Vol.2018-SE-198 No.30
2018/3/10

トピックの変化量に着目したソースコードの変更量分析に関する考察

矢野 博暉[1,a)]　阿萬 裕久[2,b)]　川原 稔[2]

概要：本論文では，ソースファイルの内容に対するトピック分析を行い，その結果を使ってソースファイルの特徴をベクトルのかたちで表現している．そして，ソースコードの変更が行われた場合の変更量を，単純なソースコードの差分ではなく，変更前後のソースファイルに対応した特徴ベクトル間の距離を使って評価する手法を提案している．提案法の有効性を確認するため，四つのオープンソースソフトウェアを使った評価実験を行い，ソースコードの変更と，その後（1 ヶ月の間）にフォールト修正が行われたかどうかの関係を分析している．その結果，提案法によるソースコード変更量評価は，フォールト修正の起こりやすさを予測する上で有用な予測子の一つとなることが確認されている．

A Study of Source Code Change Analysis Focusing on Change in Topics

Yano Hiroki[1,a)]　Aman Hirohisa[2,b)]　Kawahara Minoru[2]

1. はじめに

一般にソフトウェアシステムは，そこでの機能拡張・変更とフォールト（俗にバグと呼ばれる）の修正を繰り返しながら発展していく．つまり，さまざまな目的の下でソースコードの変更が行われながら，より高機能・高性能でより高品質なソフトウェアが作られていく．保守の過程においては，機能の拡張や変更を目的としたソースコードの変更のみが行われるのが理想的ではあるが，フォールトを全く含まない完全なソフトウェアを早期にリリースするというのは実際には難しく，上述したように機能の拡張・変更と並行してフォールトの修正も行っていくというのが現実的である．

それでもなお，フォールトの作り込みは少ない方が望ましいことは言うまでもない．ソースコードの変更作業のうち，約 7% では新たなフォールトを作り出してしまっていることもあると言われている [1]．結果的に，フォールト修正により多くの工数（コスト）がかかってしまい，保守コストの増大につながってしまうことも考えられる [2], [3], [4]．ソースコードに対する変更は必要不可欠ではあるが，このようにフォールトの作り込みや保守コスト増大のリスクも伴う．それゆえ，ソースコードの変更量とその影響については可能な限り正確に把握し，適切な変更管理及びレビューやテストの計画に反映すべきである．

ソースコードの変更という概念は，プログラム記述の観点での変更と意味的な観点での変更に大別できる．前者は，例えばソースコードの変更行数（code churn）でもってその大きさが評価され，その値はフォールト潜在の疑わしさに関係があるという報告も多い [5], [6]．後者は，例えばプログラム構造の違いに着目した評価法 [7] があり，単なる字句的な違いとは異なる視点からソースコード変更をより正確にとらえようとしている．近年，後者に関連して，ソースコードの内容に対して自然言語処理技術を応用する研究も盛んに行われるようになってきている．例えば，山本 [8] はソースファイルに対して Doc2Vec [9] を利用し，

[1] 愛媛大学大学院理工学研究科電子情報工学専攻
Electrical and Electronic Engineering and Computer Science Course, Graduate School of Science and Engineering, Ehime University, 3, Bunkyo-cho, Matsuyama, Ehime 790–8577, Japan
[2] 愛媛大学総合情報メディアセンター
Center for Information Technology, Ehime University, 3, Bunkyo-cho, Matsuyama, Ehime 790–8577, Japan
[a)] yano@se.cite.ehime-u.ac.jp
[b)] aman@ehime-u.ac.jp

ベクトル表現を用いてソースファイル間の類似性を評価することを提案している．Thomas ら [10] は，ソースファイルに対してトピック分析 [11], [12] を行い，トピックベクトル（特徴ベクトル）の近さでもってソースファイル間の内容の類似度を評価し，回帰テストの際のテストコード選択に役立てることを提案している．

このように先行する研究では，ソースファイル（ソースコード）に対して自然言語処理技術を適用し，その内容をベクトルのかたちに変換して類似性を評価したり活用したりするものが多く見られる．ここで逆に，そのようにして得られるベクトルの非類似性（ベクトル間の距離）に着目すれば，ソースコード間の違いが定量化されることになると考えられる．これが本論文における研究動機である．つまり，ソースコードの変更前後の内容をそれぞれいったんベクトル化し，ベクトル間の距離でもってそこで施されたソースコード変更の量を評価するという方法も可能ではないかというアイデアである．ベクトル化の手法によってその意味合いは異なるところではあるが，そのうちの一つとして，本論文ではトピック分析（トピックモデル）を活用することにする．詳しくは後述するが，トピックモデルでは文書間（ここではソースファイル間）で共通して登場する単語に着目した分析が行われるが，その際には間接的な共起関係も考慮されるため，表記のゆらぎにも対応した類似性（非類似性）評価が可能である．それゆえ，異なる開発者によって開発・修正されたソースコードに対しても適用しやすいのではないかと考えたため，本論文ではトピックモデルを利用することとした．

