Vol.2017-SE-196 No.27
2017/7/21

ソースコード変更履歴の字句パターン学習による自動コードレビュー手法

田端　啓一[†]　星野　隆[†]

† 日本電信電話株式会社
E-mail: †{tabata.keiichi,hoshino.takashi}@lab.ntt.co.jp

あらまし　本論文では，ソースコードの変更履歴を学習して，現在変更が必要な確度の高い部分をハイライトする手法を提案する．提案手法は，ソースコードを字句解析し数値化することで，ソースコードの構文上の特徴を N 次元のベクトルとして表現し，過去に変更されたパターンに類似するソースコード上の箇所を指摘するものである．提案手法を 3 件のオープンソースソフトウェアのリポジトリに対して適用・評価したところ，19 箇所の問題を発見した．
キーワード　コードレビュー, 自動化, 静的コード解析

An Automated Code Review Method using Lexical Pattern Learning on Source Code Change History

Keiichi TABATA[†] and Takashi HOSHINO[†]

† Nippon Telegraph and Telephone
E-mail: †{tabata.keiichi,hoshino.takashi}@lab.ntt.co.jp

Abstract　In this paper, we propose a method which highlights source code parts which are likely to be modified. The proposed method expresses source code snippets as N-dimention vector by tokenizing source codes into integer values, then pinpoints source code parts which are comformable to past changes on a source code repository. We applied the proposed method to three open source software repositories and found 19 problems.
Key words　Code Review, Automation, Static Code Analysis

1. はじめに

近年，ソフトウェアの開発スピードの向上を目的として，継続的インテグレーション [1] が盛んに行われている．継続的インテグレーションでは，ソフトウェアに変更がある都度，あるいは決められた時刻に，自動ビルドおよび自動テストが行われる．これと同時に，静的コード解析ツールによる自動コードレビュー [2] が行われることもある．

既存の自動コードレビュー技術には，ソースコードのメトリクスを活用するもの [3] や，コーディングスタイル違反を指摘するもの (注1) [4]，既知のバグのパターンを検出するもの (注2) [5]，セキュリティ上の指摘を行うもの [6] [7] など，様々な種類がある．しかしながら，過去に行われた修正を水平展開して指摘を行う技術は，著者らの知る限り実現していない．

そこで，本論文では，過去に行われた修正を水平展開して人間が指摘を行うためのコードレビューを，プログラムによる学習で自動化する手法を提案する．

(注1)：http://checkstyle.sourceforge.net
(注2)：http://findbugs.sourceforge.net/

具体的には，ソースコードの変更履歴に着目し，過去に修正された字句パターンに似たソースコードの箇所を，ニューラルネットワークで検出することで，現行のソースコードのうち，変更が必要である確度の高い部分をハイライトする．これにより，過去の修正を水平展開することが可能となる．

本論文の大まかな流れは以下の通りである．まず，2. 章で提案手法の解説を行う．次に、3. 章で評価，4. 章で考察を行う．次に，5. 章で他の研究と本研究の関連性について議論する．さらに，6. 章で今後の課題について議論する．最後に，7. 章で結論について述べる．

2. 提案手法

2.1 ソースコード変更箇所と差分の抽出

提案手法では，まず，ターゲットとなるソースコードリポジトリの最初のリビジョンから 1 リビジョンごとに，変更箇所のスニペットを取得する (図 1)．これを差分スニペットと呼ぶこととする．

ソースコードリポジトリが git [8] で管理されている場合，git diff コマンドで差分スニペットを取得可能である．

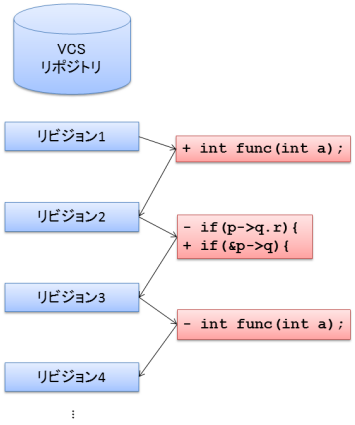

図 1　差分スニペット

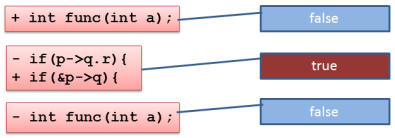

図 2　差分スニペットに対するタグ付与

git diff コマンドは，ソースコードに変更があったそれぞれの箇所で前後数行を含めた差分スニペットを出力する．差分スニペットでは，削除された行の先頭に-が付与され，追加された行の先頭には+が付与される．変更がない行の先頭には何も付与されない．

