修 士 論 文

題　目　　単語分散表現を用いた
　　バグ報告からの不具合ファイル特定

主任指導教員　　水野　修　准教授

京都工芸繊維大学大学院 工芸科学研究科

情報工学専攻

学生番号　　14622006

氏　　名　　采野　友紀也

平成28年2月10日提出

様式 2 号

学位論文の要旨（和文）

平成 28 年 2 月 10 日

京都工芸繊維大学大学院
工芸科学研究科長　殿

工芸科学研究科　　情報工学専攻
平成 26 年入学
学生番号　　14622006
氏　　名　　釆野　友紀也　㊞

（主任指導教員　　水野　修　㊞　）

本学学位規則第 4 条に基づき、下記のとおり学位論文内容の要旨を提出いたします。

1.　論文題目

単語分散表現を用いたバグ報告からの不具合ファイル特定

2.　論文内容の要旨（400 字程度）

ソフトウェアのバグが報告されると，開発者はバグ修正のためにバグに関連するソースコードファイルを特定する必要がある．この過程は，**不具合ファイル特定**と呼ばれ，不具合ファイル特定の自動化は開発者の生産性向上のために重要である．本稿では，単語分散表現作成のための自然言語処理ツール `word2vec` を用いて学習されるベクトル空間モデル *semantic-VSM* による，与えられたバグ報告に対する不具合ファイルの効果的な特定手法を提案する．また，我々は不具合ファイル特定の正答率を向上させるために，既存の不具合ファイル特定手法 *BugLocator* 及び *Bugspots* とともに semantic-VSM を用いる組み合わせ手法 *CombBL* を紹介する．我々の実験結果は，提案手法が既存手法に比べ同程度の正答率を維持したまま，高スケーラビリティかつ高速であることを示す．

Using distributed representations of words in Localizing Relevant Files for Bug Reports

2016 14622006 *UNENO Yukiya*

Abstract

Once a bug in software is reported, developers have to determine which source files are related to the bug. This process is referred as *bug localization*, and an automatic way of bug localization is important to improve developers' productivity. This paper proposes an approach that efficiently localizes faulty files for a given bug report by feeding our vector space model named *semantic-VSM* to a NLP (Natural Language Processing) tool, `word2vec`, which is used to generate distributed representations of words. We also present an approach called *CombBL* which combines using our semantic-VSM with existing bug localization techniques such as *BugLocator* and *Bugspots*, to improve the accuracy of fault localization. Our experimental results show that the proposed approach is more scalable and faster with the same level of accuracy on bug localization compared to existing approaches.

目　次

1. 緒言

バグ報告はソフトウェアの不具合すなわち**バグ**の情報を含む文書であり，ソフトウェア開発者によって記述される．与えられたバグ報告に対して，**不具合ファイル特定** *(Bug Localization)* はバグ報告に潜在的に関連する不具合ファイルを特定し，開発者を支援する．

これまでに複数の研究者が与えられたバグ報告とソースコードファイルから重要な特徴を取り出し，潜在的な不具合ファイル特定を自動化する研究に対して提案を行ってきた．文献 [1–3] の手法は**情報検索** *(Information Retrival; IR)* を用いており，文献 [4,5] の手法は**機械学習** *(Machine Learning; ML)* を用いている．また，文献 [6] の手法は IR 及び ML 手法の併用手法である．既存手法における共通の課題はバグを記述するためのバグ報告に用いられる項の語彙的意味がソースコードファイルにおける識別子の語彙的意味と異なるという，語彙の不整合の克服である．

Lam ら [7] はバグ報告とソースコードファイルの関連度を学習する *DNN (Deep Neural Network)* を用いてこの問題に対処している．彼らは DNN，**改良ベクトル空間モデル** *(revised Vector Space Model; rVSM)*，プロジェクトのバグ修正履歴から構成される特徴を組み合わせた *HyLoc* と呼ばれる彼らのモデルの中で，最高水準の IR 及び ML 手法より高い正答率を達成している．しかし，彼らの DNN 過程はそれほどスケーラブルではなくまた一定の時間を要する．

本稿では自動かつ効果的は不具合ファイル特定のための**組み合わせ不具合ファイル特定** *(Combined Bug Localization; CombBL)* と呼ぶ新たな手法を紹介する．我々の主な発想は，ソースコードファイル内のコード識別子や項と潜在的に異なるバグ報告内の項の関連性を学習するために，リカレントニューラルネットワーク言語モデル (Recurrent Neural Network Language Model; RNNLM) [8] を起点に研究された，`word2vec` [9,10] と呼ばれる既存の**単語分散表現** *(Distributed representations of words)* の作成ツールを利用することである．この目的で，我々は `word2vec` を用いて学習される**意味ベクトル空間モデル** *(semantic Vector Space Model; sVSM)* を提案する．提案する sVSM を用いることにより，バグ報告とソースコードファイルの特徴は，自動的に統合された後に，直接低次元高密度な単一ベクトル空間に射影される．我々はまた我々の sVSM を情報検索 (IR) 手法 rVSM と組み合わせて用いる．rVSM はバ

グ報告とソースコードファイル間のテキスト類似性に基づく特徴を収集する．

我々のsVSMの使用による主要な利点は以下の２点にまとめられる．

第一点目 我々のモデルでは高次元射影や深層ニューラルネットワークが不要な点である．このことが我々の手法を高スケーラビリティかつ高速演算にしている．

第二点目 多くのベクトル空間モデルにおいて見られるように，各項が**単語ベクトル**と呼ばれる分散表現として表現されるだけでなく，複数の単語のベクトル表現の和として表され，それらのフレーズが表す慣用的な意味を表現することが可能であること [10] である．単語ベクトルを要素とし，我々はバグ報告に対応するクエリベクトル及びソースコードファイルに対応する文書ベクトルを容易に得ることができる．ベクトルを用いて，我々はコサイン類似度の計算によりバグ報告とソースコードファイル間の関連度を直接的に評価できる．

我々のオープンソースソフトウェアにおける実証実験は我々のsVSMを用いると [7] におけるDNN単独とほぼ同程度の不具合ファイル特定の正答率を達成している．さらに，sVSM，コンポーネント情報，rVSM [3] によるテキスト類似度，Bugspots [11] によるプロジェクトのバグ修正履歴からのスコアの組み合わせを用いた我々のCombBLは，sVSM単独よりも高い正答率を達成している．我々は提案するCombBLが最高水準のIR及びML手法より少ない計算時間で同程度の正答率を達成できることを確かめた．本稿は以下に貢献する．

1. 既存の単語分散表現作成ツール `word2vec` を用いた不具合ファイル特定のためのsVSMと呼ばれる手法を提案し，
2. sVSMを含む４つの構成要素の組み合わせによる不具合ファイル特定のためのCombBLと呼ばれる手法を導入し，また
3. 先行手法に対するCombBLの利点を提示するためのオープンソースプロジェクトにおける実証実験を行った．

本稿の構成を以下に示す．2章では研究の基礎となる理論や概念及び関連研究について述べる．3章では本稿で提案する不具合ファイル特定手法について詳しく説明する．5章ではオープンソースプロジェクトのリポジトリを用いて提案手法を適用した実証実験の結果及び考察を述べる．6章でまとめと今後の課題について述べる．

2. 不具合ファイル特定と単語分散表現

本章では，不具合ファイル特定 (Bug Localization) と単語分散表現 (Distributed representations of words) の概念及び関連研究について説明する．2.1 節では，情報検索モデルを用いた不具合ファイル特定に関する研究 [3] に従い，不具合ファイル特定の例及び一般的な手法を説明する．2.2 節では，不具合ファイル特定の様々な手法による先行研究とその課題を説明する．2.3 節では，単語分散表現の概念と不具合ファイル特定への応用について説明する．

2.1 不具合ファイル特定

2.1.1 不具合ファイル特定の例

本章では，不具合ファイル特定 (Bug Localization) 手法を説明するための例を紹介する．不具合ファイル特定手法を説明するために，まずバグ報告について説明する．バグ報告は，開発者がソフトウェア開発においてソフトウェアの品質維持のために，変更を加えるべき不具合について記述する文章である．バグ報告には，事後保守，欠陥修正，機能拡張，リファクタリングなどの様々な変更要求が含まれる [12].

表 2.1 は本稿で実験の対象として用いるプロジェクト Tomcat における一つのバグ報告 (ID: 32157) を示している．このバグ報告が報告されると，開発者はこのバグを修正するために数千の Tomcat ソースコードファイルの中から関連ファイルを特定する必要がある．一見して，バグ報告（bug summary 及び description を含む）は JSP(=jasper の省略形)，Compiler などの語を含むことがわかる．従って，このバグは JSP Compiler についての特徴に関連していると推測できる．Tomcat プロジェクトの中には，AntCompiler.java (注 1) と名付けられた関連するソースコードファイルがあり，ここでも類似の語の出現頻度が高いと推測できる．

バグ情報と関連するソースコード編集履歴を照らし合わせることにより，AntCompiler.java は実際にこのバグを修正するために変更が加えられたファイルであること

(注 1)：ただし，ファイルのベースネームだけではファイルを一意的に特定できない点には注意が必要である．理由は，ファイルのベースネームが同一でありながら，ディレクトリ構造における保存場所も追加した名前 (フルパスネーム) の異なるファイルは複数存在することがあるため．ここで述べている AntCompiler.java は正確には java/org/apache/jasper/compiler/AntCompiler.java である．

表 2.1　バグ報告の一例と関連するソースコードファイル

Bug ID: 31257
Status: RESOLVED FIXED
Component: Jasper
Reported time:　2004-09-16 05:10 UTC by James DeFelice Fixed time:　2008-07-30 11:14 UTC
Summary: Bug 31257 - java.endorsed.dirs is not used when **JSP compilation** is forked
Description: When the **JSP compiler** forks a javac process, it should check to see what the currently endorsed directories are (check the system property "java.endorsed.dirs" - otherwise, classes end up missing and **JSP** pages do not properly **compile**. We experienced a problem when we explicitly set the java.endorsed.dirs property on the command line, and the **JSP compiler** did not use it when javac was forked. ...
Fixed file: java/org/apache/jasper/compiler/AntCompiler.java

が判明している．このようなバグ報告に対する修正済みファイルの情報は，予め SZZ アルゴリズム [13] などの方法を用いて得られる．SZZ アルゴリズムは，バグ管理システムに記録されたバグ情報とソースコードの編集履歴を対応付け，欠陥の発生要因となったコードの修正を特定する．なお，本研究では Ye ら [6] によって提供された，バグ報告と関連する修正済みファイルの情報が既に取得済みのデータセットを用いるため，直接その操作は行っていない．

また，バグ報告に関連する修正済みファイルについて注意すべきこととして次の点がある．本例では，バグを修正するために変更が加えられたファイルは 1 つであるが，バグを修正するために変更が加えられたファイルの数は複数存在する場合もある．

さらに，バグ報告には不具合ファイルを特定するのに十分な情報が含まれていない場合があることが知られている []．不具合ファイル特定手法は，このような不具合ファイルを特定するために，十分な情報が含まれていないバグ報告に対しても正しく機能することが望まれる．本稿では，この点に関する調査及び改善は行っていないので，この点について注意が必要である．

我々は，バグ報告及びソースコードファイルをテキスト文書として扱いそれらの間のテキスト類似度を計算する．ファイルコーパスに対して，我々はバグ報告に対するテキスト類似度に基づき各ファイルを順位付けする．開発者は関連ファイルを見つけるまで順位付けされたリストの最初から一つ一つファイルを調査する．このような手順を用いれば，バグ報告に関連するファイルは素早く特定できる．つまり，不具合ファイル特定の最終目的はリストにおいて不具合ファイルをより高順位に順位付けすることである．

2.1.2 不具合ファイル特定手順

情報検索モデルにおける，以下の 4 手順からなる一般的な不具合ファイル特定手法を説明する．手順は，コーパス作成，インデキシング，クエリ作成，及び検索と順位付けからなる．

コーパス作成 本手順ではソースコードに対する語彙解析を行い，語彙トークンのベクトルを作成する．複数の識別子すなわち，int, double, char などのキーワード,

セパレータ，及び演算子は全てのプログラムに共通であり，取り除かれる．英語のストップワード (‘a’,‘the’ など) も取り除かれる．プログラム内で定義された多くの変数は実際は単語の合成である．例えば，変数 TypeDeclaration は 2 語 “type” と “decolaration” を含む．変数 “isCommitable” は 2 語 “is” と “Commitable” から成る．これらの合成後は個別の識別子に分割される．多くの識別子は同じ原型を持っている．例えば，“delegating”，“delegate”，及び “delegation” は同じ原型 “delegat” を共有している．単語を原型に戻すために語幹抽出アルゴリズム Porter Stemming algorithm [14] が適用される．

インデキシング コーパス作成後，コーパス内の全てのファイルはインデキシングされる．これらのインデックスを用いて，与えられたクエリ内の単語を含むファイルを特定し，それらの関連度に従いファイルを順位付けする．

クエリ作成 不具合ファイル特定では，バグ報告をクエリとみなし，インデキシングされたコードコーパス内の関連ファイルを検索に用いる．バグの題名や (summary や) 記述内容から識別子を抽出し，ストップワードを取り除き，各語を語幹抽出し，クエリを形成する．

