情報検索に基づくBug Localizationへの不吉な臭いの深刻度の利用

高橋 碧[1,a)]　セーリム ナッタウット[1,b)]　林 晋平[1,c)]　佐伯 元司[1,d)]

概要：大規模なソフトウェア開発では，ある特定のバグを解決するために修正すべきソースコード箇所を見つける Bug Localization が必要である．情報検索に基づく Bug Localization 手法（IR 手法）は，バグに関して記述されたバグレポートとソースコード内のモジュールとのテキスト類似度を計算し，これに基づき修正すべきモジュールを特定する．しかし，この手法は各モジュールのバグの出やすさの度合いを考慮していないため精度が低い．本論文では，ソースコード内のモジュールのバグの出やすさとして不吉な臭いを用い，これを既存の IR 手法と組み合わせた Bug Localization 手法を提案する．提案手法では，不吉な臭いの深刻度と，ベクトル空間モデルに基づくテキスト類似度を統合した新しい評価値を定義している．これは深刻度の高い不吉な臭いとバグレポートとの高いテキスト類似性の両方を持つモジュールを上位に位置付け，バグを解決するために修正すべきモジュールを予測する．4 つの OSS プロジェクトの過去のバグレポートを用いた評価では，いずれのプロジェクト，モジュール粒度においても提案手法の精度が既存の IR 手法を上回り，また最大で 269%の向上がみられた．

Using Code Smell Severity to Improve IR-Based Bug Localization

Aoi Takahashi[1,a)]　Natthawute Sae-Lim[1,b)]　Shinpei Hayashi[1,c)]　Motoshi Saeki[1,d)]

1. はじめに

大規模なソフトウェア開発では，開発者は多くのバグを修正しなければならない．バグ報告に基づくバグレポートが多数作成され，それらの解決には多くの時間を要する．また，バグを修正するために最も時間がかかる作業の 1 つは，バグレポートに書かれたバグの原因となるソースコード箇所の特定である．Thung ら [1] は 374 個のバグを分析し，バグの約 90%が 1–2 個のソースコードファイルに存在することを発見した．したがって，莫大なソースコードの中から，ごく小さな修正すべきソースコード箇所を特定しなければならない．

上記の課題に対処する手段の 1 つに，Bug Localization がある．Bug Localization では，バグを解決するために修正すべきソースコード箇所を特定することを指し，情報検索（Information Retrieval; IR）手法 [2–4]，静的解析手法 [5]，動的解析手法 [6,7] などがある．また，これらを統合した手法 [8–10] も提案されている．

IR 手法では，報告されたバグレポートとソースコードとのテキスト類似度を計算することで，バグの修正箇所を特定する．このように，対象とするバグレポートとソースコードのみを入力として用いる，利用のための制約が少ない手法である一方，得られる精度に課題がある．その理由の一つに，IR 手法が主に入力のテキスト類似度に注目しており，結果として特定されるモジュールにどの程度バグが出やすいのかを考慮していないことがある．**表 1** は，オープンソースソフトウェア（OSS）ArgoUML において，バグレポート “Exception when saving to a readonly directory.” を入力として IR 手法を適用した際に得られるモジュール一覧の抜粋である．上位のクラスは入力のバグレポートとテキスト類似性が高く，例えばクラス ActionSaveProject

[1] 東京工業大学
Tokyo Institute of Technology
[a)] takahashi-a-at@se.cs.titech.ac.jp
[b)] natthawute@se.cs.titech.ac.jp
[c)] hayashi@c.titech.ac.jp
[d)] saeki@c.titech.ac.jp

表 1: IR 手法の問題例
Table 1 Application example of IR-based bug localization.

順位	クラス名	テキスト類似度
1	ActionSaveProject	0.179
2	AbstractFilePersister	0.174
3	ZipFilePersister	0.150
4	XmiFilePersister	0.150
5	UmlFilePersister	0.142
6	ProjectFilePersister	0.128
7	FileConstants	0.123
8	ActionSaveProjectAs	0.122
9	ActionOpenProject	0.121
10	ProjectBrowser	0.121

は save を含む多くの単語を共有している．しかし，実際の正解にこのクラスは含まれず，10 位の ProjectBrowser が正解に該当する．ここで注目すべき点は，このクラスは他のクラスよりもテキスト類似性に乏しいが，複数の不吉な臭い [11] を持っていることである．不吉な臭いとは，低品質な設計によりリファクタリングを必要とするソースコード箇所であり，不吉な臭いを持つモジュールにはバグが出やすいことが示されている [12]．ProjectBrowser クラスからは，行数の長さに起因する Blob Class と，責務の過多による複雑性に起因する God Class の 2 つの臭いが検出された．こういった不吉な臭いの特徴もテキスト類似度に加えて考慮することで，このクラスを高順位に引き上げられる可能性がある．

本論文では，IR 手法に加えて不吉な臭いの情報も組み合わせることで IR 手法の精度を向上させる手法を提案する．以下に本論文の貢献を述べる．

- 本論文では IR 手法とソースコードに存在する不吉な臭いの情報を組み合わせる Bug Localization 手法を提案した．本手法は，ソースコード以外の入力なしに，IR 手法の精度を向上できる．
- 4 つの OSS に関して評価実験を行い，IR 手法に比べて本手法では全てのプロジェクトで精度が向上したことを示した．