提案手法では，個々の差分スニペットに true か false の 1 ビットのタグ付けを行う (図 2)．true は，1 つの差分スニペット内に-の行 (削除された行) と+ の行 (追加された行) の両方が存在する場合に付与され，それ以外の場合 false が付与される．

タグが true であることは，ソースコードにおいて修正が生じたことを意味する．また，タグが false であることは，ソースコードにおいて行の削除のみあるいは行の追加のみが生じたことを意味する．

2.2　ソースコード差分の字句解析

取得された差分スニペットに対して，字句解析を行う (図 3)．このとき，-で始まる削除された行を字句解析の対象とするが，+で始まる追加された行は字句解析の対象としない．削除も追加もされていない行は字句解析の対象とする．つまり，変更前のソースコードに対して字句解析を行う．

字句解析の結果は，字句の種類に応じた数値の列となる．こ

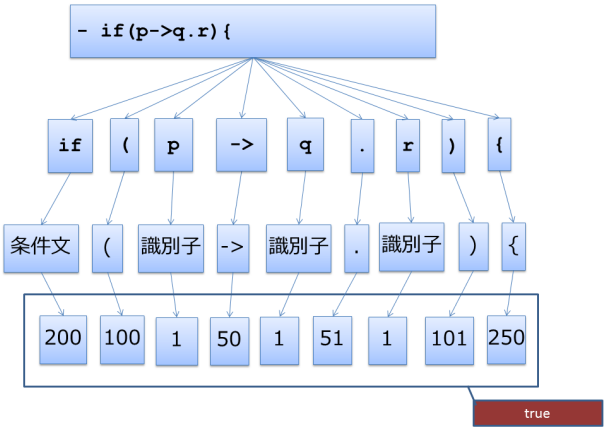

図 3　字 句 解 析

こで得られた数値の列から，中央の N 個を取り出して，N 次元ベクトルとする．これを差分ベクトルと呼ぶこととする．差分ベクトルには，元となった差分スニペットと同じ値の，true か false のタグが付与される．また，字句の数が N 個に満たない場合は，差分ベクトルの先頭および末尾にゼロを付加してパディングを行うこととする．

字句に応じた数値は，構文上の意味が遠いほど数値的に遠い値になるよう配慮されている．例えば，識別子トークンと if トークンでは，数値的に 199 の差が付けられる．これにより，抽象構文木 (Abstract Syntax Tree) を取り扱わずに、字句上の表現で、構文の特徴が得られる．

2.3　差分ベクトルによる学習

取得されたすべての差分ベクトルを，3 層のニューラルネットワークによって学習する (図 4)．このとき，ニューラルネットワークの入力段は差分ベクトルの次元数と同じユニット数であり，差分ベクトルを与えて，バックプロパゲーションによって学習を行う．また，ニューラルネットワークの出力段は 2 ユニットであり，差分ベクトルにつけられたタグの true と false が対応する．

この手順により，過去に修正された字句のパターンが 1 次元の画像としてニューラルネットワークによって学習される．

2.4　最新のソースコードに対する認識

最新のソースコードの先頭から順に，数行をスライド窓式に取得してコードスニペットとし，これを字句解析する．得られた数値の列から中央の N 個を取り出し，差分ベクトルと同じ次元数に調節したものを，認識ベクトルと呼ぶこととする．

学習で得られたニューラルネットワークの入力段に認識ベクトルをセットして，プロパゲーションを行い認識を実行する．結果は true か false となる．true であれば，過去に修正があった箇所と類似していることを表す．true となった行に対してハイライトを行う (図 5)．行頭の x 印がハイライトである．図 5 では，代入文がハイライトされている．