検索と順位付け クエリとコーパス内の各ファイル間のテキスト類似度に基づき関連する不具合ファイルを検索及び順位付けする．入力として与えられたバグ報告とコーパス内の各ファイル間の関連度スコアを計算するために様々な手法が用いられる．

2.1.3 既存の情報検索モデル

多くのバグ報告からの不具合ファイル特定が提案されてきた．それらの手法は主に結果の検索と順位付けにおいて違いがある．IR に基づく先行研究で用いられてきた多くの検索と順位付けモデルがある．

SUM Smoothed Unigram Model (SUM) はコーパス内の各ファイルの単語の頻度に対する単一の multinomial 分布にフィットする統計的モデルである [15]．単語の頻度から直接由来する unigram model (UM) はいくつかの問題に直面している．特に，明確に出現していない単語に直面するとき，すなわち未出現の単語に対する確率がゼロの場合．SUM は未出現の単語に対してゼロでない確率を割り当

てることによって確率分布を滑らかにしている [16,17]．SUM は Rao と Kak [18] らのバグ報告からの不具合ファイル特定のために用いられ，それが最も性能が高くなるモデルであると確かめられた．

LDA Latent Dirichlet Allocation (LDA) はテキストコーパスのような個々のデータの集まりに対する生成される確率によるモデルである [19]．それはベイジアンモデルであり，文書の集まりからの最新のトピックを抽出する．各トピックは割り当てられた確率を持つトークンの集まりである．各文書はトピックの確率的な組み合わせによって表現される．この手法は Lukins ら [2] によってバグ報告からの不具合ファイル特定に用いられた．

LSI Latent Semantic Indexing (LSI) [20] は，latent semantic analysis (LSA) とも呼ばれ，Singular Value Decomposition (SVD) のような数学的手法を用いてテキストの非構造的な集まりに含まれる項とコンセプトの間の関連性を識別できるインデキシング及び検索の手法である．この手法は Poshyvanyk らによってバグ報告からの不具合ファイル特定に用いられた [1,21]．

VSM Vector Space Model (VSM) では，各文書は各トークンの出現頻度 (token frequency) と逆文書出現頻度 (inverse document frequency) の積で典型的に計算されるトークンの重みのベクトルによって表現される [22]．コサイン類似度がどの程度２つのベクトルが近いかを決定するために広く用いられる．Rao と Kak [18] はバグ報告からの不具合ファイル特定における VSM の性能を評価し，SUM には劣るが (LDA や LSI) を含むその他の手法に比べ優れていることを確かめた．

2.2 不具合ファイル特定に関する先行研究

本節では，不具合ファイル特定に関する先行研究を紹介する．2.2.1 節では，Lam ら [7] により紹介された不具合ファイル特定手法の先行研究，及び Lam ら自身の研究を紹介する．2.2.2 節では，本研究で用いる手法に類似する手法を紹介する．2.2.3 節では，program spectra と呼ばれる概念を併用しさらなる精度向上を目指した，2015 年 10 月発表の最も最近の研究を紹介する．

2.2.1 語彙不整合の克服を目指す手法

(1) 基礎的な不具合ファイル特定手法

Lam ら [7] らの報告によると，これまでに複数の研究者が与えられたバグ報告とソースコードファイルから重要な特徴を取り出し，潜在的な不具合ファイル特定を自動化する研究に対して提案を行ってきた．文献 [1–3] の手法は**情報検索** *(Information Retrival; IR)* を用いており，文献 [4,5] の手法は**機械学習** *(Machine Learning; ML)* を用いている．また，文献 [6] の手法は IR 及び ML 手法の併用手法である．

IR に基づくの不具合ファイル特定手法は同じ方針に従う．与えられたバグ報告はクエリとして解釈される．プロジェクト内のソースファイルは文書の集まりとみなされる．バグ報告からの不具合ファイル特定の課題は IR における検索及び順位付けの課題としてモデル化される．Latent Dirichlet Allocation [2,23] は比較のためのトピック特徴を抽出するために用いられる．Zhou ら [3] はさらに先進的なベクトル空間モデルを開発した．また，彼らのモデルは，バグ報告とファイル間のテキスト類似性だけでなく，過去に修正された類似するバグの情報も考慮している．少数のバグ報告からの不具合ファイル特定手法は機械学習に基づいている．BugScout [5] は与えられたバグ報告に記述される技術的なトピックが不具合ファイルのトピックにも含まれるという仮定に基づいた特別なトピックモデルである．トピックモデルはバグ報告のトピックとソースコードファイルのトピックを結びつけるように学習される．Kim ら [4] は分類ラベルとして過去に修正されたファイルを用いて Naive Bayes を適用し，各バグ報告に対してソースコードファイルを割り当てる学習モデルを用いた．Ye ら [6] はソースコードファイル，API の記述内容，バグ修正と変更の履歴から適応性のある学習を用いた．

(2) 語彙不整合の問題

Lam ら [7] らの報告によると，複数の研究者ら [5], [6] は，バグ報告における自然言語とソースコードにおける技術的なトピック間での語彙的意味の不整合が，それら前述の IR に基づく不具合ファイル特定手法の主な制約だと認識している．それら ML に基づく手法も語彙的意味の不整合に直面している．精度を向上させるため，Ye ら [6] と Kim ら [4] は，バグ報告のメタ情報から得られる追加の特徴（例えば，バー

ジョン，プラットフォーム，優先度，など）とソースコードファイルに対するバグ修正履歴の情報を用いた．語彙の不整合を橋渡しするために，Ye ら [6] はソースファイルにおいて使われた API の文書から得られるテキストを追加で用いた．しかし，API 文書はプロジェクトに特化した不具合の振る舞いとしてではなく，より一般的なタスクとして見なすテキストを含んでいる．Kim ら [4] は特徴抽出のためにソースコードファイルの中身を用いていない．彼らはバグ報告の識別ラベルとしてのみ修正されたファイルの名前を用いる．そのため，彼らは新たなバグ報告に対してまだ修正されていないファイルを提案することができない．BugScout [5] は，ML 手法であり，トピックの抽象概念のある一つのレベルのみを用いる．従って，バグ報告とソースコードファイルの２つの空間における項を十分に結び付けられないと考えられる．簡潔にまとめると，語彙的意味の不整合は既存の IR 及び ML 手法において依然として課題であるといえる．

(3)　DNN を用いて語彙不整合の克服を目指す手法

Lam ら [7] は，Zhou らによる先進的な手法である改良ベクトル空間モデル (revised Vector Space Model; rVSM) [3] に加え，バグ報告に対する潜在的な不具合ファイルを推薦するために深層ニューラルネットワーク (Deep Neural Network; DNN) を組み合わせるモデル HyLoc を構築した．rVSM はバグ報告とソースコードファイル間のテキスト類似度を測定するための特徴を抽出する．DNN は，バグ報告と関連する不具合ファイルのペアの中で十分な頻度で現れる場合，バグ報告における項とソースコードファイル内の潜在的に異なるコードトークン及び項を関連付けて学習することにより，それらの間の関連度を測定するのに用いられる．DNN は，例えばパターン認識，自然言語処理 (natural language processing; NLP) のような，入力の特徴に基づく高レベルな識別情報を捉えられる性質を生かして画像及びテキスト処理などの，他の研究領域において用いられ成功を収めている．彼らは，コードの特徴を捉られる可能性を実証的に調査するとともに語彙的意味の不整合を橋渡しできそうかどうかを検討した．

Lam らは，バグ報告とソースコードファイル間の関連性に対する特徴を抽出するために，以下のように DNN を用いた．DNN にはバグ報告 (テキストトークン) と

ソースコードファイル (例えば，識別子，API) ２つの異なる性質を持つ特徴を取り扱うための２つの別々の空間が存在する．低次元の空間に射影された後，射影された特徴は２つの DNN にフィードされる: (1) バグ報告のテキストとコード内のテキストトークン (すなわち，コメント) 間の関係を学習するための DNN; また (2) バグ報告のテキストとコードトークン (すなわち，識別子，API) 間の関係を学習するもう一つの DNN．Lam らは，DNN がバグ報告内のテキスト特徴とソースコードファイル内の関連する特徴の関係を学習できるかを調査することを目的とした．

一般的に，DNN における主な制約としてスケーラビリティがある．この問題では，たくさんの例や特徴が数十万の範囲に及ぶことがある．この問題に対処するために，Lam らは重要な特徴を保持したまま冗長な特徴を除去する特徴空間の次元削減のための DNN に基づく autoencoder [24] を適応した．入力における各空間におけるそれぞれの多くの特徴は低次元の連続の特徴空間に射影される．スケーラビリティを得るために，我々はいくつかの抽象概念において “関連する” 項は NLP [25] に示される射影された空間における (少なくともいくつかの次元にわたって) 近い位置に配置される．

2.2.2 Bugspots, Buglocator 及び compositional VSM を用いる手法

Wang ら [26] はバージョン履歴，類似バグ報告，ソースコードファイルの構造を同時に組み合わせる新たな関連する AmaLgam と呼ばれる不具合ファイル特定手法を提案した．そのために，AmaLgam ではバージョン履歴を解析する Google で用いられるバグ予測技術 Bugspots [11] と，バグ報告システムから類似する報告を解析する不具合ファイル特定手法 Buglocator [3] と，ソースコードファイルの構造を考慮した最高水準の不具合ファイル特定手法 BLUiR [27] を統合する．彼らは，４つのオープンソースプロジェクト AspectJ, Eclipse, SWT 及び ZXing において 3,000 以上のバグに対する不具合ファイル特定の大規模な実験を行った．Sisman と Kak ら [28] によるバグ修正履歴を考慮する不具合ファイル特定手法と比較して，彼らの手法は mean average precision (MAP) において 46.1%の向上を達成した．Buglocator [3] との比較では，彼らの手法は MAP において 24.4%向上した．BLUiR [27] との比較では，MAP において 16.4%向上した．

Wang ら [29] はさらに不具合ファイル特定手法の中で用いられるベクトル空間モ

デル (vector space model; VSM) の未解決の課題に注目し，解決を試みている．標準的な tf-idf 重み付けスキーマ (VSMnatural と命名される) によるベクトル空間モデル (VSM) は 9 つの他の最高水準の IR 手法 (例えば, LDA, LSA, など) を上回る性能であることが示されてきた．しかし，異なる重み付けスキーマによる複数の VSM が存在し，それらの相対的な性能は異なるソフトウェアシステムによって異なる．この観察をもとに，彼らは様々に異なる異種の VSM をまとめ，最適化問題としてそれらの組み合わせをモデル化した．彼らは，可能な組み合わせによる空間を探索し，ヒューリスティックな近似最適な組み合わせモデルを出力する遺伝アルゴリズム (genetic algorithm; GA) に基づく手法を提案した．彼らは AspectJ, Eclipse, 及び SWT の数千のバグ報告において複数のベースラインと彼らの手法を比較して評価した．平均して，彼らの手法 (VSMcomposite と名付けられる) は 5 位以内での正答率 (Hit@5)，mean average precision (MAP)，及び mean reciprocal rank (MRR) において VSMnatural の結果からそれぞれ 18.4%, 20.6%, 及び 10.5%向上した．また，彼らは彼らの組み合わせモデルを前述の彼らが以前提案した最高水準の不具合ファイル特定手法 AmaLgam [26] と統合した．最終的なモデル AmaLgamcomposite は平均して Hit@5，MAP，及び MRR において AmaLgam からそれぞれ 8.0%, 14.4% 及び 6.5%向上した．

本研究においても，ヒューリスティックバグ予測ツール Bugspots [11]，及び不具合ファイル予測手法 Buglocator [3] を組み合わせて用いることにより，より良い性能の達成を目指すという共通の戦略を用いている．また，彼らの手法と同様に，より効果的と考えられる VSM を用いる．そのため，我々の提案手法の性能を上記の手法の性能と比較することが本来は望ましい．ただし，現時点では適用しているデータセットが異なるため直接的な比較が不可能である．また，この点について指摘を受けた時点から調査しているが，期間が短く十分な考察ができていないため，今後調査が必要な課題として本内容を記述するにとどめる．

2.2.3 Program Spectra を組み合わせる手法

IR に基づく手法がバグ報告の中の情報を用いるのに対し，spectrum に基づく手法は program spectra（すなわち，どのプログラム要素が各テストケースに対して実行されるかの記録）を用いる．両者は最終的にバグを含むと考えられるプログラム要

素の順位付けリストを生成する．しかし，これらの技術はバグ報告または program spectra のそれぞれ片方の情報しか考慮しておらず，最適化されていない．

この問題に対処するために，Le ら [30] は，不具合ファイル特定のためのバグ報告と program spectra の両者を考慮する新たなマルチモーダル手法を提案した．彼らの手法では，適応的に特別なバグを可能な場所にマップするバグに特化したモデルを作成し，バグに高く関連する疑わしい単語を考慮する新しいアイデアを導入した．彼らは 4 つのソフトウェアシステムから得られた 157 の実際のバグについて手法を評価し，最高水準の IR に基づく不具合ファイル特定手法，最高水準の spectrum に基づく不具合ファイル特定手法，及び 3 つの最高水準のマルチモーダルな不具合ファイル特定に適用される特徴の位置特定手法と比較した．実験結果として，彼らの手法は正しく特定された不具合ファイルの個数について，開発者が 1, 5, 及び 10 のプログラム要素 (すなわち，トップ 1, トップ 5, 及び トップ ‘0) を調べる時，及び MAP に置いてそれぞれ，ベースラインに比べ少なくとも 47.62%, 31.48%, 27.78%, 及び 28.80%上回った．

