を示す．表 3 に，Jenkins CI について，コミットを分類した結果でのあるクラスタを構成する 3 つのコミットの情報を示す．以下，分類結果が望ましくないとする理由を述べる．

まず，IOUtils クラスに対する変更を行っているコミット ID が b3553d6 と 7d0bac1 のコミットについて述べる．IOUtils クラスはソフトウェア内での入出力処理に関するユーティリティクラスであるため，機能が独立した静的なメソッドのみで構成されている．そのため，それぞれのメソッドの実装は異なるタスクといえる．コミット ID が b3553d6 と 7d0bac1 のコミットで行われた変更内容を見ると，それぞれのコミットは異なるメソッドについての実装である．そのため，これらは異なるタスクについての実装であるといえる．

さらに，ID が a545a39 のコミットで行われた変更内容を確認すると，Launcher クラスの内部クラスについてアクセスレベルの変更を行っている．これは，コミットコメントから判断できるように，該当の内部クラスについてシリアライズを行うためのものであり，入出力関係の処理を実装している同じクラスタに分類された他の 2 つのコミットとは関連がない．

このように，表 3 に挙げたクラスタに含まれている 3 つのコミットは，それぞれ異なる実装内容を実現している．そのため，このクラスタは 1 つのタスクのみに関連している，すなわちタスクに対応しているクラスタとはいえない．

4.2　タスクとクラスタの対応関係の調査

本実験では，タスク管理システム JIRA[16] で管理されている，4 つのプロジェクト WildFly[17]，HornetQ[18]，RichFaces[19]，Weld[20] を対象として提案手法で得られた分類結果で評価値を測定し，それらの比較を行うことでタスクとクラスタの対応関係を調査する．

表 3: 望ましくない分類の例 (Jenkins CI)

コミット ID	修正されたソースファイル	コミットコメント
b3553d6	IOUtils.java	added a convenience method
7d0bac1	IOUtils.java	doh
a545a39	Launcher.java	for serialization work these interfaces need to be public

[16] https://www.atlassian.com/software/jira
[17] http://wildfly.org/
[18] http://www.jboss.org/hornetq
[19] http://www.jboss.org/richfaces
[20] http://weld.cdi-spec.org/

4.2.1 JIRA でのタスク管理

JIRA では，タスクを Issue という単位で管理する．また，Issue 同士は以下の 2 種類の形式で関連付けを行える．

IssueLink デフォルトで用意された，あるいは開発者が定義する関連

サブタスク ある Issue に対して，より詳細化された複数の Issue

このように，JIRA における Issue とタスクは同等のものと考えることができる．よって以降，JIRA における Issue についてもタスクと呼ぶ．また，タスク同士の関連を種別ごとに区別しているため，タスクとクラスタの対応関係を種別ごとに測定できると考えられる．

今回，IssueLink の各関連およびサブタスクで関連付けられているタスク同士を，同一タスクに属するタスク群とみなし，これらのタスク群と関連するコミットが分類結果において全て同一のクラスタに含まれているかどうかで，提案手法を評価する．

ここで，前述のように JIRA では IssueLink においてタスク同士の関連性を開発者側で独自に定義できる．そのため，複数のプロジェクトに対する実験結果を比較する場合，同一の基準での比較ができない可能性がある．これについて，今回は対象とするプロジェクトを，全て JBoss[21] というコミュニティで開発されているものに限定した．JBoss で開発されている OSS については，全て同じ JIRA のシステムで管理されている．そのため，開発対象が異なっても，タスク同士の関連付けには同一の関連性が用いられている．ゆえに，プロジェクト間での結果の比較を同一の基準で行えると考えた．