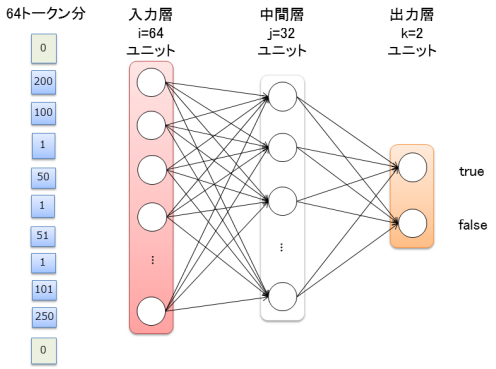

図 4　差分ベクトルによる学習

```
static int get_word_width(const char *m)
{
        int w;

x       w = 0;
        while (isalnum(*m))
                w += get_glyph_width(*m++);

        return w;
}
```

図 5　ハイライトされたソースコード

表 1　オープンソースソフトウェア A のリポジトリの概要

言語	C
アプリケーションの種類	ゲーム
行数	15714
コミット数	102

3. 評　　価

提案手法をオープンソースソフトウェア A [注3] のリポジトリに適用し，ハイライトされたソースコードの中に，実際に変更が必要な箇所があるかを，人間が探した．オープンソースソフトウェアのリポジトリの概要を表 1 に示す．

まず，直近 100 件のコミットを元に学習を行った結果，15714 行中の 2249 行 (14.3%) がハイライトされてしまい，どの行に問題があるかを判別することができなかった．

そこで，直近 15 件のコミットに限定して学習を行った結果，15714 行中の 222 行 (1.4%) がハイライトされ，問題のある行を判別することが可能になった．

ハイライトされた 222 行を人間が精査したところ，「行頭禁則文字の種類が不足している」(1 箇所)，「static 指定子を付け忘れている」(2 箇所)，「不要なキャストを行っている」(11 箇所) という，3 種類で 14 箇所の問題がみつかった (表 2)．これらの問題は，直近の修正箇所に字句の上で類似しているため，提案手法のアルゴリズムを用いて検知できたものと考えられる．

(注3)：https://github.com/ktabata/suika2

表 2　提案手法の適用結果 (リポジトリ A)

ハイライトされた行数	222
みつかった問題の種類	3
みつかった問題の箇所	14

表 3　オープンソースソフトウェア B のリポジトリの概要

言語	C
アプリケーションの種類	C コンパイラ
行数	81727
コミット数	7110

表 4　提案手法の適用結果 (リポジトリ B)

ハイライトされた行数	282
みつかった問題の種類	1
みつかった問題の箇所	4

表 5　オープンソースソフトウェア C のリポジトリの概要

言語	C
アプリケーションの種類	エディタ
行数	17212
コミット数	271

さらに，提案手法をオープンソースソフトウェア B [注4] のリポジトリに適用し，同様に，ハイライトされたソースコードの中に，実際に変更が必要な箇所があるかを，人間が探した．オープンソースソフトウェアのリポジトリの概要を表 3 に示す．

まず，直近 100 件のコミットを元に学習を行った結果，81727 行中の 1805 行 (2.2%) がハイライトされてしまい，どの行に問題があるかを判別することができなかった．

そこで，直近 15 件のコミットに限定して学習を行った結果，81727 行中の 282 行 (0.35%) がハイライトされ，問題のある行を判別することが可能になった．

ハイライトされた行を人間が精査したところ，「break 文が不足している」(4 箇所) という問題がみつかった (表 4)．これらの問題は，直近の修正箇所に字句の上で類似しているほか，switch 文〜case 文のトークン列が特徴的であるため，提案手法のアルゴリズムを用いて検知できたものと考えられる．

最後に，提案手法をオープンソースソフトウェア C [注5] のリポジトリに適用し，同様に，ハイライトされたソースコードの中に，実際に変更が必要な箇所があるかを，人間が探した．オープンソースソフトウェアのリポジトリの概要を表 5 に示す．

