ソフトウェアバグの行レベル予測の試み

An Attempt to Statement-level Software Bug Prediction

福谷 圭吾*　門田 暁人†　Zeynep Yucel‡　畑 秀明§

あらまし　本稿では，ソースコードに含まれるソフトウェアバグを，行（ステートメント）の粒度で予測することを試みる．提案方法では，ソースコードを探索するためのある大きさを持った窓を用意し，バグを含むソースコードの先頭から最後へ向けて窓をずらして走査を行うことで，バグを含む窓の集合と，含まない窓の集合を作成する．次に，各窓に含まれるソースコードのトークン列を入力として，各窓に対するバグを含む確率を出力とするスパムフィルタベースのバグ予測モデルを構築する．最後に，各窓に対する予測結果を統合し，各行に対するバグ混入確率を算出する．バグを 1 つだけ含むことが予め判明している 593 件のソースコード群を用いた実験の結果，窓の大きさを 3 行とした場合，バグを含む確率が高いと予測された上位 10 行に全バグの 62.2%が含まれることが分かり,バグをピンポイントで予測するという本試みの実現可能性を確認できた．

1　はじめに

従来，過去のソフトウェア開発実績データに基づいて，現在進行中の開発プロジェクトのソフトウェアに対し，バグを含むソフトウェアモジュール（機能，サブシステム，ソースファイル等）の予測（以下，バグ予測）の研究が行われてきた[4][7]．ただし，バグ予測では，バグが存在する箇所を絞り込むことはできるが，具体的にバグがどこにあるかまでは分からない．バグの発見のためには，バグ予測を行った後に，バグがあると予測されたモジュールに対してソフトウェアテストやレビューを行う必要があった．

本稿では，バグの箇所をさらに絞り込むことを目的として，行レベルのバグ予測を行うことを目指す．ただし，行レベルのバグ予測では，従来のファイルレベルやメソッドレベルにおいて用いられていた，ソースコードメトリクスを入力としたバグ予測モデルを用いることは困難である．なぜなら，従来用いられているサイクロマティック数や，fan-in，fan-out などの構造メトリクス，CBO，LCOM，RFC などのオブジェクト指向メトリクスは，いずれも行単位での計測ができない，もしくは，行単位の計測が前提とされていないためである．そこで本稿では，テキストベースの予測方法である fault-prone フィルタリング[5][6]を採用し，予測モデルとして CRM114[3]の識別器である Orthogonal Sparse Bigrams (OSB)モデルを用いる．Fault-prone フィルタリングは，ソースコード中の

* Keigo Fukutani,岡山大学工学部情報系学科
† Akito Monden, 岡山大学大学院自然科学研究科
‡ Zeynep Yucel, 岡山大学大学院自然科学研究科
§ Hideaki Hata, 奈良先端科学技術大学院大学情報科学研究科

トークン列を直接入力とするため，メトリクスの計測の必要がない．

また，ある行にバグを含むか否かの判別は，その前後の行の情報が必要となる場合があることから，提案方法では，複数行をまとめたものを 1 つの窓とし，窓ごとに予測を行った後に，それらの結果を統合して行ごとの予測値（バグを含む確率）を得ることとする．予測モデルの学習にあたっては，バグを含むソースコードの先頭から最後へ向けて窓をずらして走査を行うことで，バグを含む窓の集合と，含まない窓の集合を作成し，それらを OSB モデルへの入力とする．

評価実験では，バグを 1 つだけ含むことが予め判明しているソースコード群を用いた実験を行う．実験の題材としては，Codeflaws データセット[1]から，8 種類のバグを含む 593 個のソースコードを対象とした．また，実験では， 3-fold cross validation を行うことで，提案方法の効果を評価する．