4.2.2 実験の流れ

以下，具体的な実験の流れを述べる．実験は以下の 3 つのステップからなる．

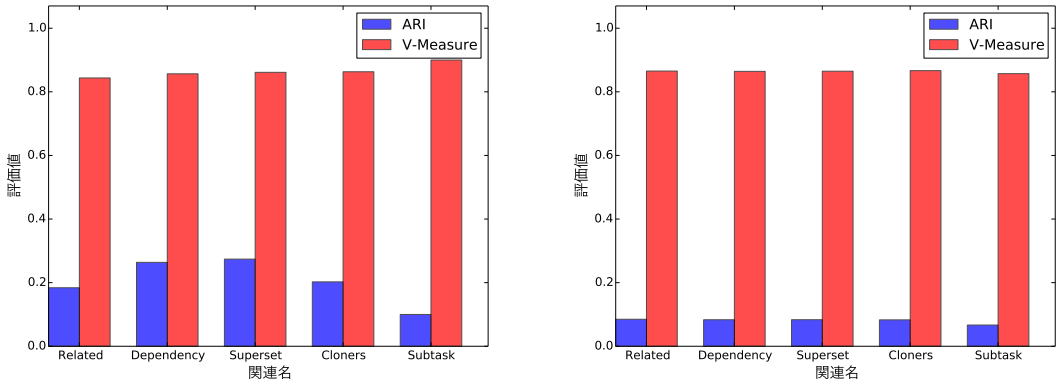
STEP1 リポジトリ中のコミットのうち，コミットコメントで対応するタスクのタスク ID が記述されているものを抽出する．

STEP2 STEP1 で抽出されたコミット群について，同一のタスクに属してると考えられるコミット同士でグループを作る．生成されるグループはそれぞれ同じタスクに関連したコミットで構成されている．

STEP3 STEP2 で作成したグループの集合を正解集合として，提案手法での分類結果と比較した際の評価値を求める．

STEP2 において，コミット群が，同一のタスクに属していると判断する基準は以下である．

[21] http://www.jboss.org/



(a) WildFly

(b) HornetQ

(c) RichFaces

(d) Weld

図 6: プロジェクトごとのタスクとクラスタの対応関係

- コミットコメントで記述されたタスク ID が全て同じである
- コミットコメントで記述されたタスク ID は異なるが，相互に IssueLink かサブタスクの形式で関連付けがなされている

2 番目の基準については，JBoss コミュニティで用いられている JIRA の IssueLink で定義されている関連には，Cloners，Related，Dependency，Superset の 4 つがあるため，これにサブタスクを加えた 5 つの関連について，それぞれで STEP3 での評価値を求める．

また，STEP3 については，2.2.2 節で述べた ARI と V-Measure を用いる．

4.2.3 実験結果

図 6 へ，各プロジェクトについての実験結果を示す．WildFly(図 6(a)) については各関連ごとに ARI の値の差はあるが，他の 3 つのプロジェクトに関しては差がないといえる．

5　考察

4 節 で得られた結果を用いて，タスクとの対応関係と他の手法に対する提案手法の有効性を考察する．さらに，変更が多岐にわたるタスクというものを考えてこれが提案手法によって正しく分類できるかどうかも考察する．

5.1　タスクとの対応関係

4.2 節で得られた結果から，提案手法で得られる結果での各クラスタとタスクとの関係がどの程度のものであるかを考察する．

4.2.3 節で示した結果から，WildFly(図 6(a)) 以外は関連の種別にかかわらず安定した割合でタスクと関連するコミット群とクラスタに含まれるコミット群が対応しているといえる．また，WildFly についても他のプロジェクトでの結果と同様，関連の種別にかかわらず V-Measure の値が 0.8 以上で安定しているため，対応関係が厳密にとれなかったタスクについても，いずれかのクラスタに関連するタスクが全て含まれている可能性が高い．このため，提案手法で得られる各クラスタに含まれるコミット群は，常に一定以上の割合でいずれかのタスクに関連するコミット群を含むあるいは，それらと一致しているといえる．

すなわち，全てのクラスタはそれぞれ安定した割合でいずれかのタスクと対応しているといえる．

5.2　提案手法の有効性

提案手法の有効性について考察する．

5.2.1　コミットコメントの記述の不備への対処

4.1 節での結果から，提案手法がコミットコメントに不備がある場合でも正しくタスク単位に分類できているかを確認する．

表 2 で示したクラスタで考える．コミット ID が a3e95d0 のコミットでは，コミットコメントでタスク ID が記述されていない場合でも正しく分類できたことから，提案手法がコミットコメントに関係なくタスク単位での分類が可能であることがわかる．