Program Spectra を組み合わせる最新の本手法についても，この論文を引用するべきであるという指摘を受けた時点から調査しているが，期間が短く十分に本研究との内容における関係の調査ができていないため，彼らの報告を紹介するにとどめる．

2.3 単語分散表現

本節では単語分散表現 (Distributed representations of words) について説明する．

2.3.1 単語分散表現の効率的な推定

Mikolov ら [31] は，非常に大きなデータセットのための連続ベクトル表現を計算するための 2 つの新しいモデル構成を提案している．これらの表現の質は単語の類似タスクによって計測され，結果は以前の異なるニューラルネットワークに基づく最も優秀な手法と比較された．彼らは，非常に少ない計算機資源で精度における向上を観察した，すなわち，1 日未満で 16 億の単語データセットから質の高い単語ベクトルを学習した．さらに，彼らは文法的及び意味的な単語の類似性のための彼らのデータセットにおいて最高水準の性能を与えられることを示した．

(1) ニューラルネットワーク言語モデル

単語の連続表現を推定するために，2.1.3節でも触れたよく知られたLatent Semantic Analysis (LSA) [19] やLatent Dirichlet Allocation (LDA) [20] を含む多くの異なる種類のモデルが提案されてきた．Mikolovら [31] の論文では，彼らはニューラルネットワークによって学習される単語の分散表現に注目し，以前にそれらが単語間の線形規則性を保持することにおいてLSAより非常に優れた性能を発揮することを紹介している [32,33]．また，LDAは大きなデータセットにおいて計算量が非常に大きくなる．彼らは計算量の複雑性をモデル化し，彼らのモデルが計算量において非常に優れていることを確かめた．

Feedforward Neural Net Language Model (NNLM)

The probabilistic feedforward neural network language modelは [34] で提案された．モデルは入力，射影，隠れ及び出力の各レイヤーからなる．入力レイヤーでは，Nの以前の単語が1-of-Vのコーディングによってエンコードされ，Vは語彙のサイズである．その後，入力レイヤーは共有される射影行列を用いて，N × D次元の射影レイヤーPに射影される．限られたNの入力がどの与えられた時間においてもアクティブであり，射影レイヤーの構成は比較的あまり最適化されていない動作である．

Recurrent Neural Net Language Model (RNNLM)

Recurrent neural network based language modelはfeedforward NNLMのある程度の制約，内容の長さ (すなわち，モデルのオーダーN) の特定する必要性など，を克服するために提案されてきており，それは理論的にはRNNsがshallow neural networksと比べ複雑なパターンをより効率的に表現できるためである [8,35]．RNNモデルは射影レイヤーを持たない; 入力，隠れ及び出力レイヤーのみである．この種類のモデルにおいて特別なのは，隠れレイヤーがそれ自身に対して接続しているrecurrent matrixであり，時間が遅れての接続を用いている．このことはrecurrent modelがある種の短期記憶を形成する．ここでの過去からの情報は，現在の入力に基づき更新される隠れレイヤーの状態，及び，過去のタイムステップにおける隠れレイヤーの状態，によって表現される．

(2) 対数線形モデル

本節では，彼らが提案する計算の複雑性を最小化するための単語分散表現の学習のための２つの新たなモデル構成を紹介する．前節からの主な観察は，複雑性の最たるものはモデルの非線形隠れレイヤーによって引き起こされることである．これはニューラルネットワークをより魅力的にすることであるが，彼らはニューラルネットワークと同等の正確さでデータを表現することができないかもしれないが，データをより効率的に学習される，よりシンプルなモデルを探索することを決断した．新たな構成は彼らの先行研究 [36,37] に直接基づくものであり，ニューラルネットワーク言語モデルが次の２つのステップで学習に成功したことを発見した：最初，連続単語ベクトルはシンプルなモデルによって学習され，その後，N-gram NNLM がこれらの単語の分散表現のトップに基づき学習される．単語ベクトルの学習に注目するその後のかなりの量の研究がある一方で，彼らは [36] で提案されたシンプルなモデルを用いた．関連するモデルはかなり前の [38,39] においても提案されてきている．

Continuous Bag-of-Words Model

最初に提案される構成は feedforward NNLM に似ており，非線形隠れレイヤーが取り除かれ射影レイヤーが全ての単語間で (ただの射影行列にとどまらず) 共有されるというものである．従って，全ての単語は同じ位置に射影される (それらのベクトルが平均される)．彼らは，過去の単語の順序が射影に影響しないので，この構成を bag-of-words model と呼んでいる．さらに，彼らは未来からの単語を用いている．彼らは，現在の（中間の）単語を正しく分類する訓練基準を用いて，４つの未来と４つの過去を入力として，対数線形分類器を構築することにより，次の節で紹介されるタスクを用いて最高の性能を得た．彼らはこのモデルを，continuous bag of words model(CBOW) と名付け，標準的な bag-of-words model と異なり，コンテキストの連続分散表現に用いる．モデル構成は図 2.1 に示される．入力と射影レイヤー間の重み付け行列は NNLM と同様に全ての単語間で共有される．

Continuous Skip-gram Model

第二の構成は CBOW に似ているが，コンテキストに基づき現在の単語を予測

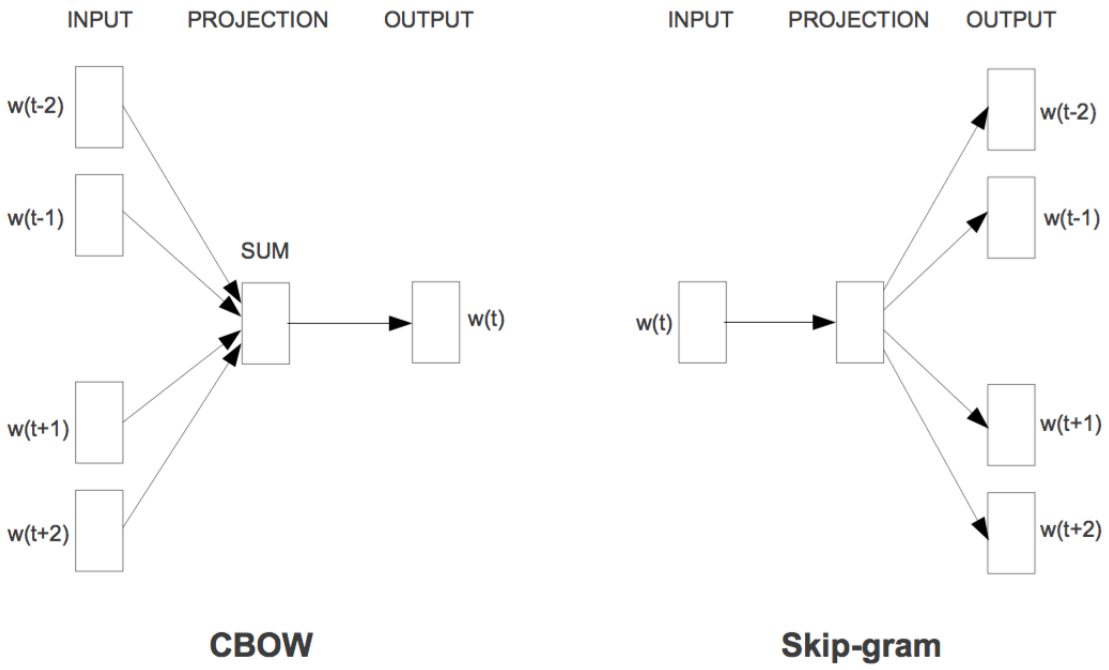

図 2.1　新たなモデル構成．CBOW 構成はコンテキストに基づき現在の単語を予想し，Skip-gram は現在の単語から周囲の単語を予測する．([31] より引用)

する代わりに，同じ文における他の単語に基づき現在の単語を分類する能力を最大化するものである．より正確には，彼らはそれぞれの現在の単語を連続射影レイヤーを持つ対数線形識別器への入力として用い，現在の単語の以前及び以後のある程度の範囲に含まれる単語群を予測する．彼らは，結果として得られる単語ベクトルの質の向上における範囲の増加を確かめたが，計算の複雑性も増大した．コンテキストにおけるより距離の離れた単語はしばしばより近い単語よりも現在の単語にあまり関係しないので，彼らは彼らの学習例においてそれらの単語からより少ないサンプリングをすることによりより離れた単語に対して，より少ない重みを与える．

2.3.2 単語分散表現の構成性

Mikolov らはさらに [10] の中で，以下のように単語分散表現の構成性について述べた．単語表現は個々の単語の組み合わせではない慣用的なフレーズを表現することができないことに制約がある．例えば，“Boston Globe” は新聞であり，これは “Boston” と “Globe” の意味の単純な組み合わせでなはい．従って，全てのフレーズを表現するためにベクトルを用いることは Skip-gram model をより有意なものにする．単語ベクトルの組み合わせにより文の意味を表現することを目的とした recursive autoencoder [40] のような他の手法があり，単語ベクトルの代わりにフレーズベクトルを使うことによる利点を享受している．彼らの単語ベースからフレーズベースへの拡張は比較的シンプルである．

Mikolov らは [10] の中で Skip-gram model によって学習される単語とフレーズの表現が単純なベクトル数式を用いて正確な類推思考を振る舞うことができる線形構造を発揮することを紹介した．面白いことに，彼らは Skip-gram 表現が彼らのベクトル表現の要素ワイズの足し算によって意味上の単語の組み合わせを可能にするその他の種類の線形構造を発揮することを確かめた．この現象は表 2.2 に示される．ベクトルの加法性は訓練対象の調査によって説明されうる．単語ベクトルは softmax 非線形性への入力を持つ線形の関係性にある．これらの値は出力レイヤーによって計算される確率に対数的に関係するので，二つの単語ベクトルの和は 2 つのコンテキストの分布の生成物に関係している．生成物はここでは AND 関数のように振る舞う．つまり，両方のベクトルによって高い確率で割り当てられた単語は高い確

率となり，その他の単語は依然低い確率となる．従って，もし “Volga River” が単語 “Russian” と “river” と一緒に同じ文において頻出する場合，これら 2 つの単語ベクトルの和は “Volga River” のベクトルに近い特徴ベクトルのような結果となる．

この構成性は言語理解における非明確な段階が単語分散表現における基本的な数学的演算を用いて得られることを示唆している．

2.3.3 単語分散表現ツール word2vec

Milcov らによる単語分散表現に関する研究 [10, 31, 32, 41]，は Google において word2vec [9] と呼ばれるツールとして実装された形式で提供され，ここ数年で自然言語処理や関連分野に関心を持つ多くの研究者の注目を集めてきた．以下では word2vec について説明する．

本ツールは単語のベクトル表現 (単語の分散表現) を計算するための continuous bag-of-words 及び skip-gram アーキテクチャの効果的な解釈を提供する．これらの表現は，多くの自然言語処理での応用あるいはさらなる研究において，後に続く研究に用いられることが想定されている．

word2vec ツールはテキストコーパスを入力として用い，単語ベクトルを出力として生成する．最初訓練用のテキストデータから語彙集を構築し，その後単語のベクトル表現を学習する．結果として得られる単語ベクトルは多くの自然言語処理及び機械学習における特徴として用いられることが可能である，学習された表現を調査する単純な方法は，ユーザが決定した単語に対する最も近接する単語を発見することである．例えば，あなたが ‘france’ と入力すると，機能 “distance” は最も似た単語とそれらの ‘france’ とのコサイン類似度を表 3.3 のように表示する．

機能 “distance” に対して，複数の単語を入力することも可能である．その場合，複数の単語の入力によって生成されるベクトルは 2.3.2 節で紹介したように，複数の単語のベクトル表現の和として表され，それらのフレーズが表す暗黙の意味を表現する [10]．出力としては，フレーズを表現するベクトルに近い単語ベクトルを持つ単語が，距離が近い順にコサイン類似度とともに表示される．

word2vec には 2 つの主要な学習アルゴリズムが実装されている．それらは，continuous bag-of-words と continuous skip-gram である．切り替えオプション -cbow を用いて，ユーザはそれら双方の学習アルゴリズムの内どちらかを選択することが

表 2.2　要素ワイズの足しあわせを用いたベクトルの構成性．最も良い Skip-gram モデルを用いて２つのベクトルの和に最も近い４つのトークン ([10] より引用)

Czech+currency	Veietnam+capital	German+airlines	Russian+river	French+actress
koruna	Hanoi	airline Lufthansa	Moscow	Juliette Binoche
Check crown	Ho Chi Minh City	carrier Lufthansa	Volga River	Vanessa Paradis
Polish zolty	Viet Nam	flag carrier Lufthansa	upriver	Charlotte Gainsbourg
CTK	Vietnamese	Lufthansa	Russia	Cecile De

表 2.3　単語とコサイン類似度 ([9] より引用)

順位	単語	コサイン類似度
1	spain	0.678515
2	belgium	0.665923
3	netherlands	0.652428
4	italy	0.633130
5	switzerland	0.622323
6	luxembourg	0.610033
7	portugal	0.577154
8	russia	0.571507
9	germany	0.563291
10	catalonia	0.534176
...	...	...