まず，直近 100 件のコミットを元に学習を行った結果，17212 行中の 1418 行 (8.2%) がハイライトされてしまい，どの行に問題があるかを判別することができなかった．

そこで，直近 11 件のコミットに限定して学習を行った結果，17212 行中の 37 行 (0.21%) がハイライトされ，問題のある行を判別することが可能になった．

(注4)：https://github.com/bobrippling/ucc-c-compiler
(注5)：https://github.com/adsr/mle

表 6　提案手法の適用結果 (リポジトリ C)

ハイライトされた行数	37
みつかった問題の種類	1
みつかった問題の箇所	1

ハイライトされた行を人間が精査した結果,「ファイルオープン時に Segmentation fault が発生する行」をピンポイントで指摘していることが判明した．この箇所は直近の 11 件のコミット内で変更があった箇所であった．(表 6).

4. 考　　察

4.1 使用するコミット数

直近のコミットを元に学習を行ったことは，最近修正された箇所ほど新たに修正されやすいという傾向 (Fix-Cache) に関する研究 [9] に基いている．

ただし，使用した 2 つのリポジトリ (オープンソースソフトウェア A,B) においては，直近 15 件未満のコミットを用いて学習を行った場合，教師データの不足となり，意味のあるハイライトを得られなかった．

また，もう 1 つのリポジトリ (オープンソースソフトウェア C) においては，直近 11 件未満のコミットを用いて学習を行った場合，教師データの不足となり，意味のあるハイライトを得られなかった。

このため，提案手法の評価では，評価に用いるちょうどよいコミット数を ad hoc に決定した．今後，評価に用いるコミット数を定量的に決定するアルゴリズムを考案する必要がある．

4.2 大きな差分スニペット

使用した 2 つのリポジトリ (オープンソースソフトウェア A,B) においては，直近 16 件目のコミットで行数の大きい差分スニペットが生じており，これを学習データに利用すると，多数の行にハイライトが行われてしまった．

もう 1 つのリポジトリ (オープンソースソフトウェア C) においては，直近 12 件目のコミットで行数の大きい差分スニペットが生じており，これを学習データに利用すると，多数の行にハイライトが行われてしまった．

このため，トークン数の多い差分スニペットは，学習に用いないなどの対策が必要であることが判明した．

5. 関 連 研 究

5.1 Fault-prone モジュール予測

ソフトウェアのある部分が欠陥を含んでいそうであるということを Fault-prone と呼び，欠陥を含んでいそうなモジュールを予測する技術を Fault-prone モジュール予測と呼ぶ. [10]Fault-prone モジュール予測について様々な研究が行われており，その予測単位は，ファイル，クラス，メソッドといったプログラム構造上の粒度である．

Fault-prone モジュール予測の技術は，その対象スコープによって，次の 2 つに分けられる．1 つは，同一のソフトウェアをスコープとし，過去のバージョンを元にして将来のバージョンでの欠陥を予測する技術である．もう 1 つは，別なソフトウェアをスコープとし，あるソフトウェアでの欠陥の傾向を元に，他のソフトウェアでの欠陥を予測する技術である．

提案手法は，モジュールの粒度ではないが，行を単位とした欠陥予測を行うものであり，同一ソフトウェアをスコープとした欠陥予測技術の一つであると言える．

5.2 Fix-Cache

ソフトウェアに含まれる欠陥は偏在することが知られており，その修正にも偏在が生じることを用いた欠陥予測に，Fix-Cache [9] がある．

Fix-Cache の研究によれば，ソフトウェアの修正箇所には時間的な局所性と，空間的な局所性がある．つまり，最近修正された箇所ほど近い将来に修正されやすく，また，修正が集中している箇所ほど修正されやすい．

本論文における提案手法の評価では，直近の修正に限ってソースコードリポジトリの解析を行っている．これは Fix-Cache における欠陥の時間的な局所性を利用していると解釈できる．

5.3 Mining Software Repositories

近年，ソフトウェア開発において，バージョン管理システム (VCS) やバグトラッキングシステム (BTS) といった，既存のリポジトリから有用な情報を抽出する技術が注目されており，Mining Software Repositories(MSR) と呼ばれる分野として，盛んに研究されている. [11]