本論文の以降の構成を以下に示す．2 章では，Bug Localization と不吉な臭いを組み合わせた手法について説明する．3 章では，提案手法の有効性を検証するための評価実験の内容とその結果の考察を行う．4 章では，Bug Localization と不吉な臭いに関する既存研究を紹介する．5 章では，本論文のまとめと今後の課題について記す．

2. 提案手法

2.1 概要

提案手法の概要を図 **1** に示す．バグレポートとソースコードを入力として IR 手法を適用し，ソースコード内のモジュールごとにバグレポートとのテキスト類似度を計算する．また，ソースコードを入力として不吉な臭いを検出し，不吉な臭いの深刻度を得る．不吉な臭いの深刻度とは，不吉な臭いの深刻さ・悪さを数値化したもので，不吉な臭いのより詳細な優先順位付けを可能にする．テキスト類似度と不吉な臭いの深刻度の両方を組み合わせることで提案メトリクスが計算される．各モジュールを提案メトリクスの高い順に並び替えたリストを作成し，そのリストを参考に開発者はバグの修正を試みる．

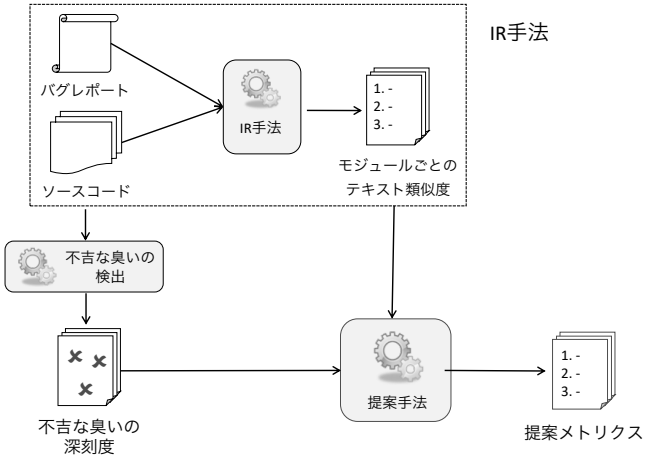


図 1: 提案手法の概要
Fig. 1 Overview of our technique.

2.2 不吉な臭いの検出

対象プロジェクトのソースコードを入力として，不吉な臭いを検出する．検出された不吉な臭いは，その種類，対象モジュール，粒度（クラス，メソッドなど），およびその深刻度（Severity）[16] の情報を伴っている．提案手法では不吉な臭いの情報の定量化方法としてこの深刻度を用いる．これは，単に不吉な臭いを持つか否かよりも細かく優先順位付けを行える分解能を与えるためである．

対象とするモジュール群 M のうち，特定のモジュール m で検出された全ての不吉な臭いの深刻度の合計 $Sev(m)$ を計算する．ただし，Bug Localization 対象のモジュール粒度と同じ粒度の不吉な臭いのみを用いる．この $Sev(m)$ を，以下のように線形正規化して用いる．

$$nSev(m) = Sev(m) / \max_{m' \in M} Sev(m')$$

2.3 IR 手法

IR 手法では，対象プロジェクトのバグの内容が記述されたバグレポートとソースコードを入力とする．これらの入力から，ソースコードのモジュールごとにバグレポートとのテキスト類似度が計算される．対象とするモジュール群 M のうち，特定のモジュール m とバグレポートとのテキスト類似度を $Prob(m)$ として得る．これを以下のように線形正規化して用いる．

$$nProb(m) = Prob(m) / \max_{m' \in M} Prob(m')$$

表 2: ArgoUML に対する適用例
Table 2 Application example to ArgoUML.

(a) 既存の IR 手法（$\alpha = 0$）

順位	クラス名	*nProb*	*nSev*	*BuggyProb*
1	ActionSaveProject	1.000	0	1.000
2	AbstractFilePersister	0.971	0	0.971
3	ZipFilePersister	0.842	0	0.842
4	XmiFilePersister	0.841	0	0.841
5	UmlFilePersister	0.795	0	0.795
6	ProjectFilePersister	0.715	0	0.715
7	FileConstants	0.691	0	0.691
8	ActionSaveProjectAs	0.683	0	0.683
9	ActionOpenProject	0.679	0	0.679
10	ProjectBrowser	0.679	0.667	0.679

(b) 提案手法（$\alpha = 0.42$）

順位	クラス名	*nProb*	*nSev*	*BuggyProb*
1	ProjectBrowser	0.679	0.667	0.674
2	ActionSaveProject	1.000	0	0.580
3	AbstractFilePersister	0.971	0	0.563
4	ZipFilePersister	0.843	0	0.489
5	XmiFilePersister	0.842	0	0.488
6	Modeller	0.222	0.833	0.479
7	UmlFilePersister	0.794	0	0.461
8	Import	0.430	0.500	0.460
9	ParserDisplay	0.051	1	0.449
10	StylePanel	0.586	0.250	0.445

表 3: プロジェクトの情報
Table 3 Projects used.