できる．両方のアルゴリズムは，文中の他の単語を予測ために有益な単語表現を学習する．これらのアルゴリズムの詳細は前述の [10,31] に示されている．

2.3.4　不具合ファイル特定への応用

本研究では，2.3.3 節で紹介した単語分散表現ツール word2vec を拡張して用いることにより，バグ報告からの不具合ファイル特定に応用する．より具体的には，“学習されるベクトル表現が構成するベクトル空間” を，“情報検索モデルを用いた不具合ファイル特定手法におけるベクトル空間モデル” とみなして応用する．以下では，その基本的なアイデア及びアイデアの長所を説明する．

(1)　基本的なアイデア

第一に，単語分散表現ツール word2vec により生成されるベクトル空間を用いて，まず，単語を学習する際の word2vec への入力として，全てのバグ報告及びソースコードファイルの単語群をテキストコーパスとして用いる．これにより，全てのバグ報告及びソースコードファイルに含まれる全ての単語は一つのベクトル空間における意味を表すベクトル表現として学習される．本研究ではこのことにより，バグ報告及びソースコードファイル内で用いられる項における語彙の不整合の解消を試みる．この点が，本研究において本アイデアを提案する目的である．

次に，word2vec の機能 “distance” を拡張し，(1) 入力として “バグ報告の単語群” のベクトル表現を，(2) 出力において順位付けされる対象のベクトル表現として “ソースコードファイルの単語群” を用いる．バグ報告及びソースコードファイルの単語群から生成されるベクトルは 2.3.2 節で紹介したように，複数の単語のベクトル表現の和として表され，それらのフレーズが表す暗黙の意味を表現する [10]．そのため，上記のように拡張した機能 “distance” の出力結果からは，多くの情報検索モデルで見られるようなファイル及びバグ報告との関連度を降順に提示する順位付けリストが得られると考えられる．表 2.4 は 2.1.1 節で示した Tomcat におけるバグ報告 (ID: 31257) の内容 (JSP Compiler) に関連するファイルとコサイン類似度(注 2) である．バグ報告内に頻出した JSP Compiler に関連するファイルがリスト内の高い順位に順位

(注 2)：ここでのバグ報告に関連する不具合ファイルのリスト作成には，後に 3 章で説明する手法の内 word2vec による関連度のみを用いた場合を実行した．

表 2.4　バグ報告 (ID: 31257) の内容 (JSP Compiler) に関連するファイルとコサイン類似度

順位	候補ファイル名	コサイン類似度
1	tomcat/java/org/apache/tomcat/jni/Stdlib.java	0.586275
2	tomcat/java/org/apache/tomcat/jni/OS.java	0.551435
3	tomcat/java/org/apache/jasper/compiler/JDTCompiler.java	0.499763
4	**tomcat/java/org/apache/jasper/compiler/AntCompiler.java**	**0.446665**
5	tomcat/java/org/apache/jasper/tagplugins/jstl/core/When.java	0.446448
6	tomcat/java/org/apache/tomcat/jni/Buffer.java	0.438573
7	tomcat/test/org/apache/tomcat/util/net/TestXxxEndpoint.java	0.426327
8	tomcat/java/org/apache/jasper/compiler/Compiler.java	0.423611
9	tomcat/java/org/apache/tomcat/jni/File.java	0.420008
10	tomcat/java/org/apache/jasper/JspC.java	0.418780
...	...	...

付けされていることがわかる．それらのファイルの内，第 4 位に本バグ報告に関連するファイル AntCompiler.java が順位付けされている．

第二に，我々は多くの試行により本モデルのみでの不具合ファイル特定の精度が十分ではないことを経験的に学んだ．また，不具合ファイル特定における多くの先行手法 [] では情報検索モデルを単独で用いるだけでなくその他の手法を組み合わせて用いることにより，精度の向上を実現してきている．それらの手法では，API 情報も合わせて用いられている．これらのことを踏まえ，上記で提案する `word2vec` による情報検索モデルを単独で用いず,

1. ヒューリスティックバグ予測ツール Bugspots [11],
2. 従来のテキスト類似度及びバグ報告の類似性を用いた最高水準の不具合ファイル予測手法 Buglocator [3]，そして，
3. ソースコードファイルのディレクトリ構造の情報による局所焦点化の手法 (API 情報に含まれる一部の情報)

を組み合わせることによりさらなる精度向上を試みる．

(2) アイデアの長所

続いて，Lam ら [7] による手法と比較しながら，本アイデアの長所を紹介する．Lam ら [7] は，語彙の不整合の問題を解決するために，バグ報告とソースコードファイルの関連性を学習する *DNN (Deep Neural Network)* を用いて問題に対処している．

彼らがベクトル表現の学習に用いる DNN は，基本的に Deep Neural Network Language Model(DNNLM) [25] に基づいている．ただし，彼らも論文内で報告しているように，DNN の主な制約としてスケーラビリティがあり，その対処のために彼らは autoencoder [24] を用いている．より詳しく述べると，最初にバグ報告及びソースコードファイルの特徴から DNNLM に基づいて学習した DNN を，autoencoder を用いて関連する autoencoder-DNN へと次元削減することにより対処している．従って，彼らの DNN 過程は高スケーラビリティあるいは高速であると言い切れない．

これに対し，我々の手法は元々DNNLM を用いることなく，計算量において有利なシンプルな単語ベクトルの学習モデルを用いる．学習モデルでは，コンテキストと共に学習を行うことにより初めから低次元で高密度のベクトルとして単語の特徴ベ

クトルを学習する．ただし，ニューラルネットワークと同等の正確さでデータを表現することができない点が制約として存在する点には注意が必要である．従って，類似のベクトル空間モデルを構築し，同程度の精度を確かめられた場合，スケーラビリティ及び速度の点で比較的有利となることが考えられる．

(3) 本節のまとめ

本節で紹介した我々のアイデアによる主要な長所は以下の２点にまとめられる．

第一点目 我々のアイデアでは高次元の射影や深層ニューラルネットワークが不要な点である．そのため，提案手法ではスケーラビリティ及び速度の点で比較的有利となることが考えられる．

第二点目 我々のアイデアでは，多くのベクトル空間モデルにおいて見られるように，各項が**単語ベクトル**と呼ばれる分散表現として表現されるだけでなく，複数の単語のベクトル表現の和として表され，それらのフレーズが表す暗黙の意味を表現することが可能であること [10] が挙げられる．

本章での説明を踏まえ，次章では `word2vec` をバグ報告からの不具合ファイル特定に応用する手法，**意味ベクトル空間モデル** *(semantic Vector Space Model; sVSM)* を提案する．また，精度向上のための**組み合わせ不具合ファイル特定** *(Combined Bug Localization; CombBL)* と呼ぶ新たな手法を紹介する．

3. 提案手法

3.1 モデル構成

我々は組み合わせモデル *CombBL* を提案する．図 3.1 は CombBL の不具合ファイル特定のアーキテクチャを示している．アーキテクチャは以下の 4 つの部分のスコア要素から構成される．

1. `word2vec` [9] による関連度スコア.
2. BugLocator [3] によるテキスト類似度スコア.
3. Bugspots [11] によるバグ修正履歴スコア.
4. バグ報告とともに報告されるコンポーネント情報によるコンポーネント局所焦点化スコア.

我々は上記 4 つのスコアを組み合わせ特定のバグ報告に対するバグを含みそうなファイルを順位付けする．我々はスコアを計算する各方法の内部処理を以下に記す.

3.2 `word2vec` に基づく関連度

我々はバグ報告とソースコードファイルの関連度を計算するために `word2vec` [9] を用いる．既存研究 [7] が DNN に基づく auto-encoder [24] を用いているのに対し，我々はバグ報告とソースコードファイル内の特徴を `word2vec` を用いてベクトル空間に射影する．`word2vec` は単語の分散ベクトル表現の実装であり，単語ベクトル学習のための Continuous Bag of Words Model [31] と呼ばれるモデルに基づいている．

図 3.2 は `word2vec` を用いたバグ報告とソースコードファイル間の特徴空間の次元削減と関連度計算を示している．最初にまず，我々はバグ報告とソースコードファイルから特徴を抽出しコーパス W を作成する．コーパス W を用いて，我々はベクトルとして語彙の中の単語を表現し，単語ベクトルの組み合わせによる一連の単語を表現する．次に，我々は単語ベクトルの組み合わせによってバグ報告及びソースコードファイルを BR ベクトル及び FL ベクトルとして表す．最後に，我々は関連する BR ベクトルと FL ベクトル間の**コサイン類似度**によって定義されるバグ報告とソースコードファイル間の関連度スコアを計算する．我々は以下に各 3 つのステッ

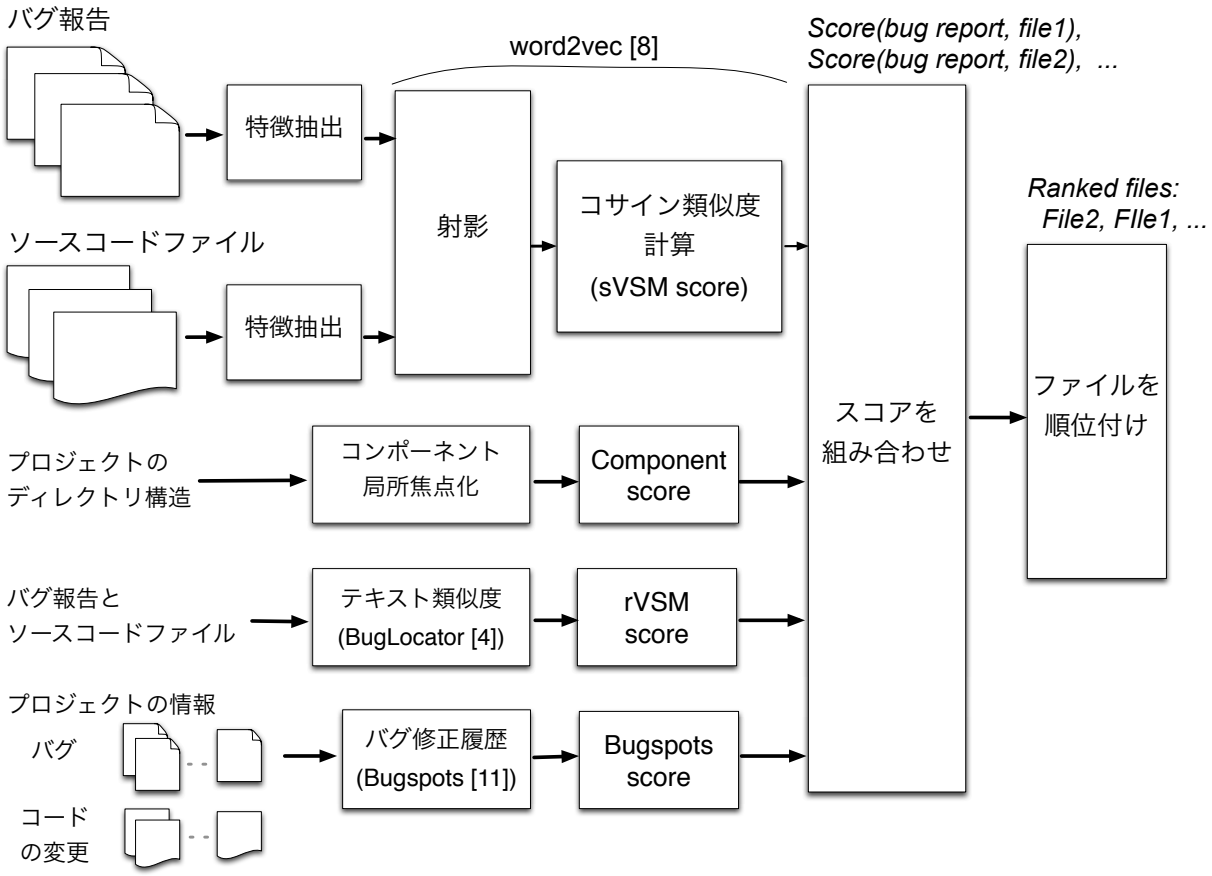

図 3.1　不具合ファイル特定モデル CombBL

プを細かく説明する．

3.2.1 単語の抽出

本節では，バグ報告及びソースコードファイルからの単語の抽出について説明する．一般に，バグ報告に対応するクエリ及びソースコードに対応する文書には同じ前処理が施されるべきであると考えられる。本研究では，以下の (1), (2) で説明するような手順を行っているが，基本的には両方とも porter stemming algorithm [14] を用いている点は同じである．ただし，複数の点において異なる処理を施している点については注意が必要であることを記しておく．