MSR のマイニング対象の中でもソースコードリポジトリは，ドキュメントが最新でないか存在しなくても利用できるため，研究対象として活用しやすい．ソースコードリポジトリの履歴を元に欠陥の予測が可能であることも示されている. [11]

提案手法は，ソースコードリポジトリを対象として変更履歴を用いた解析を行っており，MSR の一手法であると言える．

6. 今後の課題

提案手法では，実装の単純さを優先的に考え，3 層ニューラルネットワークを用いている．しかしながら，この 3 層ニューラルネットワークは，他の 2 値分類器で置き換え可能である可能性がある．特に，ニューラルネットワークの階層を深くすることで，さらなるソースコード上の特徴抽出が可能になる可能性がある．

提案手法の評価では，ハイライトされた行数に対して，みつかった問題が少なく (1.4%-6.3%)，false-positive であるハイライトが少なくないため，よりピンポイントに，効率よく，ソースコードの箇所を指摘できるような改善の余地が残されている．

提案手法の評価では，評価に用いる git のコミット数を ad hoc に決定したが，これを定量的に決定するアルゴリズムを考案する必要がある．

提案手法では，過去の変更を元に現在変更されそうな確度の

高い行を指摘しているが，過去の修正を元にソースコードを自動修正することも実現可能かもしれない．

7. ま と め

本論文では，ソースコードの中から，変更が必要な確度の高い部分をハイライトする手法を提案し，3つのオープンソースソフトウェアのリポジトリに適用した．その結果，0.21%-1.4%の行がハイライトされた．ハイライトされた行を人間が精査した結果，過去に修正された問題と似ていると考えられる19箇所の問題がみつかった．これにより，人間がコードレビューを行ったときの，過去に対処した問題を水平展開することと同様の自動コードレビューを，提案手法によって実現したと言える．

文　献

[1] D. Stahl and J. Bosch, “Industry application of continuous integration modeling: A multiple-case study,” 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), pp.270–279, May 2016.
[2] C. Bolduc, “Lessons learned: Using a static analysis tool within a continuous integration system,” 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp.37–40, Oct. 2016.
[3] A. Vogelsang, A. Fehnker, R. Huuck, and W. Reif, “Software metrics in static program analysis,” pp.485–500, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
[4] N. Funabiki, T. Ogawa, N. Ishihara, M. Kuribayashi, and W.C. Kao, “A proposal of coding rule learning function in java programming learning assistant system,” 2016 10th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS), pp.561–566, July 2016.
[5] M.N. Al-Ameen, M.M. Hasan, and A. Hamid, “Making findbugs more powerful,” 2011 IEEE 2nd International Conference on Software Engineering and Service Science, pp.705–708, July 2011.
[6] G. McGraw, G. McGraw, G. McGraw, and G. McGraw, “Automated code review tools for security,” Computer, vol.41, no.12, pp.108–111, Dec. 2008.
[7] M. Kulenovic and D. Donko, “A survey of static code analysis methods for security vulnerabilities detection,” 2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp.1381–1386, May 2014.
[8] S. Just, K. Herzig, J. Czerwonka, and B. Murphy, “Switching to git: The good, the bad, and the ugly,” 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), pp.400–411, Oct. 2016.
[9] E. Engstrom, P. Runeson, and G. Wikstrand, “An empirical evaluation of regression testing based on fix-cache recommendations,” 2010 Third International Conference on Software Testing, Verification and Validation, pp.75–78, April 2010.
[10] O. Mizuno and Y. Hirata, “A cross-project evaluation of text-based fault-prone module prediction,” 2014 6th International Workshop on Empirical Software Engineering in Practice, pp.43–48, Nov. 2014.
[11] H. Hata, O. Mizuno, and T. Kikuno, “Bug prediction based on fine-grained module histories,” Proceedings of the 34th International Conference on Software Engineering, pp.200–210, ICSE ’12, IEEE Press, Piscataway, NJ, USA, 2012. http://dl.acm.org/citation.cfm?id=2337223.2337247