5.2.2　コミットの実装内容の正確な把握

本手法が，コミットの実装内容の正確な把握に有効である点を述べる．4.1 節での 表 2 で示したクラスタについては，正しく 1 つのクラスタと対応しているため，コミットコメント

のないコミット IDa3e95d0 のコミットについて，実装内容を開発者がコードを記述した意図にそって把握できる．

このように，提案手法を用いた分類結果を利用することで，コミットコメントについて不備があるために実装内容を把握しにくいコミットについても，同時に分類された，1 つのタスクへ関連がある他のコミット群と比較することで正確に実装内容を把握できる．

5.3 変更が多岐にわたるタスク

4.1 節 での結果から，変更が多岐にわたるタスクというものを考え，このようなタスクの分類に対して手法が有効であるかを考察する．

あるコミット群について，各コミットでの差分を個別に把握するだけでは，それぞれのコミットで実現された実装内容の関連を把握しにくいが，実際には 1 つのタスクを実現しているものがある．このようなタスクのうちで，変更が多岐にわたるタスクというものを考え，提案手法でこれらのタスクがどの程度検出可能かを評価する．

まず，変更が多岐にわたるタスクの定義を述べる．実装上の問題から，単一のファイルのみに対する編集で完結しないタスクが存在する場合がある．このようなタスクが複数コミットで実現され，しかも各コミットで変更されるソースファイルが異なっている場合，主として変更されているファイルを一意に決定しにくいために，実装内容の把握が困難であると考えられる．

そこで，クラスタ内のコミットで編集対象となったファイルすべてについて，編集対象となったコミット数を考える．このコミット数が占めるクラスタ内の全コミット数に対する割合 (以下 **編集割合** と呼ぶ) が，0.5 を超えるファイルが存在しないクラスタを，変更が多岐にわたるタスクに対応するクラスタとみなす．

例えば，あるクラスタについて，x，y，z，w の 4 のコミットが分類され，各コミットで編集されていたファイルが表 4 のようになっていたとする．表 4 の各セルは，そのコミットで該当のファイルが編集されている場合を○で，そうでない場合を×で表現している．例

表 4: 変更が多岐にわたるクラスタの例

編集対象のソースファイル / コミット ID	A.java	B.java	C.java
x	○	○	○
y	×	○	×
z	×	○	×
w	○	×	×

えば A.java については，コミット x と w で編集されていることがわかる．このため，編集割合は各ファイルについて，○の数をクラスタ内のコミット数で割ったものとなる．このとき，各ファイルについての編集割合は，A.java は $\frac{2}{4} = 0.5$ ， B.java は $\frac{3}{4} = 0.75$ ， C.java は $\frac{1}{4} = 0.25$ となる．よって，ファイル B.java に関して編集割合が 0.5 を超えているため，表 4 のクラスタは， 変更が多岐にわたるタスクではないと判断する．

次に，変更が多岐にわたるタスクの例として，提案手法で実際に検出されたものを示す．4.1 節 で望ましい分類結果の例として挙げたクラスタが対応するタスクでは，4 つのコミットにより，TrieTokenizerFactory.java と TestTrie.java にまたがって編集が行われている．これらの編集割合はそれぞれ 0.5 であるから，このタスクは変更が多岐にわたるタスクである．

最後に，編集割合を利用し，変更が多岐にわたるクラスタを特定し，これらが各実験対象のプロジェクトについてどの程度存在しているかを測定する．結果は表 5 である．表から，実験対象となったプロジェクトについては，どれも 20%以上変更の多岐にわたるクラスタ，そしてそれらに対応するタスクが存在することが分かる．

以上から，変更が多岐にわたるタスクが，一定の割合でプロジェクト内で存在することがいえる．そのため，提案手法は変更が多岐にわたるタスクについて，自動で検出できるという点で有用性があるといえる．