2 提案方法

提案方法は，学習フェーズと予測フェーズから構成される．学習フェーズでは，バグを含むソースコードに対し，ある大きさ（n 行）の窓を設け，ソースコードの上端から下端に向けて 1 行ずつ窓をずらして走査する．ソースコードの行数を x，それぞれの行を s_i $(1 \leqq i \leqq x)$，窓のサイズを n（$n \leqq x$），行 s_k から行 s_{k+n} に対応する窓を $w_k=\{s_k,\ldots, s_{k+n}\}$ とすると，窓の数は x-n+1 個となり，それらは $w_1, \ldots, w_{x-n+1}$ と表すことができる．これらの窓を，(a)バグを含む窓の集合，(b)バグを含まない窓の集合のいずれかに分類し，両者を判別するためのテキスト識別器を学習させる．

バグ特定フェーズでは，バグをどの行に含むか分からないソースコードにおいて，予測したい行，すなわち，バグを含む確率を知りたい行を s_i $(1 \leqq i \leqq x)$とする．s_i を含むように窓を 1 行ずつずらして走査し，窓の集合 $W_i=\{w_j \mid s_i \in w_j\}$ を得る．それぞれの窓 $w \in W_i$ をテキスト識別器に入力し，バグを含む確率 $P_{bug}(w)$ を得る．W_i に含まれる窓の数を $|W_i|$ とおくと，行 s_i がバグを含む確率 $P_{bug}(s_i)$ を式(1)により算出する．

$$P_{bug}(s_i) = \frac{1}{|W_i|} \sum_{w \in W_i} P(w) \tag{1}$$

式(1)では，行 s_i を含む窓のバグ含有確率の単純平均により，行 s_i のバグ含有確率を求めている．この予測を，ソースコードの全ての行に対して行うことで，バグ含有確率に基づいて各行を順位付けすることが可能となる．

提案方法では，各行に対して直接予測を行うのではなく，ある程度の大きさの窓を設けて予測を行った後に，それらを統合して各行に対する予測値を算出している．このような 2 段階の予測方法を採用した理由は 2 つある．一つは，ある行がバグを含むか否かを判定する上で，その前後の行の情報が必要となると考えたためである．一般に，人間がデバッグを行う場合，バグの発見のためには，バグを含む行だけを見ていてもバグの存在に気が付くことができず，前後の行を読むことが必要となる場合があることから，提案方法は，前後の行を含めた窓を用いた機械学習を行うこととした．もう一つの理由は，バグを含む行は含まない行と比べて数が著しく少ないことから，何らかの方法でバグを含むケースを増やすことで，モデルの学習に必要なデータ量を確保したかったためである．提案方法では，窓のサイズ n を大きくすることで，バグを含むケースを増やすことが可能となる．

3 評価実験

3.1 題材

使用するデータセットとして，Codeflaws[1]を使用した．Codeflaws は，元来，自動バグ修正の研究のために整理されたデータセットであり，プログラミングコンテスト Codeforces[2]のプログラムデータベースから抽出された，バグを含む多数の C 言語プログラムを含んでいる[8]．バグは，文，演算子，被演算子，その他の 4 種類に大別され，それぞれさらに細かくカテゴリに分けられ，合計 40 種類に分類されている．

ただし，全てのバグの位置をソースコードのみの情報から自動的に予測することは困難である．そこで，本稿では，研究の第一歩として，「演算子」カテゴリに焦点を当て，その中の 8 つの種類のバグのみを対象とした．つまり，演算子の誤りに関するバグのみを自動特定することを目指すこととなる．これら 8 種類のバグは，全 40 種類のバグのおよそ 15%に相当する．8 種類以外のバグに対する予測は，今後の重要な課題となる．

また，バグが複数の行に及ぶ場合や，1 つのプログラム中に複数のバグが含まれる場合，モデルの構築や評価が複雑となり，提案アプローチの実現可能性の評価が困難となる恐れがあるため，本稿では対象外とした．また，バグを含まないプログラムも対象外としている．結果として，特定の 1 行に 8 種類のいずれかのバグを 1 つだけ含む 593 個のプログラムを実験対象とした．8 種類のバグの詳細を表 1 に示す．表に示されるように，ORRN（比較演算子の置換）と OAIS（算術演算子の削除／挿入）に属するバグが多く，OICD（条件演算子の挿入）と OMOP（演算子の優先順位の変更）に属するバグは少ない．