プロジェクト	バージョン	バグレポート数	クラス数	クラスの臭い数	メソッド数	メソッドの臭い数
ArgoUML	0.20–0.24	74	1476	61	12131	411
JabRef	2.0–2.6	36	374	37	2947	60
jEdit	4.2–4.3	86	406	51	5276	185
muCommander	0.8.0–0.8.5	81	529	41	3916	44

2.4 提案メトリクスの計算

モジュールごとに，*nSev* と *nProb* を用いて以下のようにメトリクスを計算する．

$$BuggyProb(m) = (1 - \alpha) \times nProb(m) + \alpha \times nSev(m)$$

ここで α $(0 \leq \alpha \leq 1)$ は用いている *nSev* と *nProb* の重要度の分配を表すパラメータであり，利用者が経験的に与えたり，適用したいプロジェクトの過去のデータや他のプロジェクトの値などから定める．$\alpha = 0$ のとき，*BuggyProb* が表すモジュール順位は IR 手法の結果と等しくなる．一方，$\alpha = 1$ のとき，順位には不吉な臭いの情報のみが反映される．モジュールを提案メトリクスの高い順に並び替えたリストを作成し，そのリストを参考に開発者はバグの修正を試みる．

2.5 適用例

不吉な臭いの検出に inFusion[*1]，VSM に基づくテキスト類似度の計算に TraceLab [17] を用いて，提案手法を適用した例を示す．**表 2** は，オープンソースソフトウェア ArgoUML のあるバグレポートに対するクラス粒度での適用結果の上位 10 を示しており，IR 手法（$\alpha = 0$）と提案手法 $(\alpha = 0.42)$ の結果を比較している．バグレポートに対して実際に変更された正解クラスは灰色で強調されている．IR 手法では正解のクラス ProjectBrowser が 10 位に位置付けられているが，このクラスは不吉な臭いを持っており，*nSev* の値が高かったため，提案手法を用いた場合，1 番目に位置付けられるようになった．この例では，不吉な臭いの深刻度を用いることで，実際に変更されたクラスを IR 手法より高順位に位置付けることができている．

[*1] `http://www.intooitus.com/products/infusion`
（販売は中止されており，現在はアクセスできない．）

3. 評価実験

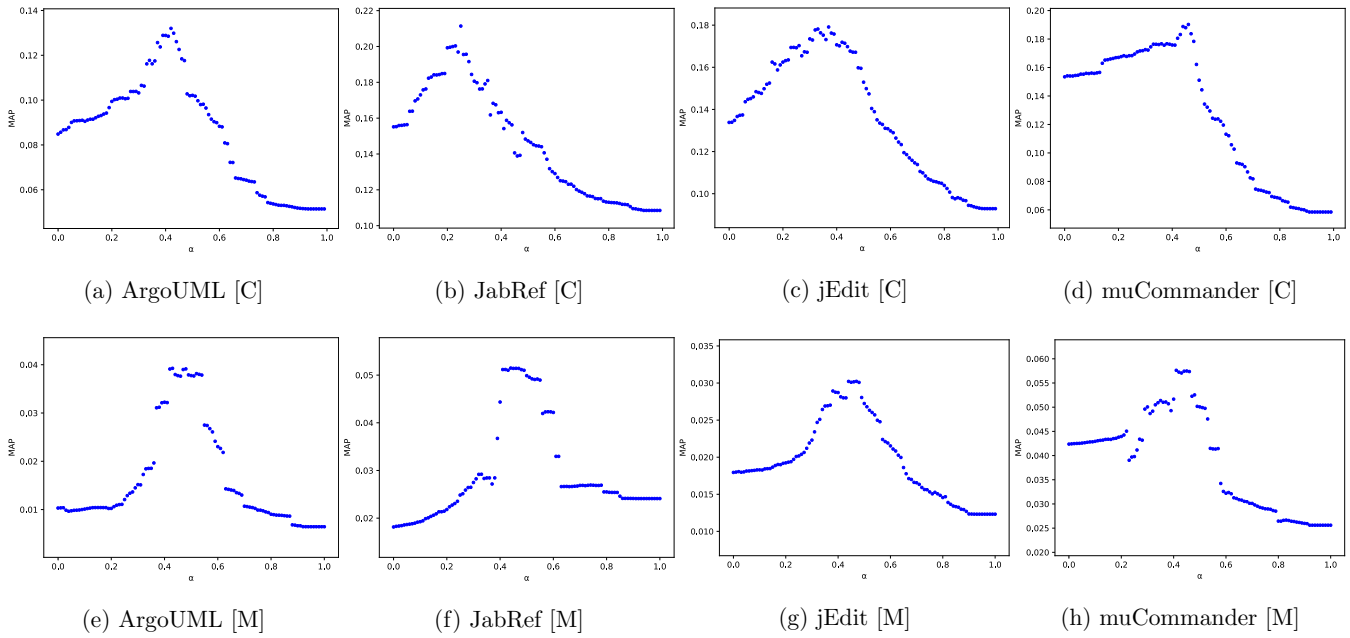
提案手法の評価のため，以下の 2 つの Research Question（RQ）を定めた．

RQ_1：**重要度のパラメータ α は手法の精度にどのように影響するか?** α は提案手法において不吉な臭いの情報と既存の IR 手法の情報の結合割合であり，この値の影響を調べることは 2 つの情報のそれぞれの重要度を求めることに相応する．また，実際に手法を利用する際に α を定めるためにも，最適な α を求めておくことは重要である．