本論文の構成は以下の通りである．2 章では本論文で利用するトピックモデルについて概説する．そして，3 章でトピックモデルを用いたソースコードの変更量評価法を提案し，4 章でオープンソース開発プロジェクトを対象とした評価実験とその結果について報告する．最後に 5 章で本論文のまとめと今後の課題について述べる．

2. トピックモデル

トピックモデルとは，文書の集合において登場する抽象的な“トピック”を見出すための統計モデルである．ただし，ここでいうトピックは，明示的に与えられるカテゴリではなく，あくまでも単語集合に割当てられるラベルの意味である．つまり，あらかじめ決められたカテゴリに文書を分類するというイメージよりも，文書の中に登場する単語について緩やかなクラスタリングを行い，各文書が各クラスタにどの程度関係しているかを分析するというイメージである．**図 1** に直感的なイメージ図を示す．この図では 4 個の文書があり，そこに全部で 8 種類の単語（$w_1 \sim w_8$）が登場しており，これらの共起関係から緩やかに 3 個のクラスタがトピックとして形成されている．同図では例えば，文書 1 がトピック 1 及び 2 に関係している．このように，一つの文書が一つの特定のトピックに結びつけられるとは限らず，複数のトピックにまたがった関係性が見られることが多い．

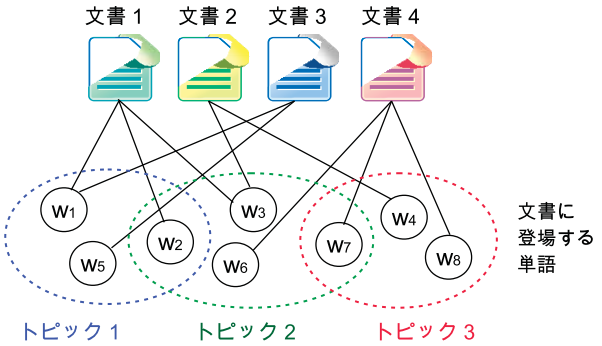


図 1 文書とトピックの関係（イメージ図）

表 1 共起行列 X の例

	単語の登場回数					
	edit	run	error	bug	pitcher	base
文書 1	10	10	5	0	0	0
文書 2	10	5	0	10	0	0
文書 3	0	10	0	0	10	5
文書 4	0	5	5	0	5	5

次に，このようなトピックモデルの構築について簡単に紹介する．

(1) 潜在的意味解析

まず，単語の共起性に着目し，それによって創発される情報を潜在的意味という [12]．例えば，“edit”, “compile”, “run”, “error” といった単語から我々はプログラミングに類することを想像するかもしれない．これが潜在的な意味ということになり，潜在的な意味のカテゴリがトピックということになる．実際には，これらがすべて同一の文書に登場する（共起する）とは限らず，一部のみが登場したり，“error” ではなく “bug” や “failure” といった関係する別の単語が登場することもある．また，別のトピックに同じ単語が登場することもある．例えば，スポーツに類する文書の中にも “run” という単語が登場しても不思議ではない．

潜在的意味解析では，まず文書から単語を抽出し，それらの登場回数を整理した共起行列 X を用意する．**表 1** に X の例を示す．ここでは文書 1 及び 2 はプログラミング関係，文書 3 及び 4 は野球関係の内容を想定している．そして，X に対して特異値分解を行って，トピックの解析を行う．具体的には，X を次のように特異値分解する：

$$X = USV^T$$

ただし

$$U = \begin{bmatrix} 0.611 & 0.245 & -0.725 & -0.201 \\ 0.505 & 0.611 & 0.589 & 0.154 \\ 0.513 & -0.639 & 0.343 & -0.458 \\ 0.328 & -0.397 & -0.093 & 0.852 \end{bmatrix},$$

$$S = \begin{bmatrix} 21.757 & 0 & 0 & 0 \\ 0 & 14.716 & 0 & 0 \\ 0 & 0 & 8.093 & 0 \\ 0 & 0 & 0 & 4.428 \end{bmatrix},$$

$$V = \begin{bmatrix} 0.513 & 0.582 & -0.168 & -0.104 \\ 0.708 & -0.195 & -0.166 & -0.352 \\ 0.216 & -0.052 & -0.506 & 0.735 \\ 0.232 & 0.415 & 0.728 & 0.349 \\ 0.311 & -0.569 & 0.366 & -0.073 \\ 0.193 & -0.352 & 0.154 & 0.444 \end{bmatrix}.$$

特異値分解によって得られた行列では，U の行が文書に対応し，S の行と列，及び V の列（V^T の行）がトピックに対応する．そして，V の行（V^T の列）が単語に相当する．