実験対象となるプログラム群の行数は，最小で 6 行，最大で 198 行であった．また，行数の平均値は 38.2，中央値は 32 であった．

3.2 手順

今回の実験の手法として 3-fold cross validation を用いる．まず実験用データ（593 件）を無作為に 3 等分し，2 つを学習，1 つを評価に用いる．学習用のデータと評価用のデータを入れ替えて計 3 通り行う．窓のサイズは，本稿では n=3 に固定している．異なる n に対する実験は今後の重要な課題となる．

本稿で用いた OSB モデルは，プログラム中の各行がバグを含む確率を算出できる．そこで，OSB モデルによる予測の結果，バグを含む確率の高い順に行を順序付けした場合と，無作為に行を順位付けした場合とを比較し，予測の効果を評価する．また，バグの種類ごとの評価も行い，バグの種類間で予測精度に差があるかどうかを分析する．

3.3 結果

OSB モデルによる予測に基づいて行を順位付けした場合（予測あり）と，無作為に順位付けを行った場合（予測なし）の結果を cumulative-lift chart として表したものを図 1 に示す．図 1 のグラフの横軸は行の順位を示しており，縦軸は，それぞれの順位までに実際にバグを含んでいたプログラムの割合を示している．

図 1 より，無作為に行を順位付けした場合，上位 10 行に全バグの約 36.3%含まれているのに対し，予測を行った場合では，上位 10 行に全バグの約 62.2%含まれていた．このことから，予測を行うことで，ある程度バグ位置を絞り込めているといえる．その一方で，予測を行った場合においても，上位 1 行だけに着目した場合には 11.8%にとどまることから，現状ではピンポイントでの予測は難しいといえる．ただし，上位 3 行に 30.2%

表 1　バグの分類（[1]より抜粋）

タイプ名	クラス名	説明	例	データ数
Control flow	ORRN	比較演算子の置換	- if(sum>n) + if(sum>=n)	286
	OLLN	論理演算子の置換	- if((s[i] == '4') && (s[i] == '7')) + if((s[i] == '4') || (s[i] == '7'))	17
	OEDE	=と==の置換	- else if(n=1 && k==1) + else(n==1 && k==1)	17
	OICD	条件演算子の挿入	- printf("%d¥n", i); + printf("%d¥n", 3 == x ? 5 : i);	4
Arithmetic	OAAN	算術演算子の置換	- v2-=d; + v2+=d;	29
	OAIS	算術演算子の削除/挿入	- max += days%2; + max += (days%7)%2;	204
	OAID	++, --の挿入/削除/置換	+ i++;	27
	OMOP	演算子の優先順位の変更	- ans=max(ans,l-arr[n]*2); + ans=max(ans,(l-arr[n])*2);	9

のバグを含むことから，プログラム中の 3 行に着目することで 3 割のバグの発見に役立つといえる．

予測に成功したプログラム（上位 1 行にバグが含まれていたプログラム）と，予測に失敗したプログラム（上位 20 行にバグが含まれていなかったプログラム）の例を付録に示す．前者については，バグ種別は ORRN（比較演算子の置換）であり，バグは 14 行目で，全体で 35 行あるうちの 1 位であった（"if (Amax<arr[i])" とするのが正しい）．後者については，バグ種別は ORRN（比較演算子の置換）であり，バグは 12 行目で，全体で 33 行あるうちの 27 位であった（"if(d<=0) {" とするのが正しい）．