RQ_2：**既存の IR 手法と比べ，提案手法は優れているか?** 実際に提案手法を用いることで既存の IR 手法よりも優れているかを調べることにより，不吉な臭いを既存の IR 手法に組み合わせることが効果的であり，精度を向上させるかを検証する．α を定める際には，RQ_1 の結果を利用する．

3.1 データ収集

本実験では，Dit らの変更影響分析のデータセット [18] を用いた．このデータセットには，4 つのオープンソースソフトウェアプロジェクトに対して，特定バージョンのソースコード，次期バージョンまでに解決されたイシュー，イシュー解決時に変更されたモジュール（正解セット）が含まれている．本実験では，バグカテゴリに属するイシュー



(a) ArgoUML [C] (b) JabRef [C] (c) jEdit [C] (d) muCommander [C]

(e) ArgoUML [M] (f) JabRef [M] (g) jEdit [M] (h) muCommander [M]

図 2: α ごとの MAP の値．[C]: クラスレベル，[M]: メソッドレベル．
Fig. 2 MAP values for each α. [C]: class level, [M]: method level.

をバグレポートとして使用した．各プロジェクトの情報を**表 3** に示す．不吉な臭いは，データセットに含まれる変更前バージョンのソースコードを inFusion に入力として与えて得た．またこのソースコードは，バグレポートとのテキスト類似度の計算にも用いられた．

3.2 評価メトリクス

提案手法では，提案メトリクス *BuggyProb* に基づいて各モジュールを順位付けしたリストが得られる．これと，データセットから得られる正解セットとを比較し，順位付けリストの精度の評価を行う．これには，順序付きランキングを評価するために用いられる精度メトリクスである Mean Average Precision（MAP）を用いた．MAP は全てのバグレポートの Average Precision（AP）の平均である．1 つのバグレポートの AP は以下のように計算される．

$$AP = \sum_{i=1}^{N} \frac{Pre(i) \times pos(i)}{\text{正解セットのモジュール数}}$$

ここで，i は順序付けされたモジュールの順位，N は順序付けしたモジュールの総数を表す．$Pre(i)$ は i を含む i よりも順位の高いモジュールにおける正解セットのモジュールの割合を表す．$pos(i)$ は i 番目にランク付けされたモジュールが正解セットに含まれていれば 1，さもなくば 0 を返す．MAP は，与えられた全てのバグレポートでの AP の平均値となる．

表 4: α の最適値
Table 4 Most appropriate α values.

プロジェクト	クラスレベル	メソッドレベル
ArgoUML	0.42	0.43
JabRef	0.25	0.44
jEdit	0.37	0.47
muCommander	0.46	0.41

3.3 RQ_1：重要度のパラメータ α は手法の精度にどのように影響するか?

3.3.1 調査方法

RQ_1 に答えるために，α を横軸，MAP を縦軸とするグラフを描く．α を 0.01 刻みで 0 から 1 まで変化させ，それぞれの *BuggyProb* を計算し各モジュールを順位付けた場合の MAP を計算する．このグラフをモジュール粒度がクラスレベルとメソッドレベルの両方で対象プロジェクトに関して作成する．これらにより得られたグラフの形状および MAP が最大値となる α の値について議論する．

3.3.2 結果と考察

結果を**図 2** に示す．また，それぞれのグラフの α の最適値，すなわち MAP が最大となる α の値を**表 4** に示す．最も重要な結果として，既存の IR 手法のみを用いたり（$\alpha = 0$），不吉な臭いのみを用いたりする（$\alpha = 1$）のではなく，それら 2 つを組み合わせた場合の方が効果的だったことがある．クラスレベルの結果では，α が 0.25–0.46 の値で MAP が最大となった．メソッドレベルの結果では，α が 0.41–0.47 の値で MAP が最大となった．メソッドレベルの結果では，クラスレベルの結果に比べて α の最適値が広範囲にわたっていない．一方で，クラスレベルでは，

表 5: RQ_2 で用いた α の値
Table 5 α values used in answering RQ_2.

プロジェクト	クラスレベル	メソッドレベル
ArgoUML	0.36	0.44
JabRef	0.42	0.44
jEdit	0.38	0.43
muCommander	0.35	0.45

JabRef のみ他のプロジェクトに比べて α の値が小さい．その理由として，JabRef ではクラスレベルの不吉な臭いの検出数が他のプロジェクトに比べて少なく，不吉な臭いの影響が大きくならなかった可能性がある．しかし，クラスに対する不吉な臭いの検出量の割合は ArgoUML が最も少ないため，原因の特定にはより詳細な分析が必要である．JabRef のクラスレベルでの結果を除き，α の最適値は 0.40 付近に集中しており，この値は経験的に他のプロジェクトにも利用できる可能性があると考える．

いずれの試行においても，テキスト類似度，不吉な臭いの両情報を適切に組み合わせた際（クラスレベルでは $\alpha \in [0.25, 0.46]$，メソッドレベルでは $\alpha \in [0.41, 0.47]$）に MAP が最大となった．