表 5: 実験対象での変更が多岐にわたるクラスタの割合

プロジェクト名	クラスタ数	変更が多岐にわたるクラスタ数	割合 (パーセント)
Lucene/Solr	919	321	34.9
Jenkins CI	1,102	234	21.2
WildFly	1,204	457	38.0
JRuby	1,487	264	24.0

6　手法と結果の妥当性

本研究の手法と結果の妥当性について，以下で説明する点に留意する必要がある．

6.1　手法の妥当性

識別子を利用したことについてやコミットとタスクとの対応に関する妥当性を考える．

6.1.1　識別子の利用

提案手法ではソースコード中に出現する識別子が実装内容を反映していることを仮定している．このため，識別子に対する命名に不備がある場合，望ましい分類が得られない．以下，望ましい分類が得られないと考えられる命名の例を挙げて説明を行う．

省略語の利用　例えば environment という識別子と，それの表記を省略した env という表記の識別子を同時に利用した場合，特徴語として別のものとして認識されてしまうために，特徴語の出現回数を正しくカウントできないため，分類精度に影響を与える可能性がある．この問題については，省略語から元の語を復元する既存研究が存在しており，これを利用することで解決につながる可能性はある [22] .

実装内容を反映しない識別子の利用　変数がループなどで一時変数として利用される場合，識別子の表記は `foo` ， `bar` など，単体では実装内容を反映しないものが利用される場合がある．このような表記のされ方が，ソースコードにおいて，実装内容が関連しない箇所で複数回利用されていた場合，特徴語の出現回数及び最終的な分類精度に影響を与える可能性がある．

関連度の誤った導出　4.1 節では，望ましくない分類結果の例として表 3 のクラスタを挙げた．このクラスタでは，重心の結果から，クラスタと最も関連度の高い特徴語が io であることがわかった．このことから，表 3 で挙げたコミット群同士は，タスクという点で比較的関連度が低いものの，差分に io を含んだ識別子があるために，同じクラスタへ分類されたと考えられる．

このように，本手法ではタスクという点では関連度の低いコミット同士でも，同一の識別子を含むために同一のクラスタとして分類されてしまう場合がある．

6.1.2 特徴語の抽出

提案手法では識別子から特徴語を抽出する際，識別子がキャメルケースかスネークケースで表記されていることを前提としている．しかし，これによって意図しない特徴語の抽出が行われる場合がある．例えば JRuby という表記の識別子があったとする．ここで，この識別子をキャメルケースとして，特徴語を抽出すると j と ruby となる．しかし，元の識別子は， j の字そのものには意味はなく，jruby という表記によって意味をなすものであるため，この特徴語の抽出の仕方は誤っている．もし，ソースコードの他の場所で j という表記の識別子が利用されていた場合，意図せず多く j という特徴語の出現回数をカウントする結果となる．

このような問題を解決する手法としては，抽出された特徴語間の共起確率を利用することで，意図した形で識別子を抽出することが考えられる [22] ．先程の例であれば，一旦は j と ruby に分割されるものの，お互いが同時に出現する確率が高いならば， jruby という 1 つの特徴語として抽出されることになる．

6.1.3 コミットにおける複数タスクの混在

本手法では，タスクとコミットが 1 対 1 で対応していることを前提としている．そのため，複数の独立したタスクが 1 つのコミットに混在している場合は分類結果として不正なものが出る場合がある．

6.2 結果の妥当性

実験結果についての妥当性を述べる．

6.2.1 実験対象

本研究における実験では，コミット分類の対象として OSS を利用している．OSS は不特定多数の開発者が，不定期に開発に携わるために，企業などでのソフトウェア開発とは異なった開発工程を経ている可能性がある．そのため，本研究における提案手法がそのまま利用できるとは限らない可能性がある．また，OSS であっても，各コミットの差分において，これまでの小節で説明した識別子の利用やコミットにおける複数タスクの混在に起因する誤った分類結果が出力されうる．

7　あとがき

本研究ではソフトウェア開発において，タスク単位での実装内容の把握が有用であることを踏まえ，これを支援するために，バージョン管理システムのリポジトリに蓄積されたコミットをタスク単位で分類した．

その結果，タスク単位での分類として望ましい結果が存在することを確認した．

今後の課題は，以下のようにして，分類精度を向上させることである．

- 分類精度向上のための，識別子からのより正確な特徴語抽出
- 複数のタスクに関連したコミットに対応した分類

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究に関して，有益かつ的確なご助言を頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

本研究において，多大なるご助言を頂きました 井垣 宏 特任助教に深く感謝申し上げます.