また，バグ種別ごとの cumulative-lift chart を図 2 に示す．図 2 に示されるように，OAID（++, --の挿入/削除/置換）は他のバグと比べて予測精度が低い．このことは，バグ修正時に++や--をどこに挿入すべきかをソースコードの字面だけから判断することが難しいことを示唆している．また，一般に，バグは，ソースコード中の複雑な部分に混入することが多いが，++や--を挿入すべき位置は，むしろプログラム中の複雑でない部分が多いことから，予測しづらいのではないかと推察される．

また，図 2 より，OEDE（=と==の置換）と OMOP（演算子の優先順位の変更）は他のバグと比べて，上位 5 行，上位 10 行における予測精度が高い．これらのバグは，他のバグと比べると，ソースコードの字面だけからバグの有無を判断しやすいのではないかと

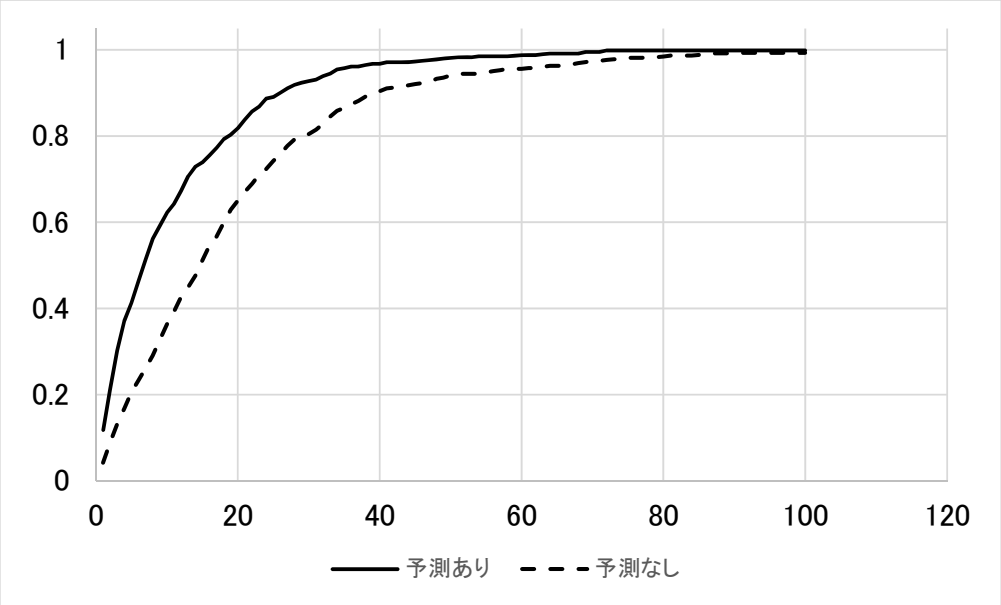

図 1　予測結果の Cumulative-lift グラフ

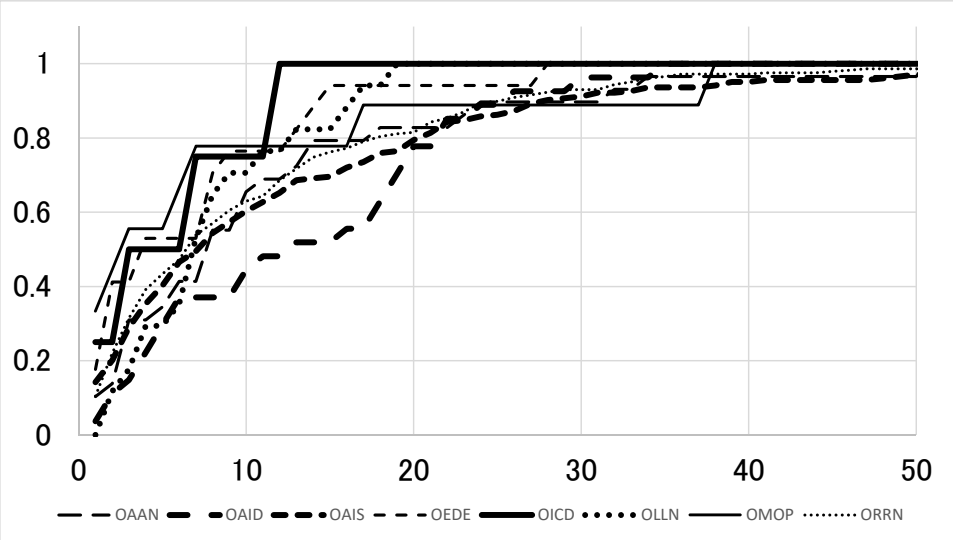

図 2　バグ分類ごとの cumulative-lift グラフ

推察される.

4　まとめ

本稿では，ソースコードに含まれるソフトウェアバグを，行の粒度で予測することを

試みた．593 個のプログラムを対象とした実験において，ソースコードを走査する窓の大きさを 3 とし，予測モデルとして Orthogonal Sparse Bigrams (OSB)モデルを用いた結果, バグを含む確率が高いと予測された上位 10 行に全バグの 62.2%が含まれることが分かった．ランダム予測では，上位 10 行に含まれるバグは全バグの 36.3%であったことを考慮すると，バグをピンポイントで予測するという本試みの実現可能性を確認できた．

5 参考文献

[1] Codeflaws, https://codeflaws.github.io/
[2] Codeforces, http://codeforces.com/
[3] CRM114 – the Controllable Regex Mutilator, http://crm114.sourceforge.net/
[4] 畑秀明, 水野修, 菊野亨, “不具合予測に関するメトリクスについての研究論文の系統的レビュー,” コンピュータソフトウェア, vol.29, no.1, pp.106-117, Feb. 2012.
[5] 水野修, 菊野亨, “Fault-prone フィルタリング : 不具合を含むモジュールのスパムフィルタを利用した予測手法,” SEC journal, No.13, pp.6-15, Feb. 2008.
[6] Osamu Mizuno, Hideaki Hata, “Prediction of fault-prone modules using a text filtering based metric,” International Journal of Software Engineering and Its Applications, Vol.4, No.1, pp.43-52, Jan. 2010.