ここでトピック数を 2 に限定したとする．これは U の左から 2 列のみ，S の左上の 2 行 2 列のみ，及び V の左から 2 列のみに着目することになる．これらをそれぞれ $\tilde{U}$, $\tilde{S}$ 及び $\tilde{V}$ と表す．これらの積によって，元の共起行列 X の低ランク近似 $\tilde{X}$ が得られることになる：

$$\tilde{X} = \tilde{U}\,\tilde{S}\,\tilde{V}^T = \begin{bmatrix} 8.92 & 8.71 & 2.68 & 4.59 & 2.08 & 1.30 \\ 10.87 & 6.03 & 1.91 & 6.29 & -1.70 & -1.04 \\ 0.26 & 9.75 & 2.90 & -1.31 & 8.83 & 5.47 \\ 0.27 & 6.20 & 1.84 & -0.77 & 5.55 & 3.44 \end{bmatrix}.$$

ここで注目すべきは， 1, 2 行目（文書 1, 2 に対応）の 3 列目（error に対応）と 4 列目（bug に対応）である．元の X では，error と bug は同じ文書では登場していないが，この解析を通じてどちらにも正の値が割り振られている．こうすることで，表現のゆらぎがあってもトピック分析を通じて関連性を見ることができる．

なお，文書とトピックの関係は行列 U （あるいはその限定版である $\tilde{U}$）に現れており，1 列目と 2 列目がそれぞれトピック 1 と 2 に対応している．この例の場合，トピック 1 は四つの文書すべてに同程度にまたがっているが，トピック 2 は文書 1, 2 と 3, 4 で明らかに分かれている（前者が正なのに対し，後者は負となっている）．このような違いが，元の X におけるプログラミングと野球の違いを反映していると思われる．

(2) 統計的潜在意味解析

(1) で説明した潜在的意味解析は基本的な方法ではあるが，特異値分解に基いているためトピックどうしがベクトル空間上で直交している必要があり，そのことが一種の制約となっている．その点で，トピック分析として柔軟性に欠けるという短所がある．本論文では説明の都合上，いったん特異値分解による分析例を示したが，今日ではこれに確率モデル（階層ベイズモデル）を導入した方法が一般に使われている．それらを統計的潜在意味解析という．

これは，直感的に言えば次のような考え方である．いくつかのトピックがあった場合に，トピックごとに各単語の出現確率（確率分布）は異なる．つまり，ある確率分布（ただし陽には観測されない）に従って各トピックでは各単語が出現する．これに対し，実際には各文書において各単語が観測される．したがって，観測された（結果として得られた）単語をもとにして，その発生源であるトピックを推定しようというものである．詳細については文献 [11], [12] 等を参照されたい．

一般に多く使われているモデルとしては，確率分布としてディリクレ（Dirichlet）分布を使用した Latent Dirichlet Allocation (LDA) が挙げられる．本論文ではこれの実装として，統計処理ソフトウェア R における topicmodels パッケージの LDA 関数を用いた．

表 1 に示した共起行列 X の例について，LDA によるトピック分析の結果を **表 2** 及び**表 3** に示す．

表 2 は，各トピックに対して各単語がどの程度関係しているかを表している[*1]．“run” と “error” という二つの単語に関しては，トピック 1 と 2 の両方に関係しているが，それら以外の単語はトピック 1 か 2 のいずれか一方のみとなっている．実際に，“run” と “error” という単語だけに注目すると，プログラミングと野球のいずれの話題に登場しても不思議ではない．

表 3 は，表 2 で示された単語とトピックの結び付き

表 2 表 1 の X に対する LDA での結果：トピックと単語との関係

	edit	run	error	bug	pitcher	base
トピック 1	0.400	0.300	0.100	0.200	0	0
トピック 2	0	0.333	0.111	0	0.333	0.222

表 3 表 1 の X に対する LDA での結果：文書とトピックの関係

	トピック 1	トピック 2
文書 1	0.999	0.001
文書 2	0.999	0.001
文書 3	0.001	0.999
文書 4	0.001	0.999

[*1] 表中に 0 がいくつか見られるが，正確には 10^{-43} 未満の数であったため 0 と表記している．

をもとに各文書におけるトピックの出現確率を算出した結果となっており，文書ごとに合計が 1 となるようになっている．この例の場合は明らかに，文書 1 と 2 においてはトピック 1 が対応し，文書 3 と 4 においてはトピック 2 が対応している．この結果は，もともとの想定（文書 1, 2 はプログラミング関係，文書 3, 4 は野球関係）と合致している．

以上のようにして，文書に対するトピック分析を通じて，文書の特徴を数値化できる．つまり，表 3 に示したような結果が文書の特徴量（特徴ベクトル）である．本論文では，ソースファイルを文書に見立てて同様の数値化を行い，このようなかたちでソースファイルの特徴をベクトルで表現する．冒頭の 1 章で述べたように，そのようなベクトル表現を使ってソースファイルの類似性を評価するという研究がいくつか報告されている [8], [10]．本論文では，これをソースファイルの変更量評価に応用することを考える．これについては次章で述べる．

3. トピックモデルを用いたソースコード変更量評価

本章ではまず，ソースファイルを対象としたトピックモデルの構築について説明する．そして，トピック分析によって得られるソースファイルのベクトル表現を使って，ソースファイルの変更量を評価する手法を提案する．