本研究に用いたツールの大部分を設計, 実装してくださり，また本研究に関して多大なるご助言，ご助力を頂きました，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研究員の 堀田 圭佑 氏に深く感謝申し上げます.

本報告を行うにあたり，多大なるご助言，ご助力を頂きました，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博士後期課程 1 年の 楊嘉 晨 氏に深く感謝申し上げます.

大阪大学基礎工学部情報科学科計算機科学コース 4 年の 三谷 康晃 君並びにその他の楠本研究室の皆様 のご助言，ご協力に心より感謝致します.

また，本研究に至るまでに，講義，演習，実験等でお世話になりました大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます.

参考文献

[1] A. Murgia, G. Concas, M. Marchesi, and R. Tonelli. A machine learning approach for text categorization of fixing-issue commits on cvs. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 6:1–6:10, 2010.

[2] N. Dragan, M.L. Collard, M. Hammad, and J.I. Maletic. Using stereotypes to help characterize commits. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, pp. 520–523, 2011.

[3] C.C. Williams and J.K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, pp. 466–480, 2005.

[4] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pp. 121–124, 2008.

[5] A. Hindle, D.M. German, M.W. Godfrey, and R.C. Holt. Automatic classication of large changes into maintenance categories. In *Proceedings of the 2009 IEEE 17th International Conference on Program Comprehension*, pp. 30–39, 2009.

[6] L.P. Hattori and M. Lanza. On the nature of commits. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering 2008.*, pp. 63–71, 2008.

[7] B. J. Frey and D. Dueck. Clustering by passing messages between data points. *Science*, pp. 972–976, 2007.

[8] G. Karypis. CLUTO - a clustering toolkit. Technical report, Digitical Technology Center, 2003.

[9] Y. Zhao and G. Karypis. Criterion functions for document clustering: Experiments and analysis. Technical report, Digitical Technology Center, 2002.

[10] J. M. Santos and M. Embrechts. On the use of the adjusted rand index as a metric for evaluating supervised classification. In *Proceedings of the 19th International Conference on Artificial Neural Networks: Part II*, pp. 175–184, 2009.

[11] A. Rosenberg and J. Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pp. 410–420.

[12] 和明岸田. 情報検索の理論と技術. 勁草書房, 1998.

[13] 大折原, 彰内海. Html タグを用いた web ページのクラスタリング手法. 情報処理学会論文誌, pp. 2910–2921, 2008.

[14] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[15] 大蔵君治, 井垣宏. タスク管理システムと連動するソフトウェア開発データ計測システムの提案. ウィンターワークショップ 2009・イン・宮崎 論文集, pp. 13–14, 2009.

[16] A. Kuhn, S. Ducasse, and T. Grba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, pp. 230–243, 2007.

[17] J. Koskinen, A. Salminen, and J. Paakki. Hypertext support for the information needs of software maintainers. *Journal of Software Maintenance and Evolution*, pp. 187–215, 2004.

[18] T. Hofmann. Probabilistic latent semantic indexing. In *Proceedings of the 22Nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 50–57, 199.

[19] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein. Studying software evolution using topic models. *Science of Computer Programming*, pp. 457–479, 2014.

[20] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, pp. 725–743, 2007.

[21] B. Fluri and H.C. Gall. Classifying change types for qualifying change couplings. In *Proceedings of the 2006 14th IEEE International Conference on Program Comprehension*, pp. 35–45, 2006.

[22] D. Lawrie, D. Binkley, and C. Morrell. Normalizing source code vocabulary. In *Proceedings of 2010 17th Working Conference on Reverse Engineering*, pp. 3–12, 2010.