3.4 RQ_2：既存の IR 手法と比べ，提案手法は優れているか?

3.4.1 調査方法

RQ_1 の結果を用いて α を実際に定め，提案手法を適用する．ここでは，あるプロジェクトを予測する際には，他の 3 プロジェクトの α の最適値の平均を用いた．例えば，対象プロジェクトが ArgoUML の場合には，JabRef，jEdit，muCommander の 3 つのプロジェクトの RQ_1 で求めた α の最適値の平均を α として定めて提案手法を適用した．これは，複数の既存プロジェクトの α の最適値が既知である状況で対象プロジェクトの α を定める場合を想定している．実際に用いた α の値を**表 5** に示す．こうして MAP を求め，提案手法が既存の IR 手法（$\alpha = 0$）に比べて優れているかを検証する．

3.4.2 結果と考察

結果を**図 3**，**図 4** に示す．図上部には IR 手法に対する提案手法の MAP の上昇率を示している．いずれのプロジェクト，モジュール粒度でも，既存手法に比べて提案手法の MAP が向上している．ただし，全体としてクラスレベルの方が，上昇率が低くなっている．この原因のひとつに，クラスレベルの JabRef での α の最適値が他のプロジェクトと傾向が異なり，これが本実験で用いる α の値に悪影響を及ぼしていることがある（表 4）．一方で，メソッドレベルではプロジェクト毎の α の最適値が近いため，MAP の上昇率が高くなったと考える．

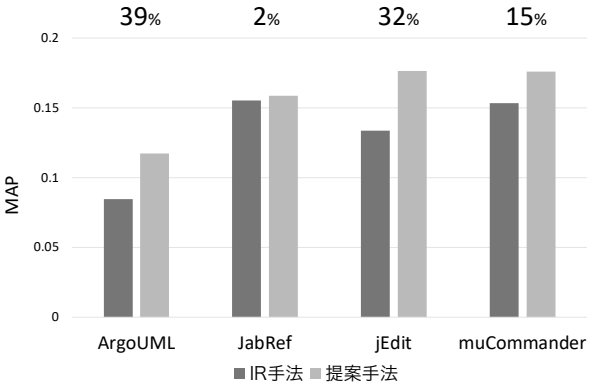


図 3: MAP の比較 (クラスレベル)
Fig. 3 Comparison of MAP values (class level).

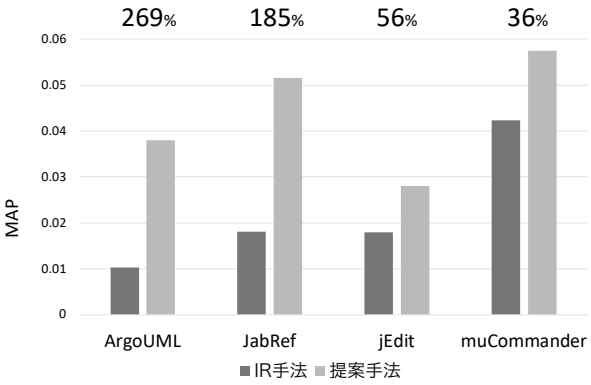


図 4: MAP の比較 (メソッドレベル)
Fig. 4 Comparison of MAP values (method level).

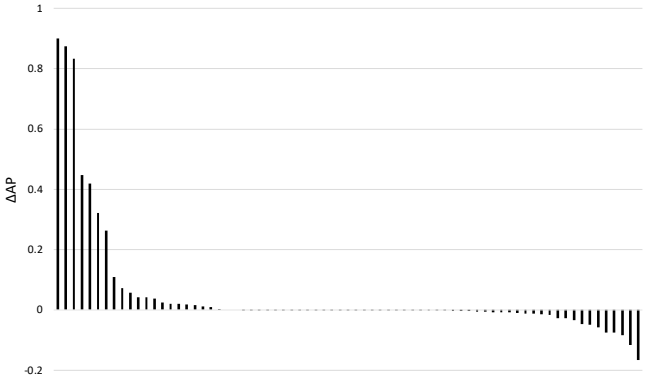


図 5: ArgoUML でのバグレポートごとの AP の変化
Fig. 5 Delta of AP for each bug report of ArgoUML.

すべての試行において，提案手法の MAP が既存の IR 手法を上回った（最大では 269%の向上）．

3.5 議論

提案手法による MAP の向上がどのような要因によって起こったのかを詳細に調べるために，バグレポートごとの AP を調査した．**図 5** は，ArgoUML においてクラスレベルの Bug Localization を行った際に，IR 手法（$\alpha = 0$）に比べて提案手法（$\alpha = 0.42$）がどの程度 AP を上昇させたかを示したものである．横軸はそれぞれのバグレポート，

縦軸は AP の上昇量（$\Delta AP = AP_{\alpha=0.42} - AP_{\alpha=0}$）を表しており，$\Delta AP$ が高い順に左からバグレポートを並べている．グラフから，それぞれのバグレポートがバランスよく AP を向上させたわけではなく，偏りがあることがわかる．また，中央付近には変化がほとんど見られないバグレポートがあり，右端の AP 減少量は左端の AP 上昇量に比べて小さい．0.01 以下の微小な変化を無視したとして，ΔAP が 0.01 より大きい（増加）バグレポートは 20，-0.01 より小さい（減少）バグレポートは 17，それ以外の変化の小さいものが 37 あり，減少したものよりも増加したものが多かった．また，AP の定義により，ΔAP の値には上位の正解モジュールの順位変動が強く影響し，下位の正解モジュールの順位変動の影響が小さい．これらのことから，提案手法の導入が，ランキング上位の正解モジュールをより高い順位に引き上げる効果を与えたものと考える．今後，より多様な観点からこういった提案手法の効能を分析していきたい．