(1) ソースコードファイルからの単語抽出

我々は lscp [42] と呼ばれる語彙分析ツールを用いてソースコードファイルから識別子を抽出する．lscp は軽量ソースコードプリプロセッサーであり，ソースコードファイルから識別子，コメント，文字列リテラルを抽出できる．本稿では，我々は perl スクリプト上で，以下の lscp コマンドを用いてソースコードファイルから識別子のみを抽出した．

```
use lscp;
my $lscp = lscp->new;
$lscp->setOption("logLevel", "error"); #lscp 実行時の冗長性の設定
$lscp->setOption("inPath", $OPTS->input_dir); #入力ファイルのディレクトリ設定
$lscp->setOption("outPath",$OPTS->output_dir); #出力ファイルのディレクトリ設定
$lscp->setOption("isCode", 1); #入力はソースコードファイルである
$lscp->setOption("doIdentifiers", 1); #isCode = 1 の時，識別子を抽出する
$lscp->setOption("doStringLiterals",0); #isCode = 1 の時，文字列リテラルを抽出しない
$lscp->setOption("doLowerCase",1); #小文字化を行う
$lscp->setOption("doRemovePunctuation",1); #ピリオド，コンマ，疑問符などの句読点を除去
$lscp->setOption("doTokenize", 1); #合成語分割 (camelCase,under_scores,dot.notation 等)
$lscp->setOption("doStemming", 0); #語幹抽出を行わない
$lscp->setOption("doComments", 0); #コメントを抽出しない
$lscp->setOption("doStopwordsEnglish", 1); #英語のストップワードを除去
$lscp->setOption("doStopwordsKeywords", 1); #キーワードのストップワードを除去
$lscp->setOption("doRemoveDigits", 1); #[0-9] からなる数字を除去
$lscp->preprocess();
```

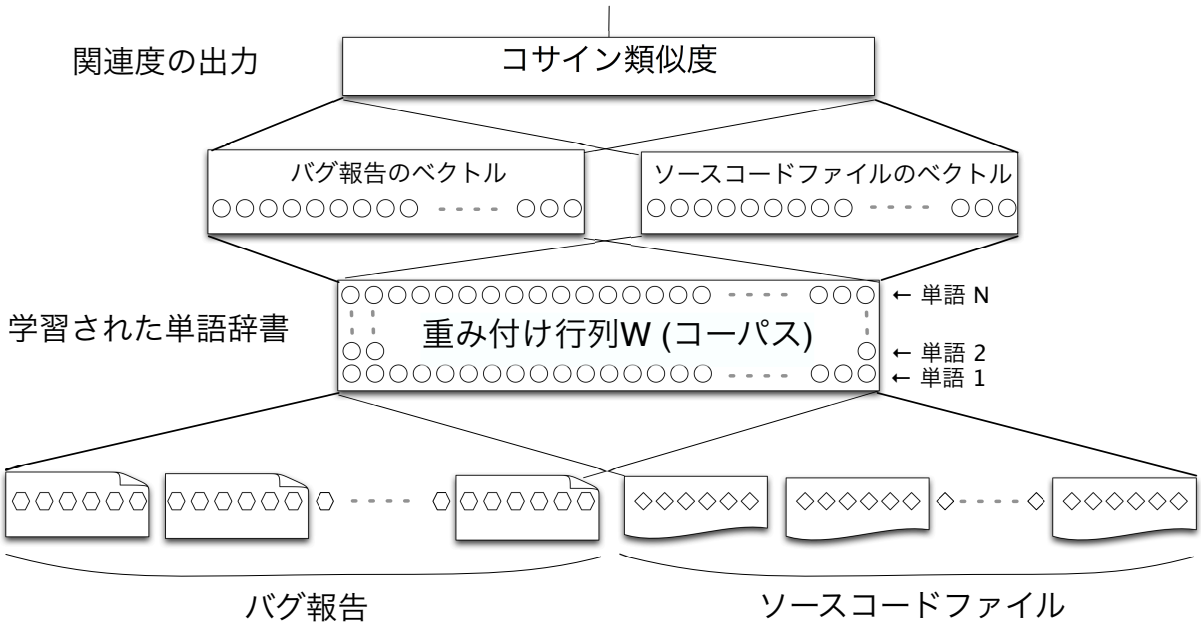

図 3.2　射影のための重み付け行列 W とコサイン類似度計算

尚，lscp のウェブサイト [42] が用いている perl モジュール Linga::Stem の関連モジュール Lingua::Stem::En [43] のウェブサイトによると，語幹抽出のアルゴリズムとして Porter Stemming Algorithm [14] が用いられている．

上記の `lscp` の設定により，文字列リテラル及びコメントを抽出せず，識別子を抽出した．これは，識別子がソースコードの機能及び動作を表現する特徴として，重要な情報を含んでいると考えるためである．また，文字列リテラル及びコメントは，文書として検索するにあたり，文書を表現するための特徴として余計な情報を含んでいると考えたため取り除いた．小文字化は，単語を学習する際に，大文字か小文字化の違いによって異なる単語に区別されるのを防ぐために適用した．さらに，`word2vec` の単語単位のベクトル表現による意味の足しあわせの効果を利用するため，合成語を構成後に分割した．また，語幹抽出は行っていない．これは

1. ソースコード上の識別子では語形変化させることが少ない，また，
2. 識別子の語形を保存しながら抽出すれば検索の際に見落とされることを防げる

と考えるためである．ストップワード及び数字は情報検索において，検索対象の個々の文書内の特徴を表すための重要な情報ではないので，取り除いた．

(2) バグ報告からの単語抽出

また，我々は MySQL 上で Ye のデータセット [6] に含まれる，SQL データベースにおける `bag_of_word_stemmed` 領域を用いて，バグ報告に対する語幹抽出処理済みの単語群を得た．`bag_of_word_stemmed` 領域には，bug summary と description の単語群が含まれている．

```
use tomcat;
set group_concat_max_len = 10000000;
SELECT  bug_and_files.bug_id, comp.comp,
        GROUP_CONCAT(bug_and_files.files SEPARATOR '\\\\\n'),
        bug_and_files.bag_of_word_stemmed
FROM  bug_and_files LEFT OUTER JOIN comp ON bug_and_files.bug_id = comp.bug_id
GROUP BY bug_and_files.bug_id
INTO OUTFILE "/tmp/selected_bugs_tom.txt";
FIELDS TERMINATED BY '\\\\\\\\n'
```

表 3.1　Tomcat における AntCompiler.java のソースコードファイルの内容の一部

```
...
public class AntCompiler extends Compiler {

private final Log log = LogFactory.getLog(AntCompiler.class); // must not be static

protected static final Object javacLock = new Object();

static {
System.setErr(new SystemLogHandler(System.err));
}

// ----------------------------------------- Instance Variables

protected Project project = null;
protected JasperAntLogger logger;

// ----------------------------------------------- Constructor

// Lazy eval - if we don't need to compile we probably don't need the project
protected Project getProject() {

if (project != null)
return project;

// Initializing project
project = new Project();
logger = new JasperAntLogger();
logger.setOutputPrintStream(System.out);
logger.setErrorPrintStream(System.err);
logger.setMessageOutputLevel(Project.MSG_INFO);
project.addBuildListener( logger);
if (System.getProperty(Constants.CATALINA_HOME_PROP) != null) {
project.setBasedir(System.getProperty(Constants.CATALINA_HOME_PROP));
}
...
```

表 3.2　Tomcat における AntCompiler.java のソースコードファイルにより得られた単語群

tomcat/java/org/apache/jasper/compiler/AntCompiler.java
add ant apache append arg argument array baos basedir buf build builder capture catalina check classpath close compilation compile compiled compiler constants cp create ctxt current data debug detail dir dirs dispatcher djava element elements enabled encoding endorsed entry env err error errors exception execute ext extdirs extension exts factory file find flush fork found generate generated handler home implementation includes index info init install io jasper java javac jsp juli len length level line list listener local localizer location lock log logger logging message millis mode ms msg newline nodes obj object optimize options org output override page parse path pattern print println priority project prop property prototype ps quote report repository result scratch sep separator servlet set smap source specific src srcdir st stream streams suppressed system target task taskdefs tempdir thread time token tokenizer tokens tools types unset util vm work wrapped

```
OPTIONALLY ENCLOSED BY '';
```

今回利用した bag_of_word_stemmed 領域を含むデータベースを提供している Ye ら [6] は，論文内でバグ報告からの特徴の抽出について以下のように記している．バグ報告に対して，VSM の表現を作成するために bug summary 及び description を用いる．入力文書をトークン化するために，彼らはテキストを空白を用いて bag of words へと分割した．その後，句読点，数字列，標準的な接続詞や限定詞のような IR ストップワードを除去した．“WorkBench” のような合成語は大文字基づいてそれらの構成後に分割された．文書の bag of words 表現は，元々のトークンを含みながら，結果のトークン，すなわち，本例では “Work” と “Bench” も追加される．最終的に，全ての単語は文献 [44] に基づく NLTK のパッケージ (注 1)(GitHub では [45]) 内で実装される Porter stemmer [14] を用いて，それらの語幹へと変換された．

3.2.2 コーパス作成

まず，我々は単純にソースコードから抽出された識別子とバグ報告から抽出された語幹抽出済みの単語群を連結する．この際の具体的な手順は以下である．

1. 各ソースコードファイルから (1) 節の手順で抽出された単語群を含む各テキストファイルを，全てのファイルに対して抽出の際にディレクトリを走査した順に上から下へ連結していき，一つのテキストファイルとして結合する．
2. 各バグ報告から (2) 節の手順で抽出された単語群を含む各テキストファイルの内，bug summary 及び description に対応する部分のみを用いる．該当部分を，全てのバグ報告に対してバグ ID の小さい方から順に上から下へ連結し，一つのテキストファイルとして結合する．
3. 上記の 1. のテキストファイルを上側に 2. のテキストファイルを下側に配置し，一つのテキストファイルとして結合する．最終的にできたこのテキストファイルを word2vec に対する入力 (“input.txt”) として用いる．

次に，それらを以下のコマンドオプションを用いて word2vec へ入力する:

(注 1)：http://www.nltk.org/api/nltk.stem.html

表 3.3　Tomcat におけるバグ報告 (Bug ID: 31257) に対するデータベースより得られた単語群

31257\\\
Jasper\\\
java/org/apache/jasper/compiler/AntCompiler.java\\\
java endors dir use jsp compil fork jsp compil fork javac process check see
current endors directori check system properti java endors dir otherwis class
miss jsp page properli compil experienc problem explicitli set java endors dir
properti command line jsp compil use javac fork patch apach jasper compil compil
support src origin jakarta tomcat jasper jakartatomcatjasp jasper2 src share apach
jasper compil compil java jun src apach jasper compil compil java thu sep extdir
set path setpath ext javac set extdir setextdir extdir info append extens ext add
endors directori string endors system get properti getproperti java endors dir endors
null javac implement specif argument implementationspecificargu endors arg
endorsedarg javac creat compil arg createcompilerarg endors arg endorsedarg set
line setlin djava endors dir endors info append endors endors els info append endors
dir specifi configur compil object javac set encod setencod java encod javaencod
javac set classpath setclasspath path javac set debug setdebug ctxt get option getopt
get class debug info getclassdebuginfo true probabl check ctxt get option getopt get
fork getfork well sinc alway fork doesn matter

```
./word2vec -train input.txt -output W.bin -size 200 -binary 1 -cbow 1
 -min-count 0 -threads 4
```

出力はコーパス W と名付ける．我々は単一のコーパス W から BR ベクトル及び FL ベクトルを生成できる．それぞれのオプションの意味を以下に示す．

-train input.txt 入力ファイルを “input.txt” とする．

-output W.bin 出力ファイルを “W.bin” とする．

-size 200 ベクトルの次元数を 200 次元とする..