3.1 トピックモデルの構築

ソースファイルを対象としたトピックモデルの構築手順を説明する．

(1) ソースファイルに前処理を施す．

ソースファイルの内容自体は一種のテキストデータであるため，そのまま単語（トークン）に分割してトピックモデルを構築することは可能である．しかしながら，ソースファイルの場合，記号や（プログラミング言語の）予約語も多く登場するため，それらも単語集合に含めるべきか否かには議論の余地がある．本論文では，ソースコードの意味的な側面に着目することを考えているため，記号や予約語は対象外とすることにする．また，数値も対象外とする．本来，数値にも何らかの意味があると思われるが，同じ数値が異なる意味（目的）でソースコード中に登場する可能性を否定できないため，それらが混在することの影響を排除するため今回は対象外とした．

次に，残った単語に対し，それらを原形に置換する．これは，単語によっては複数形や過去形で書かれている場合もあるため，それらを統一的に扱うことを目的としている．原形への変換には形態素解析のツールを利用することにした．これが可能なツールはいくつかあるが，Tian ら [13] の比較実験の結果を参考に，Tree Tagger [14] を使用することにした．

最後に，“the” や “is” といった英語におけるストップワード[*2]も除去する．

(2) 文書ターム行列（共起行列）を作成する．

ソースファイルに対して前処理を施した後に，単語（正確にはトークン）を抽出して文書ターム行列（共起行列）を作成する．文書ターム行列の作成例を **図 2** に示す．この例では二つのソースファイル A, B について，それぞれ前処理を経て得られた単語のリスト（同図の左）があり，それらをもとに文書ターム行列（同図の右）が作られるイメージを表している．文書ターム行列では，少なくとも一つのソースファイルに出現していた単語が列挙され，各ソースファイルにおける登場回数が記載されている．つまり，ソースファイルによっては登場しない単語もあり，その場合は 0 として行列に記載される．

このような文書ターム行列を入力として， LDA によるトピック分析が可能となる．

(3) LDA によってトピックモデルを構築する．

文書ターム行列を用意した後，前章で説明したように LDA を行いてソースファイル集合に対するトピック分析を行う．それにより，各ソースファイルに対して表 3 に示したようなトピック出現確率のベクトルが得られることになる．これが，ソースファイルの持つ特徴のベクトル表現となる．

ただし，LDA によるトピック分析を行うにあたっては，あらかじめトピック数を指定する必要がある．つまり，ベクトルの次元を先に決めてしまわなければならない．指定するトピック数が異なれば，得られる結果（ベクトル）も異なることになるため，適切なトピック数を選ぶ必要がある．一般に，トピック数が少なすぎると得られる情報の次元も低すぎることになり，結果としてベクトルではソースファイルの類似性や違いを適切に表現できない．一方，トピック数が多すぎると，ソースファイル間の些細な違いさえも強調されてしまい，クラスタリングとして不適切になってしまう．極端な場合，ソースファイル一つ一つに対して異なる

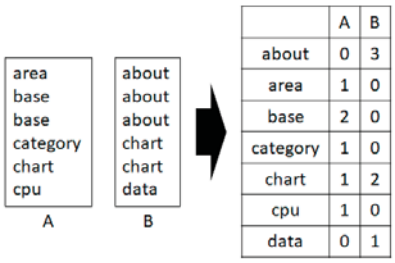


図 2 文書ターム行列作成のイメージ図

[*2] https://docs.oracle.com/cd/E16338_01/text.112/b61357/astopsup.htm#i634475

トピックが割当てられることになる．

そこでトピック数の決定にあたっては，いくつか指標が提案されている．代表的なものとして，CaoJuan らの指標 [15]，Arun らの指標 [16]，Griffiths らの指標 [17] 及び Deveaud らの指標 [18] がある．いずれも，指定されたトピック数に対して一つの指標値を与えるものとなっている．ただし，前者の二つは指標値が小さいほど良く，後者の二つは指標値が大きいほど良いという解釈になっている．

本論文では，トピック数を 1 から 50 まで変化させながら，上記の指標を使って適切なトピック数を決定する．**図 3** に一例を示す．この場合，四つの指標すべてに共通して評価の高いトピック数というものは見当たらず一意に決めることは難しいが，強いて決めるとすると，Deveaud らの指標以外では概ねトピック数の増加に対して単調に減少または増加していることから，指標値が比較的最適値（図の縦方向での中央）に近い範囲で Deveaud らの指標が急降下している 16 の手前，つまり 15 を一つの目安と考えることができる．決め方については議論の余地があるが，今回はこのようなかたちで決めることにし，さらなる検討については今後の課題としたい．

トピック数を指定した上でトピックモデルを構築し，各ソースファイルに対して，そこでの各トピックの出現確率を表したベクトルを得る．

3.2 ソースコード変更量の評価

本論文では，上述した方法によってソースファイルの内容をベクトルで表現する．そして，ソースコードの変更が行われた際に，変更前後のソースファイルについてそれぞれベクトル表現を取得し，それらの間の距離でもってソースコード変更の大きさを評価することを提案する．