3.6 妥当性の脅威

内的妥当性．本実験では 4 つのプロジェクトを用いたが，プロジェクトのバグレポートやソースコードの質が脅威となる．バグレポートの質が悪ければ悪いほど，IR 手法の重要度が下がる．また，ソースコードの質が悪ければ悪いほど，不吉な臭いが多く検出される可能性が高いため，不吉な臭いの重要度が上がる．しかし，データセットとして整備されているプロジェクトのバグレポートやソースコードを用いたため，この脅威は小さいと考える．

外的妥当性．本実験ではソースコードが Java 言語で書かれているプロジェクトに関して検証したが，他のプログラミング言語でも同じように手法を用いて MAP が上がるとは限らない．また，4 つのプロジェクトに対してしか検証できていないため，どのようなプロジェクトでも MAP が向上するとは言えない．そのため，他のプログラミング言語や他の多くのプロジェクトで検証する必要がある．

4. 関連研究

4.1 Bug Localization

IR 手法は Bug Localization の分野で幅広く用いられており，ソフトウェア開発の際に作成されたバグに関する記述であるバグレポートと，現在のソースコードの 2 つを入力として用いる．これまでに，Latent Dirichlet Allocation（LDA）[2, 19]，Latent Semantic Indexing（LSI）[3, 20]，Vector Space Model（VSM）[4] など，多くの IR 手法が提案されている．Rao ら [21] は，これらの代表的な手法の中でも VSM が最も効果的であると述べている．この VSM に加えて，過去の類似のバグレポートを用いた BugLocator [22] が Zhou らによって提案されている．

IR 手法では，バグレポートの質に強く依存するため，情報不足のバグレポートでは，良い結果が得られないことが問題となる．Kim ら [23] はバグレポートの質が十分かを判定し，十分な質であれば IR 手法を適用してバグの箇所を予測し，そうでなければ予測を行わない手法を提案している．Le ら [24] も IR に基づくツールでの結果が効果的かを判定する手法を提案している．これらの手法を用いれば，開発者は IR 手法が誤って提案してしまったモジュールに時間を取られずに済む．他にも Chaparro ら [25] は，低品質なバグレポートを再構成する手法を提案している．

静的解析は古くから用いられており，その典型的なものに静的プログラムスライシング [5] がある．静的スライシングでは，ある関心事に対応するスライス基準を与え，プログラム中の依存関係に注目して，関心事に関係のあるモジュール群を抽出する．この手法では，開発者がスライス基準を与えなければならない点，依存関係のあるモジュールが膨大となる点など，実践には多くの課題が残っている．Acharya ら [26] は，近年の大規模なソフトウェア開発に対するプログラムスライシング技術の問題点とその改善案について言及している．

動的解析手法では，実際にプログラムを実行した際に得られる情報を用いて，Bug Localization を行う．Wong ら [7] はスタックトレースを用いたツール BRTracer を提案した．スタックトレースには，テストに失敗するまでの間にどのような処理をどのような順番で呼び出したのかが表現されており，この情報を活用する．また，プログラムの実行トレースから，実行に成功したトレースと失敗したトレースの間のスペクトル統計情報を分析したスペクトルベースの Bug Localization も提案されている [27]．Dao ら [6] は，IR に基づく Bug Localization に動的な実行情報がどのように役立つかを調査している．この調査では，失敗したテストのカバレッジや，動的なスライス情報が IR 手法の検索範囲を効果的に減らすなど，改善が見られることが報告されている．動的解析手法では，高い精度を持つ代わりに，開発者が実際にプログラムを実行する必要があり，開発者に高い知識と時間的な手間を要求することとなる．

Shi ら [28] は IR に基づく Bug Localization に追加的な情報を付与し統合することが有益であると述べている．Saha ら [8] はソースコード構造に基づいた情報を用いて IR に基づく手法を適用した上で，利用可能であれば過去の類似のバグレポートを用いるツール BLUiR を提案した．他にも，Wang ら [9] は，BugLocator [22] と BLUiR [8]，さらに Bug Prediction ベースの変更履歴を統合したツール AmaLgam を提案している．さらに，Youm ら [10] はバグレポートやソースコード構造情報，変更履歴，スタックトレースを用いた動的情報を統合したツール BLIA を提案している．これらの追加的な情報を含める手法は IR ベースの手法に比べて大きく精度を向上させてきた．ただし，Garnier ら [29] は，BLUiR [8] や AmaLgam [9] の有効性に

対して疑問を抱き，問題点の分析を行っている．また，これらの追加的な情報は必ず利用可能であるとは限らず，開発者の入力を必要とするものもある点に注意が必要である．

不吉な臭いと組み合わせた例もある．Abreu ら [30] は Bug Localization に不吉な臭いを組み合わせる手法を提案している．しかし，この手法はスペクトルベースの Bug Localization との組合せである点，適用対象がソースコードではなくスプレッドシートである点が提案手法と異なる．