[7] A. Monden, T. Hayashi, S. Shinoda, K. Shirai, J. Yoshida, M. Barker, K. Matsumoto, “Assessing the cost effectiveness of fault prediction in acceptance testing,” IEEE Trans. on Software Engineering, vol.39, no.1, pp.1345-1357, 2013.
[8] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, “Codeflaws: a programming competition benchmark for evaluating automated program repair tools,” Companion Proc. 39th International Conference on Software Engineering (ICSE2017), pp.180-182, 2017.

6 付録

予測が的中したプログラムの例（抜粋）

```
01: #include <stdio.h>
02: #include <stdlib.h>
03:
04: int main(int argc, char *argv[])
05: {
06:         int L, b, c, i, arr[105], Amax=0,
Alow=999, m, n, res=0;
07:       scanf("%d", &L);
08:       for (i=1; i<=L; i++)
09:       {
10:             scanf("%d", &arr[i]);
11:       }
12:       for (i=1; i<=L; i++)
13:       {
14:             if (Amax<=arr[i])
15:             {
16:                   Amax=arr[i];
17:                   n=i;
18:             }
```

予測が外れたプログラムの例（抜粋）

```
01: #include <stdio.h>
02: #include <math.h>
03:
04: int main(int argc, char *argv[])
05: {
06:       int n,d,l;
07:       scanf("%d%d%d",&n,&d,&l);
08:       int pl=n/2+n%2, min=n/2;
09:       int add[pl],
10:             sub[min];
11:       int i;
12:       if (d<0) {
13:             for (i=0; i<pl; i++) add[i]=1;
14:             d=pl-d;
15:             for (i=0; i<min; i++) {
16:                   sub[i]=d/(min-i);
17:                   d-=sub[i];
18:                     if (sub[i]>l) {printf("-
1"); return 0;}
```