これにあたって，変更前後のソースファイルのベクトル化について補足しておく．一般に，トピック分析では，入力として与えられた文書集合（この場合はソースファイル集合）に応じてベクトルが作られることになる．したがって，ソースコード変更が行われた場合，変更前と変更後では内容が異なるため，得られるトピックモデルも変わってくる．つまり，他の変更の無かったソースファイルにおいても，ベクトルに変化が生じてしまう恐れがある．例えば，**図 4** は簡単なイメージ図を示しており，三つのソースファイル s_1, s_2 及び s_3 あり，ここでは s_3 に対してソースコード変更が行われたとする．この場合，トピックモデルは変更前の $\{s_1, s_2, s_3\}$ に対してと変更後の $\{s_1, s_2, s_3\}$ に対してそれぞれ作られることになる．それゆえ，変更の無かった s_1 や s_2 に対して，どちらのトピックモデルを使うかによってベクトルに違いが生じる可能性がある．

この問題を回避するため，本論文では **図 5** に示すように変更前後のソースファイル両方を同時に入力としてモデル構築に使用し，変更の無かったソースファイルには影響が及ばないようにする．

次に，得られたベクトルの距離を算出する．どういった距離を採用すべきか絶対的な基準はないが，本論文では，自然言語処理の分野でよく見られるコサイン距離と Thomas ら [10] が採用しているマンハッタン距離の 2 種類をそれぞれ利用することにする．

コサイン距離は，コサイン類似度を 1 から減じたものである．ただし，コサイン類似度とは，ベクトル間の類似度を両者のなす角の余弦（コサイン）で表したものとなっている．いま，二つのソースファイル s_A と s_B があり，それぞれが n 次元ベクトル $\boldsymbol{x}_A = (x_{A1}, x_{A2}, \ldots, x_{An})$，$\boldsymbol{x}_B = (x_{B1}, x_{B2}, \ldots, x_{Bn})$ で表されているとする．このとき，両者のコサイン距離 $d_c(s_A, s_B)$ は次式で得られる：

$$d_c(s_A, s_B) = 1 - \frac{\left| \sum_{i=1}^{n} x_{Ai} \cdot x_{Bi} \right|}{\sqrt{\sum_{i=1}^{n} x_{Ai}^2} \sqrt{\sum_{i=1}^{n} x_{Bi}^2}} .$$

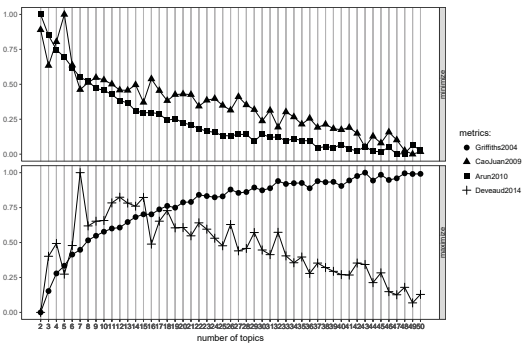


図 3 トピック数の決定例

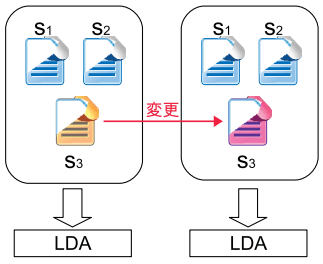


図 4 変更前後で別々にモデル構築

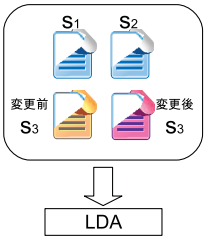


図 5 変更前後のものを一緒にモデル構築

同様に，マンハッタン距離 $d_m(s_A, s_B)$ は，ベクトルの各要素の差を累積したものとなっており，次式で得られる：

$$d_m(s_A, s_B) = \sum_{i=1}^{n} | x_{Ai} - x_{Bi} | .$$

4. 評価実験

4.1 目的と対象

本実験の目的は，前章で提案した手法，即ち，ソースコードの変更量をトピックベクトルの距離でもって評価する手法を実際にオープンソースソフトウェアへ適用し，その結果について検討することである．

実験対象としては，**表 4** に示す四つのオープンソースソフトウェアを使用した．いずれも GitHub において公開され，stars の多い著名なものとなっている．

4.2 手順

実験の大まかな流れを以下に示す（**図 6**）．

(1) まず，`git tag` コマンドを使用して，リポジトリに記録されているバージョン情報を確認する．そして，比較的開発が進んだ（初期でない）バージョンを注目バージョンとして，そこから 3 ヶ月間を調査期間とする．本実験では，リポジトリの最初のコミットから十分な時間が経過している時点でのバージョンを注目バージョンとする．一般に開発初期の段階では，機能の追加や変更，フォールト修正が比較的高い頻度で行われるため，分析対象として必ずしも適切ではないと考えた．そこで，リポジトリの最初のコミットから 1 年以上経過しており，その上である程度安定した状態にあると思われる（バージョン番号から推察される）バージョンのリリースを調査期間の始まりとした．