4.2 不吉な臭い

Fowler [11] によれば，不吉な臭いを持つソースコード内のモジュールは設計や実装などに問題があり，リファクタリングを行うべきと述べられている．Chatzigeorgiou ら [31] は，モジュールを開発する過程で，不吉な臭いがどのように増減するかを調査している．Yamashita ら [13] はソフトウェア保守において不吉な臭いがどの程度影響力を持っているのかを調査している．他にも，不吉な臭いを持つソースコード上のモジュールは，将来的に変更されやすく，バグが出やすいという調査もある [12, 15]．Palomba ら [15] は，不吉な臭いが，将来的に変更されたりバグが起きやすい可能性が高いことを大規模な実験により示している．不吉な臭いには，多くの種類が [11] の中で紹介されているが，それらの不吉な臭いに密度 [32] や深刻度 [16] を割り当て，不吉な臭いに優先順位付けを行う手法もある．これらと同様に，本論文でも深刻度に注目し，詳細な順位付けを試みている．また Palomba ら [33] は Bug Prediction に関して，不吉な臭いを組み合わせることの有意性について述べている．Bug Localization とは異なり，Bug Prediction では具体的なバグレポートを用いずにソースコード上のバグを特定することを目的としている．

5. おわりに

本論文では，Bug Localization 手法の中でも基盤となる IR 手法に対して，不吉な臭いの情報を組み合わせた新しいメトリクス及びそれを用いたモジュールの推薦手法を提案した．4 つのプロジェクト，クラスレベルとメソッドレベルの 2 粒度に関して，既存の IR 手法と提案手法を比較したところ，既存の IR 手法に比べて最大で 269%の MAP の向上が見られた．また，どのプロジェクトに関しても MAP は向上しており，減少しているプロジェクトはなかった．本手法で用いた不吉な臭いはソースコードのみを入力とするため，IR 手法を適用できる状況では適用が可能である．すなわち，これまでに IR 手法をベースの手法とした発展的な Bug Localization 手法においても，ベースとする IR 手法として本論文の提案手法を用いることが可能である．

本論文の今後の課題を以下に挙げる．

- **提案手法のパラメータ α をより厳密に定める．**提案手法を実際に用いるためには，パラメータ α の値を定める必要がある．例えば，対象プロジェクトの過去のバグレポートとその際に変更されたモジュールの情報を用いて，RQ_1 での調査と同様に α の最適値を計算しておくことにより，プロジェクト特有の α を得られると考える．こういった試みを含めた，パラメータ調整のための枠組みが必要である．
- **より多くの評価実験の実施．**本論文での評価では 4 つのプロジェクトのみを対象としており，またそれらのいずれも Java 言語を用いたものである．そのため，本手法がどのようなプロジェクトに対しても効果的な手法であることを十分には示せていない．より多くのプロジェクトに関して提案手法を検証し，効果的であることを示す必要がある．また，Zhou ら [22] が Bug Localization 手法の評価をするためのデータセットを提案しており，提案手法もこういったデータセットを用いて評価したい．さらに，本論文では IR 手法のみに不吉な臭いの深刻度を組み合わせた手法が精度向上に繋がることを示したものの，IR 手法よりも高精度の手法に対しても同様に精度が向上することを示す必要がある．IR 手法に実行情報や過去の変更履歴などの追加的な情報を組み合わせた手法に対しても，不吉な臭いの深刻度を組み合わせた際に精度が向上するかを検証することが望ましい．

謝辞 本研究の一部は，日本学術振興会科学研究費補助金（JP15K15970，JP15H02683，JP15H02685）の助成を受けた．

参考文献

[1] Thung, F., Lo, D., Jiang, L. and Lucia: Are faults localizable?, *Proc. MSR*, pp. 74–77 (2012).
[2] Nguyen, A. T., Nguyen, T. T., Al-Kofahi, J., Nguyen, H. V. and Nguyen, T. N.: A topic-based approach for narrowing the search space of buggy files from a bug report, *Proc .ASE*, pp. 263–272 (2011).
[3] Marcus, A., Sergeyev, A., Rajlich, V. and Maletic, J. I.: An information retrieval approach to concept location in source code, *Proc. WCRE*, pp. 214–223 (2004).
[4] Gay, G., Haiduc, S., Marcus, A. and Menzies, T.: On the use of relevance feedback in IR-based concept location, *Proc. ICSM*, pp. 351–360 (2009).
[5] Weiser, M.: Programmers use slices when debugging, *Comm. ACM*, Vol. 25, No. 7, pp. 446–452 (1982).
[6] Dao, T., Zhang, L. and Meng, N.: How does execution information help with information-retrieval based bug localization?, *Proc. ICPC*, pp. 241–250 (2017).
[7] Wong, C.-P., Xiong, Y., Zhang, H., Hao, D., Zhang, L. and Mei, H.: Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis, *Proc. ICSME*, pp. 181–190 (2014).
[8] Saha, R. K., Lease, M., Khurshid, S. and Perry, D. E.: Improving bug localization using structured information retrieval, *Proc. ASE*, pp. 345–355 (2013).
[9] Wang, S. and Lo, D.: Version history, similar report, and structure: Putting them together for improved bug