-binary 1 出力はバイナリ形式で行う．

-cbow 1 学習のモデルとして continuous bag-of-words model を使用する．

-min-count 0 学習するために必要な単語の最低出現頻度を 0 回に設定し，出現頻度の低い単語の情報を失わず学習するように設定する．すなわち，出現した全ての単語を設定するようにする．

-threads 4 学習を 4 スレッド並列で行う．

なお，Mikolov ら [10, 31, 32] によって単語ベクトルにより単語の意味を表現する性能が高いとされている continuous skip-gram model ではなく，その前段階として説明された continuous bag-of-words model を用いる理由を説明する．本研究の実験を行う以前に試行を行い，continuous bag-of-words model を用いた場合と continuous skip-gram model を用いた場合における不具合ファイル特定の性能を比較した．その際，continuous bag-of-words model の方が性能が高い傾向が見られたため，continuous bag-of-words model を用いている．なお，Mikolov らによって報告された，

1. 学習における計算効率の高さ．
2. 単語ベクトルにより単語の意味を表現する性能．また，単語ベクトルの和によりフレーズの慣用的な意味を表現する性能．

は共通していると考えられる．ただし，continuous skip-gram model の方がよりも continuous bag-of-words model に比べ 2. に関する性能が高いと報告されているため，この点は注意が必要である．

3.2.3 関連性計算

ソースコードファイルとバグ報告間の関連度の評価値はそれらに対応するベクトル表現を用いて計算できる．この特徴を用いて，我々は連続するベクトル空間においてバグ報告に対する関連ソースコードファイルを検索する．ファイルは関連度値によって順位付けされた後に，出力として返される．この観点から，我々はこの手順を，バグ報告に基づいてソースコードファイルを検索するモデルを作成する情報検索手法と見なす．我々はソースコードファイルを索引し順位付けする意味ベクトル空間モデル (semantic Vector Space Model; sVSM) を提案する．sVSM では，我々はソースコードをテキストコーパス，バグ報告を検索クエリと見なす．この時，クエリ q と文書 d 間の関連度はそれらに対応するベクトル表現間のコサイン類似度によって次式のように計算される．

$$Relevancy(q,d) = \cos(q,d) = \frac{\overrightarrow{V_q} \cdot \overrightarrow{V_d}}{|\overrightarrow{V_q}||\overrightarrow{V_d}|}, \tag{3.1}$$

ここで，$\overrightarrow{V_q}$ と $\overrightarrow{V_d}$ はクエリ q と文書 d を表す代表ベクトルである．$\overrightarrow{V_q} \cdot \overrightarrow{V_d}$ は 2 つのベクトルの内積を表す．我々は，クエリ q と文書 d を表す代表ベクトルを，クエリ q と文書 d に含まれる単語ベクトルの平均ベクトルとして，それぞれ定義する．クエリ q と文書 d を表すベクトル表現は以下の式で計算される:

$$\overrightarrow{V_q} = \frac{1}{N_q}\sum_{i=1}^{N_q} \overrightarrow{v_q}, \quad \overrightarrow{V_d} = \frac{1}{N_d}\sum_{j=1}^{N_d} \overrightarrow{v_d} \tag{3.2}$$

ここで，$\overrightarrow{v_q}$ と $\overrightarrow{v_d}$ はクエリ q と文書 d の単語ベクトルである．N_q と N_d は，クエリ q と文書 d 内の総単語数を表している．

我々は，sVSM のための `word2vec` スコア評価アルゴリズムを以下の式で提案する:

$$sVSMScore = word2vecScore = \cos(q,d) \tag{3.3}$$

バグ報告が与えられると，我々は式 (3.3) を用いてバグ報告とソースコードファイル間の関連度スコア (*sVSMScore*) を決定する．我々はこのスコアを基準に順位付けリスト (*sVSMrank*) を得る．リスト内では，高いスコアであるほど高順位となる．表 3.4 は，表 2.1 で示した Tomcat のバグ報告 (ID: 32157) に対する，*sVSMScore* のみによる順位付けリストを示している．バグ報告に対する修正済みファイル AntCompiler.java は，本リスト内において 4 位に順位付けされていることがわかる．

表 3.4　*sVSMScore* のみによるファイルの順位付けリスト

順位	候補ファイル名	コサイン類似度
1	tomcat/java/org/apache/tomcat/jni/Stdlib.java	0.586275
2	tomcat/java/org/apache/tomcat/jni/OS.java	0.551435
3	tomcat/java/org/apache/jasper/compiler/JDTCompiler.java	0.499763
4	**tomcat/java/org/apache/jasper/compiler/AntCompiler.java**	**0.446665**
5	tomcat/java/org/apache/jasper/tagplugins/jstl/core/When.java	0.446448
6	tomcat/java/org/apache/tomcat/jni/Buffer.java	0.438573
7	tomcat/test/org/apache/tomcat/util/net/TestXxxEndpoint.java	0.426327
8	tomcat/java/org/apache/jasper/compiler/Compiler.java	0.423611
9	tomcat/java/org/apache/tomcat/jni/File.java	0.420008
10	tomcat/java/org/apache/jasper/JspC.java	0.418780
...	...	...

3.3 BugLocator に基づくテキスト類似度

BugLocator は tf-idf インデキシング [3] に基づくテキスト類似性を評価するベクトル空間モデルを用いた IR 手法である．また，BugLocator は，バグ報告とファイル間のテキスト類似性だけでなく，過去に修正された類似するバグの情報も考慮している. 我々は最高水準の IR 手法と組み合わせることにより我々の手法の底上げを図る．我々の手法に Buglocator スコアを組み合わせるために，我々は以下の手順を行う．我々は，Ye ら [6] によって提供されたバグ報告と修正済みファイルとともに，対象リポジトリ内で Buglocator を実行する．Buglocator はテキストファイルとして以下の 3 つのスコアを提供する．

LengthScore Zhou ら [3] が論文内で用いる，ソースコードファイルの大きさ (含まれる項数) を考慮するための指標 (g(#terms)) であり，彼らの提供するツール上ではこのように名付けられている．

unweightedVSMScore Zhou らが論文内で用いる，彼らが用いたテキスト類似性をさらに最適化したベクトル空間モデルによるスコア．ツールでは *VSMScore* と名付けられているが，ここでは以下による *LengthScore* と掛け合わせた後の *VSMScore* の定義と区別するため *unweightedVSMScore* と名付ける．

SimiScore Zhou らが論文内で用いる，類似するバグによる情報を考慮するためのスコア．ツールでは *SimiScore* と名付けられている．

まず，*VSMScore* を以下で定義する．

$$VSMScore = LengthScore \times unweightedVSMScore \tag{3.4}$$

次に，以下の正規化関数を用いて *VSMScore* と *SimiScore* を組み合わせる:

$$N(x) = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{3.5}$$

BugLocator score は以下で定義される:

$$BLScore = (1 - \alpha) \times N(VSMScore) + \alpha \times N(SimiScore) \tag{3.6}$$

ここで，α は 2 つのスコアを組み合わせる際の重み因子 ($0 \leq \alpha \leq 1$) である．Zhou らは [3] の中で，一般的に $\alpha = 0.2, 0.3$ が好ましいと説明している．我々は，その周辺

の $\alpha = 0.2, 0.3, 0, 4$ の場合の性能を予め試した．その結果，$\alpha = 0.3$ の際の性能が最も高いことを発見している．本稿では，我々は $\alpha = 0.3$ と設定する．表 3.5 は，表 2.1 で示した Tomcat のバグ報告 (ID: 32157) に対する，*BugLocatorScore* のみによる順位付けリストを示している．

3.4 Bugspots に基づくバグ修正履歴

性能向上のために，我々は Google 開発チームによって提案されたヒューリスティックな単純バグ予測の実装である Bugspots [11] を用いる．本ツールはあらゆる git リポジトリにおいて Bugspots を指摘し，開発者に対してホットスポットを提示する．Bugspots で出力されるスコアは以下の式 3.7 の右辺による計算に基づいている．Bugspots が出力する本スコアを，我々は *Bugspots score* と名付ける:

$$BugspotsScore = \sum_{i=0}^{n} \frac{1}{1 + e^{-12t_i+12}} \tag{3.7}$$

ここで，n はコードへのバグ修正の総回数である．また，t_i はコード修正のタイムスタンプであり，0 から 1 の小数で定義される (コードが生成された時に 0 となり，現在時刻に 1 となる)．表 3.6 は，表 2.1 で示した Tomcat のバグ報告 (ID: 32157) に対する，*BugspotsScore* のみによる順位付けリストを示している．

3.5 報告情報に基づくコンポーネントの局所焦点化

検索性能向上のために，我々は sVSM に開発者から報告されたコンポーネント情報を組み合わせる．例えば，表 2.1 に示した Tomcat における例ように，開発者によって “Jasper” に分類されたバグ報告に対して，我々は “jasper” が関連ファイルのフルパス名に含まれると推測する．ここでのコンポーネントの分類で用いられる文字列は，Ye ら [6] のデータセットからバグ報告とともに抽出されたコンポーネント情報を記した文字列を用いる．なお，本文字列は表 3.2 の 2 行目に示されている．本例では，関連ファイル AntCompiler.java を含む `word2vec` を用いて得られた表 2.4 に見られるように，“jasper” が関連ファイルのフルパス名に含まれることがわかる．我々はこの情報を用いて検索における緩い局所焦点化を行う．仕組みは単純であり，我々はフルパス名に関連語が出現するたびに 1 カウントアップを行う．我々は *Component*

表 3.5　*BugLocatorScore* のみによるファイルの順位付けリスト

順位	候補ファイル名	BugLocator Score
1	**tomcat/java/org/apache/jasper/compiler/AntCompiler.java**	**0.700000**
2	tomcat/java/org/apache/catalina/startup/ContextConfig.java	0.385640
3	tomcat/java/org/apache/jasper/compiler/ErrorDispatcher.java	0.283191
4	tomcat/java/org/apache/jasper/EmbeddedServletOptions.java	0.244282
5	tomcat/java/org/apache/jasper/compiler/ErrorHandler.java	0.243747
6	tomcat/java/org/apache/jasper/Options.java	0.235803
7	tomcat/java/org/apache/jasper/compiler/JavacErrorDetail.java	0.232765
8	tomcat/java/org/apache/jasper/JspCompilationContext.java	0.217829
9	tomcat/java/org/apache/jasper/compiler/DefaultErrorHandler.java	0.214800
10	tomcat/java/org/apache/jasper/servlet/JspServletWrapper.java	0.204875
...	...	...

表 3.6　*BugspotsScore* のみによるファイルの順位付けリスト

順位	候補ファイル名	BugspotsScore
1	tomcat/java/org/apache/tomcat/util/net/AprEndpoint.java	10.3675
2	tomcat/java/org/apache/catalina/core/StandardContext.java	6.4257
3	tomcat/java/org/apache/coyote/http11/Http11Processor.java	6.2728
4	tomcat/java/org/apache/tomcat/util/net/NioEndpoint.java	5.6000
5	tomcat/java/org/apache/coyote/http11/AbstractHttp11Processor.java	4.0404
6	tomcat/java/org/apache/catalina/connector/Request.java	3.7771
7	tomcat/java/org/apache/tomcat/websocket/WsSession.java	3.7333
8	tomcat/java/org/apache/catalina/startup/HostConfig.java	3.5982
9	tomcat/java/org/apache/tomcat/websocket/WsWebSocketContainer.java	3.5955
10	tomcat/java/org/apache/tomcat/websocket/WsRemoteEndpointImplBase.java	3.4951
...	...	...
565	**tomcat/java/org/apache/jasper/compiler/AntCompiler.java**	**0.1991**
...	...	...

score を以下で定義する:

$$ComponentScore = \begin{cases} +1 & (\text{各コンポーネント関連語毎}) \\ 0 & (\text{コンポーネント関連語なし}) \end{cases} \tag{3.8}$$

表 3.7, 3.8 は本研究で対象とする Tomcat 及び Birt における，コンポーネント文字列とフルパス名に見られる関連度の組み合わせを示している．

また，表 3.9 は，表 2.1 で示した Tomcat のバグ報告 (ID: 32157) に対する，*ComponentScore* のみによる順位付けリストを示している．表 2.1 に示されている Component 情報文字列 “Jasper” に関連する関連語，“jasper” 及び “jsp” をフルパス名に含むファイルが取り上げられていることがわかる．

3.6 最終スコア

ソースコードファイルの問い合わせによる *sVSMScore*，Buglocator を用いた *BugLocatorScore*，Bugspots を用いた *BugspotsScore*，コンポーネント情報を用いた *ComponentScore* の 4 つのスコアを計算後，以下のように各ファイルに対してそれらのスコアを組み合わせる:

$$FinalScore = \beta \cdot sVSMScore + BugLocatorScore$$

$$+\gamma \cdot N(BugspotsScore) + \delta \cdot ComponentScore \tag{3.9}$$

ここで β, γ 及び δ は重み因子である．従って，*FinalScore* は *sVSMScore*, *BugLocatorScore*, *BugspotsScore* 及び *ComponetScore* の加重和である．我々の手法は *FinalScore* によって降順にソースコードファイルを順位付けし，最終的に順位付け済みのファイルのリスト (*FinalRank*) を返す．ファイルの順位が高いほど，バグ報告により関連している．3.3 節での説明と同様に，我々がスコアを組み合わせる前に，我々は式 (3.5) で定義される正規化関数を用いて *BugspotsScore* を 0 から 1 の範囲に正規化する．パラメータ β, γ, 及び δ は 4 つのスコアの重みを調整する．我々は実験を通して β, γ, 及び δ の値を決定した．我々が行った実験を通して，我々は $\beta = 0.5, \gamma = 0.3$ 及び $\delta =$ { 0.1 (Tomcat の場合), 1.0 (Birt の場合)} に設定した際，提案手法がより良い性能を発揮することを発見した．β, γ, 及び δ の値の決定については，ある程度手

表 3.7　Tomcat におけるコンポーネント関連語

コンポーネント文字列	フルパス名に見られる関連語
Jasper	jasper, jsp
Connecto	connection, connect
Servlet	servlet
Catalina	catalina
Cluster	build
Manager	manager
Common	common
Integrat	integrate
Document	docs, document
Servlets	servlets
jdbc-poo	jdbc-pool
Library	lib
Native:I	native
Bayeux	bayeux
Webapps	webapps
Examples	example
Jasper2	jasper2
Specific	specific
WebSocke	websocket
EL	el
Util	util
Packagin	build

表 3.8　Birt におけるコンポーネント関連語

コンポーネント文字列	フルパス名に見られる関連語
Build	build
Chart	chart
Data	data
DataAcc	data, access
Report	report
ReportE	report, engine
ReportD	report, design
ReportV	report, viewer
TestSui	test, suite
Document	document
Converte	convert

表 3.9　*ComponentScore* のみによるファイルの順位付けリスト

順位	候補ファイル名	ComponentScore
1	tomcat/java/org/apache/ **jasper**/compiler/ELInterpreterFactory.java	1.00000
2	tomcat/java/javax/servlet/ **jsp**/tagext/PageData.java	1.00000
3	tomcat/java/org/apache/ **jasper**/el/JspMethodExpression.java	1.00000
4	tomcat/test/org/apache/ **jasper**/compiler/TestCompiler.java	1.00000
5	tomcat/java/org/apache/ **jasper**/tagplugins/jstl/core/Out.java	1.00000
6	tomcat/java/javax/servlet/ **jsp**/JspTagException.java	1.00000
7	tomcat/java/org/apache/ **jasper**/JspC.java	1.00000
8	tomcat/java/org/apache/ **jasper**/compiler/JspConfig.java	1.00000
9	tomcat/java/javax/servlet/ **jsp**/tagext/TagAttributeInfo.java	1.00000
10	tomcat/java/org/apache/ **jasper**/compiler/ELFunctionMapper.java	1.00000
...	...	...
171	**tomcat/java/org/apache/jasper/compiler/AntCompiler.java**	**1.00000**
...	...	...