(2) 調査期間内に行われた各ソースファイルの変更に対し，前章で説明した手法によって変更量を算出する．

(3) 各変更に対して，そこから 1 ヶ月以内にそのファイルでフォールト修正が行われたかどうかを調べる．
フォールト修正が行われたかどうかは，ソースファイルの変更ログ（コミットメッセージ）にフォールト修正に関連するキーワードが登場するかどうか [19] でもって判断する．

以上の手順でデータ収集を行い，ソースコードの変更量と（1 ヶ月以内での）フォールト修正発生率との関係を分析する．

表 4 評価実験で用いたソフトウェア

名称	主要言語	備考
React	JavaScript	JavaScript UI 用ライブラリ
Electron	C++	アプリケーション構築用ライブラリ
netdata	C	リアルタイム監視システム
Redis	C	データベース

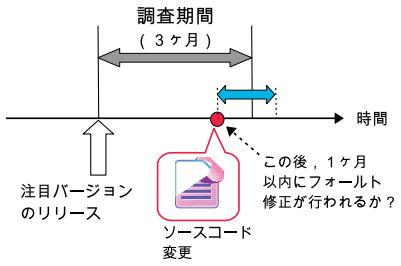


図 6 実験のイメージ図

4.3 結果

本実験では，各ソフトウェアに対して**表 5** に示すバージョンを注目バージョンとした．同表にはその時点でのソースファイル数とモデル構築に用いたトピック数も載せておく．

まずは，各ソフトウェアにおけるソースファイル変更事例を，提案法によるソースコード変更量の四分位数に従って **表 6** に示すかたちで四つに分類し，それぞれにおけるその後のフォールト修正発生率を比較した．その結果を**図 7** に示す．同図の横軸は 表 6 で示した分類 $C_1 \sim C_4$ であり，縦軸は各分類に入るソースコード変更のうち，その後の 1 ヶ月以内にフォールト修正が発生したものの割合を示している．各図では 2 本の折れ線グラフが描かれているが，それぞれコサイン距離を用いた場合とマンハッタン距離を用いた場合に対応している．

四つのソフトウェアいずれにおいても，フォールト修正発生率の推移に単調性は見られなかった．つまり，単純にベクトル間の距離（変更量）が大きいほどフォールト修正に結びつきやすいとか，逆に小さいほどフォールト修正に結びつきやすいという傾向は見られなかった．むしろ，中央値よりやや小さめに相当する分類 C_2 に入るようなソースコード変更が最もフォールト修正の起こりやすさと関係していた．なお，コサイン距離とマンハッタン距離のいずれを用いても結果に大きな違いは生じなかった．

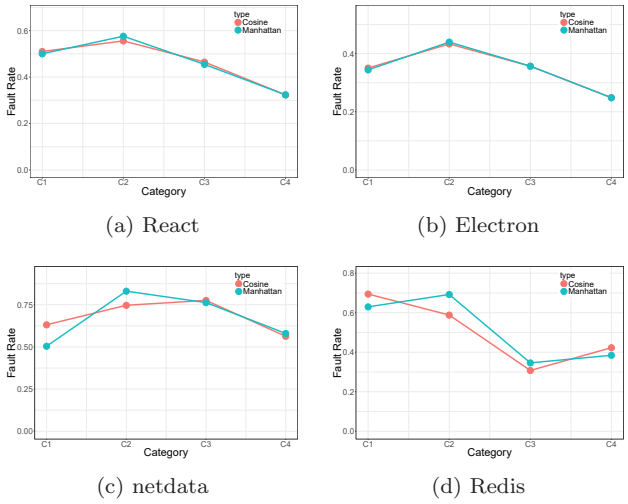
表 5 各ソフトウェアにおける注目バージョン（調査期間の始まり）とその時点でのソースファイル数，及びモデル構築に使用したトピック数

ソフトウェア	注目バージョン（タグ名）	ソースファイル数	トピック数
React	3.0.0	167	14
Electron	v1.0.0	110	14
netdata	v1.0.0	56	15
Redis	v15.0.0	170	15

表 6 ソースファイル変更事例の分類

分類	該当範囲
C_1	変更量 $\leq Q_1$
C_2	$Q_1 <$ 変更量 $\leq Q_2$
C_3	$Q_2 <$ 変更量 $\leq Q_3$
C_4	変更量 $> Q_3$

※ Q_i： 第 i 四分位数（$i = 1, 2, 3$）



(a) React (b) Electron

(c) netdata (d) Redis

図 7 分類ごとでのフォールト修正発生率の比較

4.4 考察

本実験では，トピックモデルを使ってソースファイルの特徴をベクトル化し，ソースコードの変更量を変更前後のソースファイルに対応したベクトル間の距離でもって評価することを試みた．四つのオープンソースソフトウェアにおけるソースコード変更事例についてデータ分析を行ったところ，単純な距離の大小（トピックの変化という観点でのソースコード変更量）という観点では，その後のフォールト修正の起こりやすさとの間には単調な関係性は見られなかった．むしろ，やや小さめの変更の場合が最もフォールト修正と関係しているという傾向が見られた．