localization, *Proc. ICPC*, pp. 53–63 (2014).
[10] Youm, K. C., Ahn, J. and Lee, E.: Improved bug localization based on code change histories and bug reports, *Information and Software Technology*, Vol. 82, pp. 177–192 (2017).
[11] Fowler, M.: *Refactoring: Improving the design of existing code*, Addison-Wesley Professional (1999).
[12] Khomh, F., Di Penta, M., Guéhéneuc, Y.-G. and Antoniol, G.: An exploratory study of the impact of antipatterns on class change-and fault-proneness, *Empirical Software Engineering*, Vol. 17, No. 3, pp. 243–275 (2012).
[13] Yamashita, A. and Moonen, L.: Do code smells reflect important maintainability aspects?, *Proc. ICSM*, pp. 306–315 (2012).
[14] Yamashita, A. and Moonen, L.: Exploring the impact of inter-smell relations on software maintainability: An empirical study, *Proc. ICSE*, pp. 682–691 (2013).
[15] Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R. and De Lucia, A.: On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation, *Empirical Software Engineering*, pp. 1–34 (online), DOI: 10.1007/s10664-017-9535-z (2017).
[16] Marinescu, R.: Assessing technical debt by identifying design flaws in software systems, *IBM Journal of Research and Development*, Vol. 56, No. 5, pp. 9–1 (2012).
[17] Dit, B., Moritz, E. and Poshyvanyk, D.: A tracelab-based solution for creating, conducting, and sharing feature location experiments, *Proc. ICPC*, pp. 203–208 (2012).
[18] Dit, B., Revelle, M., Gethers, M. and Poshyvanyk, D.: Feature location in source code: a taxonomy and survey, *Journal of Software: Evolution and Process*, Vol. 25, No. 1, pp. 53–95 (2013).
[19] Lukins, S. K., Kraft, N. A. and Etzkorn, L. H.: Bug localization using latent dirichlet allocation, *Information and Software Technology*, Vol. 52, No. 9, pp. 972–990 (2010).
[20] Poshyvanyk, D., Marcus, A., Rajlich, V., Gueheneuc, Y.-G. and Antoniol, G.: Combining probabilistic ranking and latent semantic indexing for feature identification, *Proc. ICPC*, pp. 137–148 (2006).
[21] Rao, S. and Kak, A.: Retrieval from software libraries for bug localization: a comparative study of generic and composite text models, *Proc. MSR*, pp. 43–52 (2011).
[22] Zhou, J., Zhang, H. and Lo, D.: Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports, *Proc. ICSE*, pp. 14–24 (2012).
[23] Kim, D., Tao, Y., Kim, S. and Zeller, A.: Where should we fix this bug? a two-phase recommendation model, *IEEE Transactions on Software Engineering*, Vol. 39, No. 11, pp. 1597–1610 (2013).
[24] Le, T.-D. B., Thung, F. and Lo, D.: Will this localization tool be effective for this bug? Mitigating the impact of unreliability of information retrieval based bug localization tools, *Empirical Software Engineering*, Vol. 22, No. 4, pp. 2237–2279 (2017).
[25] Chaparro, O., Florez, J. M. and Marcus, A.: Using Observed Behavior to Reformulate Queries during Text Retrieval-based Bug Localization, *Proc. ICSME*, pp. 376–387 (2017).
[26] Acharya, M. and Robinson, B.: Practical change impact analysis based on static program slicing for industrial software systems, *Proc. ICSE*, pp. 746–755 (2011).
[27] Naish, L., Lee, H. J. and Ramamohanarao, K.: A model for spectra-based software diagnosis, *ACM Transactions on Software Engineering and Methodology*, Vol. 20, No. 3, pp. 11:1–11:32 (2011).
[28] Shi, Z., Keung, J., Bennin, K. E. and Zhang, X.: Comparing learning to rank techniques in hybrid bug localization, *Applied Soft Computing*, Vol. 62, pp. 636–648 (2018).
[29] Garnier, M., Ferreira, I. and Garcia, A.: On the influence of program constructs on bug localization effectiveness, *Journal of Software Engineering Research and Development*, Vol. 5, No. 1, pp. 1–29 (2017).
[30] Abreu, R., Cunha, J., Fernandes, J. P., Martins, P., Perez, A. and Saraiva, J.: Faultysheet detective: When smells meet fault localization, *Proc. ICSME*, pp. 625–628 (2014).
[31] Chatzigeorgiou, A. and Manakos, A.: Investigating the evolution of bad smells in object-oriented code, *Proc. QUATIC*, pp. 106–115 (2010).
[32] Fontana, F. A., Ferme, V., Zanoni, M. and Roveda, R.: Towards a prioritization of code debt: A code smell intensity index, *Proc. MTD*, pp. 16–24 (2015).
[33] Palomba, F., Zanoni, M., Fontana, F. A., De Lucia, A. and Oliveto, R.: Toward a Smell-aware Bug Prediction Model, *IEEE Transactions on Software Engineering*, (online), DOI: 10.1109/TSE.2017.2770122 (2017).