動でパラメータをずらしながら試行を重ね，その中でより優れていると考えられたパラメータを選択している．詳細なパラメータの変化による CombBL の性能に対する各構成要素の寄与率の調査は今後の課題とする．

表 3.10 は，表 2.1 で示した Tomcat のバグ報告 (ID: 32157) に対する，*FinalScore* による順位付けリストを示している．*sVSMScore* のみによる表 3.4 では 4 位に順位付けされていた，バグ報告に対する修正済みファイル AntCompiler.java は，表 3.10 では 1 位に順位付けされていることがわかる．

表 3.10　最終スコアによるリスト

順位	候補ファイル名	最終スコア
1	**tomcat/java/org/apache/jasper/compiler/AntCompiler.java**	**1.029094**
2	tomcat/java/org/apache/jasper/JspC.java	0.566072
3	tomcat/java/org/apache/jasper/compiler/ErrorDispatcher.java	0.549519
4	tomcat/java/org/apache/jasper/JspCompilationContext.java	0.548431
5	tomcat/java/org/apache/jasper/compiler/ErrorHandler.java	0.546335
6	tomcat/java/org/apache/jasper/EmbeddedServletOptions.java	0.537397
7	tomcat/java/org/apache/jasper/compiler/JavacErrorDetail.java	0.536135
8	tomcat/java/org/apache/catalina/startup/ContextConfig.java	0.528452
9	tomcat/java/org/apache/jasper/Options.java	0.527744
10	tomcat/java/org/apache/jasper/compiler/Generator.java	0.522090
...	...	...

4. 実験準備

4.1 対象プロジェクト

先行研究との比較のために，我々は [6] で提供された同じデータセットを用いる．表 4.1 は我々の実験で用いられた対象プロジェクトの緒元を示す(注 1)．我々はデータセットに含まれる以下の 2 つのプロジェクトを用いる:

1. Birt(注 2): Eclipse をベースとするビジネス効率化及び報告書作成ツール．
2. Tomcat(注 3): Web アプリケーションサーバーとサーブレットコンテナ．

彼らがデータセットで対象とするソースコードは，.java のみとされている．Ye ら [6] 及びその手法に従って実験した Lam ら [7] らは，対象の Git リポジトリから関連するコミットをたどり java ソースコードのファイルハッシュを取り出している．これに対し，我々は，以下でも記述するように一つのコードリビジョンを用いて，その全てのファイルをディレクトリを走査することにより取得したのち，その他のファイルを取り除き，java ファイルのみを抽出した．手順は異なるが，このように我々も .java のみを対象とし実験を行った．つまり，java ファイルのみを用いる点については Ye ら及び Lam らと同様である．また，lscp が java のみに対してしか機能しない点もここに記しておく．

さらに，我々の実験と先行研究 [6,7] の間の条件における注意すべき重要な 2 つの相違点を以下に示す．

1. 我々は複数のバグ報告においてシステムの性能を評価するために一つのコードリビジョンを用いる．
2. 我々はクロスバリデーションを行っていない．

この二点について以下で詳しく説明する．

(注 1)：全てのバグ報告, ソースコードへのリンク及び 不具合ファイル, API 文書, 及びバグとファイルのマッピングについての事前情報は全て http://dx.doi.org/10.6084/m9.figshare.951967 thanks to the authors. より公的に入手可能である
(注 2)：https://www.eclipse.org/birt/
(注 3)：http://tomcat.apache.org

表 4.1　対象プロジェクト

プロジェクト名	期間	# バグ報告数	# java ソースコードファイル数
Tomcat	2007/02-2014/01	920	2,041
Birt	2006/05-2013/12	3,928	9,731

4.1.1 コードリビジョンの取得方法

本研究で対象とするデータセットを作成したYeら [6] は，実施すべき実験手順について以下のように記している．また，Lamら [7] もYeらの手順と同様に行っている．

バグ報告に対する不具合ファイル特定の既存研究では，複数のバグ報告におけるシステムの性能を評価するのにただ一つのみのコードリビジョンを用いている．しかし，ソフトウェアのバグはソースコードパッケージの異なるリビジョンにおいてよく発見される．つまり，ただ一つのみのコードリビジョンを用いて評価を行うことは，実際に用いられる際のシステムの実際の性能と合致しない性能評価へと誤って導くおそれがある．例えば，評価のために用いられる修正済みのリビジョンには古いバグ報告に対するバグ修正の情報が含まれかねない．さらに，不具合ファイルは，バグが報告された後に削除された場合，修正済みのリビジョンに含まれていない可能性もある．修正済みのコードリビジョンを用いることに関連する上記の問題を回避するために，彼らは各バグ報告に対する修正前のバージョンへとチェックアウトを行った．

バグが報告されたソフトウェアパッケージの正確なバージョンは不明である．従って，各バグ報告に対して，修正がコミットされる直前のソフトウェアパッケージに関連するバージョンが実験内で用いられた．これはバグが元々報告された時のものと完全に一致するバージョンでないかもしれない．従って，この関連付けは現実世界で行われたことを正確に反映していない可能性もある．しかしながら，関連する修正はチェックインされておらず，バグは依然そのバージョンに存在しているため，彼らは評価においてこの関連付けを用いることは賢明であると述べている．

上記の内容に対し，我々はデータセットにおける最新の修正済みファイルに対する修正コミットのタイムスタンプ時の一つのコードリビジョンのみを用いた．そのため，上記に書かれている問題点を抱えていることに注意が必要である．

4.1.2 クロスバリデーション適用の有無

本研究で対象とするデータセットYeら [6] は，実施すべき実験手順について以下のように記している．また，Lamら [7] もYeらの手順と同様に行っている．

訓練用とテスト用のデータを分割するために，各ベンチマークデータセットによ

るバグ報告は報告のタイムスタンプによって時刻の順に並び替えられる．彼らは，AspectJ, Birt, Eclipse UI, JDT, SWT, 及び Tomcat の 6 つのデータセットを用いているが，AspectJ 以外のすべてのデータセットに対し，データセットを 10 の均等な規模の fold に分割した．そのようにして，データセットに対して 10-fold のクロスバリデーションを適用している．

これに対し，我々はクロスバリデーションを行っていない．そのため，訓練用データと同じデータを用いてテストデータを評価することに関連する，様々な問題を抱えている可能性がある．この点についても注意が必要である．

4.2 研究設問

我々の実験は以下の研究設問に回答するために設計される．

設問 1*:* 不具合ファイル特定において提案する *sVSM* 及び *CombBL* の各構成要素による正答率はどの程度か？

設問 2*: DNN [7]* と比較して我々の *sVSM* によって学習された関連性の違いは何か？

設問 3*:* 先行研究と比較して *CombBL* はどの程度の正答率で不具合ファイルを特定するか？

4.3 評価メトリクス

我々は提案する不具合ファイル特定の性能を測定するために，以下の 3 つのメトリクスを用いる．

Top-k **正答率**は以下のように測定される．与えられたバグ報告に対して，提案手法はファイルをスコアに従って順位付けする．この時，我々はカットオフ順位 k $(k = 1, 2, 3, \cdots, 20)$ を設定する．もし，バグ報告に対する不具合ファイルがファイルの top-k リストに順位付けされている場合，我々はこの例を不具合ファイル特定において正答とみなす．また，その逆の例は不正答とみなす．正答率がより高いほど，不具合ファイル特定の性能はより高い．

以下の 2 つのメトリクスは複数の不具合ファイルを持つバグ報告の例を考慮するために必要である．

MRR (Mean Reciprocal Rank) は複数の関連文書を持つクエリに対して最初の正しい回答が現れる順位を評価するメトリックである．最初，我々は各クエリに対する正しい回答が現れる順位 (RR (Reciprocal Rank)) を得る．次に，システムの検索の正確さを測定するために，我々は全ての検索クエリに対する RR の平均 (MRR) を得る．式 (4.1) は一連のクエリ Q に対する MRR を示す．

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{4.1}$$

MRR の値が高いほど，不具合ファイル特定の性能は良い．

MAP (Mean Average Precision) は複数の関連文書を持つクエリに対する情報検索の質を評価するメトリックである．MAP の値を計算するために，我々は単一のクエリの平均適合率 ($AvgP$) を考える．$AvgP$ はクエリに対して得られる適合率の値の平均であり，式 (4.2) によって計算される．

$$AvgP_i = \sum_{i=1}^{M}\frac{P(j)\times pos(j)}{\text{number of positive instances}}, \tag{4.2}$$

ここで，j は順位，M は検索されたインスタンス数，$pos(j)$ は順位 j のインスタンスが関連しているか否かを表す．また，$P(j)$ は与えられたカットオフ順位 j における適合率であり，式 (4.3) で定義される．

$$P(j) = \frac{\text{number of positive instances in top } j \text{ positions}}{j} \tag{4.3}$$

この時，一連のクエリ Q に対する MAP は各クエリに対する平均適合率スコアの平均である．

$$MAP = \frac{1}{|Q|}\sum_{i=1}^{|Q|}AveP_i \tag{4.4}$$

我々は全ての関連ファイルを特定するための我々の手法の平均性能を測定するために MAP を用いる．MAP の値が高いほど，不具合ファイル特定の性能は良い．

5. 実験結果

5.1 研究設問に対する実験結果

研究設問 1 及び 2 では，我々は最小プロジェクトである Tomcat について議論する．

設問 1*:* 不具合ファイル特定において提案する *sVSM* 及び *CombBL* の各構成要素による正答率はどの程度か？

最初，我々は我々の手法における各構成要素の top-k 正答率を調査する．図 5.1 は実験結果を示している．sVSM 単独では高い正答率ではないことがわかる．その理由は，sVSM は低次元の新しい空間への射影のために，与えられたバグ報告に対して意味的に関連するソースコードを無差別に抽出するからである．つまり，sVSM は真の目的ファイルを抽出できるが，与えられたバグ報告に対して修正する対象ではない多くの無関係なソースコードファイルも含んでしまう．この洞察から，本手法の利点を生かしながら欠点を取り除くために，我々は他の手法から得られる順位付けのためのスコアを組み合わせるべきであるという考えをもつ．コンポーネント情報を加えることにより，正答率は top-20 において 8% 向上する．Bugspots スコアを加えることにより，正答率は top-20 において sVSM+Comp から 7% 向上し，sVSM 単独からは 15% 向上する．さらに，上記の両方のスコアを足し合わせることにより，正答率は top-20 において 25% 向上する．

次に，我々は Buglocator について考える．テキスト類似度を用いるモデル (rVSM) は Buglocator に相当する (なお，sVSM，バグ修正履歴情報及びコンポーネント情報を用いてない)．Buglocator(短縮形, BL) は非常に効果的な手法であり高い正答率を誇る．top-1 正答率では，sVSM より 35% も高い．BL は top-1 で 36%，top-20 で 75% を達成している．その上で，我々は sVSM と BL を組み合わせたが，BL+sVSM に示されるように BL より 1% 高い正答率を示している．折れ線に沿って見ていくと，sVSM は BL と組み合わせることで正答率の向上に貢献している．

最後に，*ComponentScore* と *BugspotsScore* を組み合わせることにより，CombBL を構成する．図 5.1 から，CombBL は top-1 で 36%，top-20 で 80% を達成している．