まず，トピックベクトル間の距離が小さいということは，ソースファイルに登場する識別子の種類や個数にほとんど変化がないことを意味する．したがって，ソースコードの変更としては，特に文の追加や削除が行われるようなものではなく，一部分のみの変更に留まっていた可能性が高い．一方，ベクトル間の距離が大きいということは，新たな識別子が登場したり逆に既存のものが削除されたりした場合やそれらの登場回数が大きく変化したことを意味する．つまり，ソースコードの変更としては，限られたごく一部の修正というよりも，機能の追加や削除，変更が行われたものである可能性が高い．もちろんそういった修正が新たなフォールトを生み出さないわけではないが，今回の結果を見る限りでは，必ずしも “すぐに”（1 ヶ月以内に）フォールトの修正を必要とするような状況ではなかったと推察される．結果として，両者の中間にあってどちらかと言えば前者に近い分類 C_2 の変更，即ち，比較的限られた範囲の修正ではあるが識別子にはそれなりに変化も見られるようなソースコード変更が最もフォールト修正と関係していたのではないかと思われる．

次に，従来からよく知られている code churn （ソースコードの変更行数; ΔLOC）との関係についても見ておく．なお，ここでいう ΔLOC は変更前後のソースファイルに対して diff による差分を調べ，追加行数と削除行数を合計した単純なものとなっている[*3]．

表 7 提案法によるソースコード変更量と code churn（ΔLOC）の間の相関

ソフトウェア	コサイン距離	マンハッタン距離
React	0.371	0.395
Electron	0.324	0.334
netdata	0.444	0.556
Redis	0.469	0.526

各ソフトウェアにおける変更量をベクトルのコサイン距離及びマンハッタン距離で測定したものに対して，ΔLOC との相関係数（スピアマンの順位相関係数）を **表 7** に示す．結果として，両者の間には弱い正の相関が見られた．つまり，提案法であるトピックベクトルを使ったソースコード変更量評価は，従来から使われている code churn による評価と大きく矛盾することはないが，両者の間に強い相関があるわけでもなく，また違った見方をしていることを確認できた．

提案法の効果を code churn と比較するため，フォールト修正予測モデルを構築して，そこでの説明変数の重要度を算出してみる．モデルとしては，フォールト予測に関する研究において有望なモデルの一つである [20] ことからランダムフォレストを用いることにした．具体的には，フォールト修正の有無を目的変数とし，次の三つを説明変数とした予測モデルをランダムフォレスト法によって構築する：

(1) 提案法による変更量評価値（ベクトル間の距離）
(2) ΔLOC
(3) 当該ソースコード変更がフォールト修正を目的としたものであったかどうか

フォールト修正そのものが次のフォールト修正を引き起こしていることもある [21] ため，(3) も説明変数に追加した．そして，Breiman の手法 [22] に従って，各説明変数の重要度を算出した．結果を **表 8** 及び **表 9** に示す．各ソフトウェア内で最も高い重要度に下線を引いてある．

表 8 重要度の比較：(1) にコサイン距離を用いた場合

ソフトウェア	(1)	(2)	(3)
React	27.9	18.9	2.4
Electron	25.0	18.8	2.7
netdata	23.6	27.1	2.2
Redis	11.9	9.9	1.9

表 9 重要度の比較：(1) にマンハッタン距離を用いた場合

ソフトウェア	(1)	(2)	(3)
React	29.0	17.4	2.1
Electron	26.4	19.7	2.7
netdata	32.7	24.1	3.0
Redis	14.0	9.4	1.9

[*3] ここでは，1 行の内容変更は 1 行の削除と 1 行の追加と解釈され，ΔLOC = 2 となっている．

表 8 及び表 9 から分かるように，ほとんどの場合で，提案法による変更量（説明変数（1））が最も高い重要度を示していた．つまり，フォールト予測を行う上で有用な一つの変数となっていることを確認できた．

5. おわりに

本論文では，ソースファイルに対するトピック分析を通じて，ソースファイルの特徴をベクトル化し，ソースコード変更の大きさを変更前後のソースファイルに対応するベクトル間の距離で評価する手法を提案した．提案法では，ソースコードの意味的な側面の変更をとらえることを狙いとしており，コード変更によって識別子の種類数や登場回数が変化すると，それがベクトル間の距離となって現れるようになっている．

四つのオープンソースソフトウェアを対象とした評価実験の結果，ベクトルの変化量とフォールト修正の起こりやすさの間に単調性は見られなかったが，フォールト修正とのつながりがより強いと思われる値の範囲は存在していることが確認された．したがって，線形なモデルではフォールト予測は難しいと思われるが，適切に場合分けを行うことでフォールト予測に役立つことも考えられた．そこでランダムフォレストを用いた評価を行ったところ，提案法は（従来から広く知られている）“code churn による評価”や “コード変更そのものがフォールト修正目的であったかどうか” と比較しても有用な予測子となっていることが分かった．

今後，さらに多くのソフトウェアに対して評価実験を行い，ジャスト・イン・タイムフォールト予測 [23] といった分野への応用について検討したいと考えている．なお，今回は実験の都合上，トピックモデルの構築にあたってはトピック数をソフトウェアごとに一つに固定して実験を行ったが，トピック数を変化させた場合の影響についても分析を行う必要がある．これについては今後の課題としたい．

謝辞 本研究は JSPS 科研費（16K00099）の助成を受けたものです．

参考文献

[1] Jones, C.: *Applied Software Measurement: Global Analysis of Productivity and Quality*, McGraw-Hill, New York, 3rd edition (2008).
[2] LaToza, T. D. and Myers, B. A.: Developers Ask Reachability Questions, *Proc. 32nd ACM/IEEE Int'l Conf. Softw. Eng.*, Vol. 1, pp. 185–194 (2010).
[3] Vliet, H. V.: *Software Engineering: Principles and Practice*, Wiley Publishing, 3rd edition (2008).
[4] Pigoski, T. M.: *Practical Software Maintenance: Best Practices for Managing Your Software Investment*, John Wiley & Sons, Inc., NY (1996).
[5] Nagappan, N. and Ball, T.: Use of relative code churn measures to predict system defect density, *Proc. 27th Int'l Conf. Softw. Eng.*, pp. 284–292 (2005).
[6] Moser, R., Pedrycz, W. and Succi, G.: A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, *Proc. ACM/IEEE 30th Int'l Conf. Softw. Eng.*, pp. 181–190 (2008).
[7] 吉田敦，山本普一郎，阿草清滋：意味を考慮した差分抽出ツール，情報処理学会論文誌，Vol. 38, No. 6, pp. 1163–1171 (1997).
[8] 山本哲男：分散表現ベクトルを用いたソースコード検索及び分類の検討，電子情報通信学会技術報告，Vol. 116, No. 512, pp. 67–72 (2017).
[9] Le, Q. and Mikolov, T.: Distributed Representations of Sentences and Documents, *CoRR*, Vol. abs/1405.4053, pp. 1–9 (2014).
[10] Thomas, S. W., Hemmati, H., Hassan, A. E. and Blostein, D.: Static test case prioritization using topic models, *Empir. Softw. Eng.*, Vol. 19, No. 1, pp. 182–212 (2014).
[11] 岩田具治：トピックモデル，講談社，東京 (2015).
[12] 佐藤一誠：トピックモデルによる統計的潜在意味解析，コロナ社，東京 (2015).
[13] Tian, Y. and Lo, D.: A comparative study on the effectiveness of part-of-speech tagging techniques on bug reports, *Proc. IEEE 22nd Int'l Conf. Softw. Analysis, Evolution, and Reeng.*, pp. 570–574 (2015).
[14] Schmid, H.: Probabilistic Part-of-Speech Tagging Using Decision Trees, *Proc. Int'l Conf. New Methods in Language Processing*, pp. 44–49 (1994).
[15] Cao, J., Xia, T., Li, J., Zhang, Y. and Tang, S.: A density-based method for adaptive LDA model selection., *Neurocomputing*, Vol. 72, No. 7-9, pp. 1775–1781 (2009).
[16] Arun, R., Suresh, V., Madhavan, C. E. V. and Murty, M. N.: On Finding the Natural Number of Topics with Latent Dirichlet Allocation: Some Observations., (Zaki, M. J., Yu, J. X., Ravindran, B. and Pudi, V., eds.), Lecture Notes in Computer Science, Vol. 6118, Springer, pp. 391–402 (2010).
[17] Griffiths, T. L. and Steyvers, M.: Finding scientific topics, *Proc. National Academy of Sciences*, Vol. 101, No. Suppl. 1, pp. 5228–5235 (2004).
[18] Deveaud, R., SanJuan-Ibekwe, E. and Bellot, P.: Accurate and effective latent concept modeling for ad hoc information retrieval., *Revue des Sciences et Technologies de l'Information - Série Document Numérique*, Vol. 17, No. 1, pp. 61–84 (2014).
[19] Śliwerski, J., Zimmermann, T. and Zeller, A.: When Do Changes Induce Fixes?, *Proc. Int'l Workshop on Mining Softw. Repositories*, pp. 1–5 (2005).
[20] Lessmann, S., Baesens, B., Mues, C. and Pietsch, S.: Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings, *IEEE Trans. Softw. Eng.*, Vol. 34, No. 4, pp. 485–496 (2008).
[21] Rahman, F., Posnett, D., Hindle, A., Barr, E. and Devanbu, P.: BugCache for Inspections : Hit or Miss?, *Proc. of 19th ACM SIGSOFT Symposium and 13th European Conf. Foundations of Softw. Eng.*, pp. 322–331 (2011).
[22] Breiman, L.: Random Forests, *Machine Learning*, Vol. 45, No. 1, pp. 5–32 (2001).
[23] Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A. and Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance, *IEEE Trans. Softw. Eng.*, Vol. 39, No. 6, pp. 757–773 (2013).