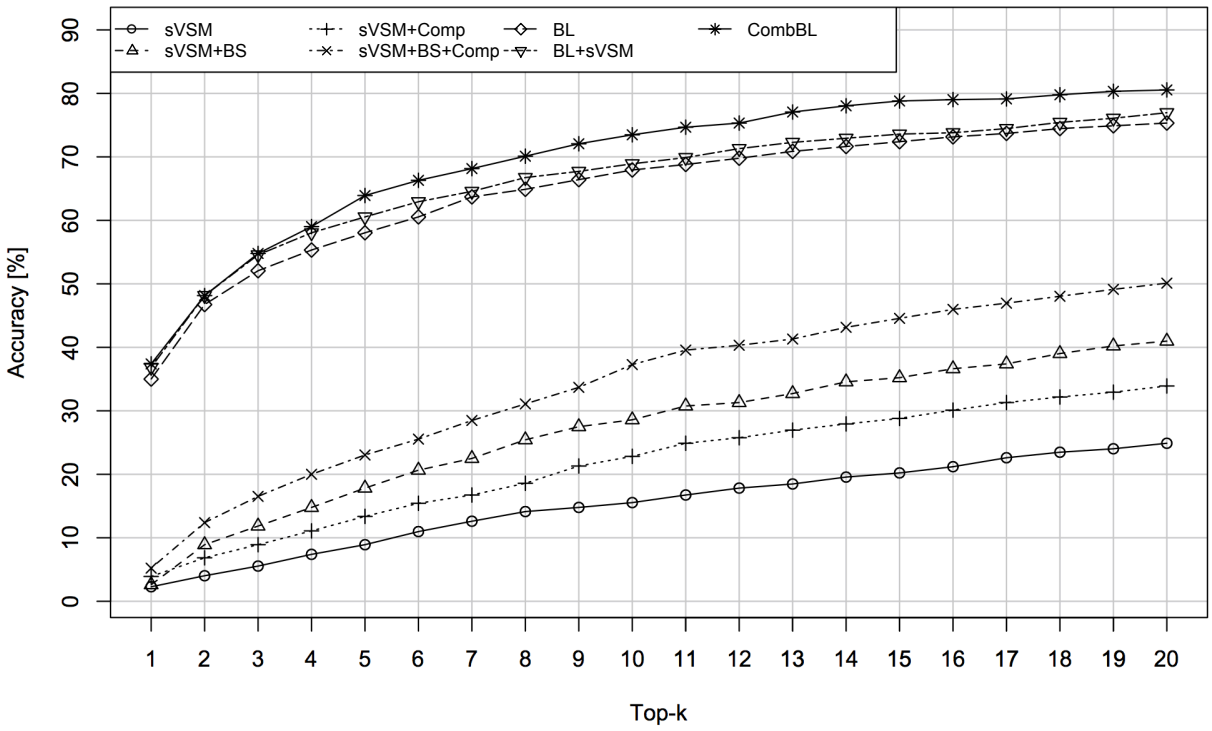

図 5.1　異なる構成要素における top-k 正答率

設問 2*: DNN [7]* と比較して我々の *sVSM* によって学習された関連性の違いは何か？

RQ2 に回答するために，我々は sVSM と DNN 間での学習されたベクトル空間の違いを調査する．NLP において，研究者は DNN が意味的あるいは文法的に関係している単語を連続する値を持つ特徴空間近い位置に射影しそれらを結び付けられることを紹介している [7]．`word2vec` によって学習される単語ベクトルも似た性質を持つ．sVSM を DNN と比較する前に，注意すべきこととして以下がある．彼らの DNN モデルはバグ報告と関連する修正済みファイルの情報と結びつけて学習している一方で，我々の手法はその情報を用いずに学習している．このことによって sVSM と DNN 間の学習されたベクトル空間に違いが生じる．我々はコーパスを学習した後に図 3.2 における sVSM に対して実験を行った．先行研究 [7] において，彼らは DNN モデルによって計算された際に関連度がトップスコアとなった，バグ報告内の項と関連する識別子を数例示している．我々は彼らの DNN モデルによってトップスコアが算出された例において，我々の sVSM 手法ではどの程度の関連度が算出されるかを調査する．sVSM において，識別子及び項は合成語からの分解，語幹抽出及び小文字化の処理が施されている．その後，我々は例に対する関連度を計算する前に手動で前処理を施した．ここでの前処理とは，合成語の識別子を予め構成語に分解する処理のことを言う．この処理が必要となる理由は，[7] における DNN では識別子における合成語をそのまま入力としていたが，sVSM での関連度計算の際には合成語を分解した構成単位で計算がなされるので，予め分解した単位での入力が求められるためである．我々は学習されたコーパスを用いてコサイン類似度を計算し，それを目的の単語あるいは一連の単語の間の関連度とみなす．

表 5.1 は先行手法 [7] に示された接続項の我々の sVSM 手法におけるコサイン類似度を示している．DNN が最上位に順位付けした識別子であっても，sVSM によって計算された関連度はそれほど高くないことがわかる．ここで，省略語 (すなわち，context に対する envCtx = env ctx) に注目すると．異なる語彙的な項にもかかわらず，`word2vec` はある程度の関連付けができている．また，元の単語を含む一連の単語を見ると，モデルはこれらの関連度を高く計算できている．この特徴はテキスト類似度に似たものであるが，`word2vec` は rVSM と比較して元の単語を含み構成される合成語を認識することに対して寛容である．これらの観察から，我々は sVSM は

表 5.1　先行手法 [7] における接続項のコサイン類似度

バグ報告内の項	識別子 1	識別子 2	識別子 3	識別子 4
context	authorization	ctx	envCtx	asyncContext
	authorization	ctx	env ctx	async context
	−0.062478	0.265840	0.281267	0.736054
resource	virtualClasspath	changeSessID	setSecureClass	addLocalEjb
	virtual class path	change sess id	set secure class	add local ejb
	0.145604	−0.151775	0.018194	0.288019
writer	globalCacheSize	charset	index	charsWritten
	global cache size	char set	index	chars written
	0.108635	0.055311	−0.008568	0.094212
read	headerLength	InternalBuffer	readBytes	dir
	header length	internal buffer	read bytes	dir
	0.099273	0.606341	0.772343	0.067292

省略語や合成語を認識する寛容性を持つとまとめられる．sVSM は DNN と完全に同じものではないが，テキスト類似度だけでは認識できない関連性を確かに認識できると考えられる．そのため，sVSM はバグに関連するファイルを特定し，比較的より高く順位付けできると考えられる．

設問 3*:* 先行研究と比較して *CombBL* はどの程度の正答率で不具合ファイルを特定するか？

RQ3 に回答するために，我々は Kim ら [4] によるナイーブベイズ (NB)，LR (learn-to-rank) [6]，BL [3] 及び HyLoc を含む最高水準の手法と CombBL を比較する実験を行った．NB は ML 手法であり，BL は IR 手法，LR は併用手法，また HyLoc は併用かつ DNN を実装した手法である．Lam ら [7] は彼らの論文の中で正答率をまとめている．彼らは [6] に報告された結果を用いている．NB に対しては，彼らは元の論文の記述に従って再実装している．我々は Ye のデータセットを用いて，できる限り近い手順と同じメトリクスで我々の実験を行った．そのため，我々は [7] に報告された最高水準の手法と正答率を比較する．

我々は結果を表 5.2 に示す．表 5.2 では，各プロジェクトに対して先行研究による結果は破線の上側に，我々の結果は破線の下側に示されている．

注意すべきこととして，我々の実装による BL の正答率は実験条件の違いのため [6] において報告された結果より比較的低い値となっている．この違いを示すために，我々は我々の実装による BL の結果を示す．次に，我々は組み合わせモデル CombBL を各先行手法を比較する．

表 5.2 に示されるように，top-5 正答率では，CombBL は BL と比較して Tomcat において 1.8% 高い，Birt において 2.6% 高い値を達成している．top-20 正答率では，向上の幅は Tomcat で 0.3%，Birt で 5.4% である．learn-to-rank (LR) 手法 [6] と比較すると，top-5 正答率では，CombBL は Tomcat で 2.6% 低い，Birt で 0.4% 低い値となった．top-20 正答率では，CombBL は 1.6% 低く，Birt では同程度の値となった．この結果から，CombBL は LR 手法より低い正答率であるが，正答率は近い値となっている．

HyLoc 手法 [7] と比較すると，top-5 正答率では，CombBL は Tomcat で 6.1% 低い，Birt で 8.5% 低い値である．top-20 正答率では，CombBL は Tomcat で 5.1% 低い，Birt で 8.6% 低い値である．予想通り，CombBL は HyLoc より低い正答率となった．

表 5.2　正答率の比較

プロジェクト	モデル	1	2	3	4	5	10	15	20	MRR	MAP
TomCat	HyLoc	51.6	59.6	64.1	68.3	71.0	77.6	82.2	85.6	0.60	0.52
	LR	46.2	54.2	59.8	62.3	66.5	74.7	80.1	82.1	0.55	0.49
	BL	35.5	48.7	52.9	58.7	61.8	71.1	77.3	80.2	0.48	0.43
	NB	5.2	6.9	8.3	8.8	9.0	11.9	14.5	16.6	0.08	0.07
	DNN	2.3	4.0	-	-	9.1	15.4	-	25.0	-	-
	CombBL	**37.3**	**48.1**	**54.7**	**59.0**	**63.9**	**73.4**	**78.8**	**80.5**	**0.49**	**0.44**
	BL+sVSM	**36.8**	**48.1**	**54.5**	**58.0**	**60.5**	**68.9**	**73.5**	**76.9**	**0.48**	**0.43**
	BL	**35.0**	**46.7**	**52.0**	**55.3**	**58.0**	**67.9**	**72.3**	**75.3**	**0.46**	**0.41**
	sVSM	**2.2**	**4.0**	**5.5**	**7.3**	**8.9**	**15.5**	**20.2**	**24.8**	**0.06**	**0.05**
Birt	HyLoc	19.1	24.8	29.7	33.5	36.0	44.6	51.1	55.4	0.28	0.20
	LR	12.4	18.1	22.5	25.1	27.9	37.3	42.4	46.0	0.20	0.15
	BL	11.1	16.2	20.0	22.4	24.9	32.1	37.0	40.6	0.18	0.14
	NB	2.9	4.7	6.5	7.9	8.7	13.8	15.9	17.6	0.06	0.05
	CombBL	**12.4**	**18.1**	**21.9**	**25.4**	**27.5**	**36.6**	**41.7**	**46.0**	**0.20**	**0.15**
	BL+sVSM	**9.6**	**14.1**	**16.6**	**19.0**	**21.0**	**28.2**	**32.6**	**36.4**	**0.15**	**0.11**
	BL	**8.7**	**12.8**	**15.9**	**18.1**	**20.0**	**26.8**	**31.1**	**34.5**	**0.14**	**0.10**
	sVSM	**1.1**	**2.4**	**3.3**	**4.1**	**4.7**	**8.1**	**10.8**	**13.1**	**0.03**	**0.02**

NB 手法 [4] と比較すると，top-5 正答率では，CombBL は Tomcat で 53.1% 高い，Birt で 17.5% 高い値である．top-20 正答率では，CombBL は Tomcat で 63.9% 高い，Birt で 28.4% 高い値である．予想通り，NB との比較では，CombBL は正答率において大幅に高い値である．我々はこの理由について，Lam らの考察 [7] と同様に考えている．主な理由として，NB 手法は学習のために過去に修正されたファイルの情報をラベルとして用いているが，その内容を考慮していない．

一般に，CombBL は MRR 及び MAP において常に高い値を示している．すなわち，最も良い場合の 2 例を挙げると，2 回に 1 回は正しいバグ関連ファイルを 1 位に順位付けする，あるいは，2 回とも正しいバグ関連ファイルを 2 位に順位付けすることが考えられる．上記の結果を踏まえて考察すると，我々は CombBL は正答率，MRR 及び MAP において HyLoc を除く最高水準の IR 及び ML 手法と同程度であると言える．

5.2 スケーラビリティと時間効率

この項では，我々は我々のアプローチのスケーラビリティと時間効率を議論する．その前に、我々はデータセットにクロスバリデーションを適用しないので，各データセット毎に一回のみの実行となることに注意する必要がある．また，我々は DNN [7] はバグ報告と関連する修正済みファイルの情報と結びつけて学習しているが，我々のモデルではそのような情報なしで学習していることに注意が必要である．さらに，彼らのモデルはスコアを組み合わせるための係数を学習しているが，我々の手法では行っていない．従って，我々はスケーラビリティと時間効率を直接的に比較することはできない．我々の手法の利点を示すにあたり，我々は特徴射影のアーキテクチャにおける違いに注目する．

我々の全ての実験は Intel Core i7 CPU 4558U 2.80GHz (2 cores), 8 GB RAM のラップトップコンピュータで実施された．対照的に，HyLoc は Intel Xeon CPU E5-2650 2.00GHz (32 cores), 126 GB RAM のワークステーションコンピュータで実施された [7].

表 5.3 は CombBL におけるコーパス学習，ベクトル作成及びスコア計算にかかる時間を示している．我々は結果を実際にかかった実行時間から得た．

このことは我々の手法が高スケーラビリティ，非常に軽量，高電力効率及びメモ

リ占有量少であることを示す．従って，CombBL は単純な実装を行うだけで，プロジェクトに対する不具合ファイル特定の適用を真に必要とする開発者が，彼らの持つコンピュータ上で容易に実行できる．

表 5.3　学習及び予測時間 (秒)

プロジェクト	コーパス学習	BR ベクトル作成	FL ベクトル作成	CombBL 順位計算
Tomcat	3.3	0.6	0.5	6.2
Birt	18.2	2.3	2.2	227.9

6. 結言

本稿は既存の単語分散表現作成のための自然言語処理ツール word2vec を用いる sVSM と呼ばれるモデルを提案した．また，我々は Buglocator, Bugspots 及びコンポーネント情報を我々の sVSM と統合する，バグ報告に対する不具合ファイル特定のための組み合わせ手法 CombBL を提案した．オープンソースプロジェクトにおける我々の実験結果は，組み合わせによる我々の手法が個々のモデルよりも不具合ファイル特定において高い正答率となることを示している．さらに，我々は我々の手法 CombBL が最高水準の情報検索及び機械学習手法による結果に対して，提案手法は既存手法と比較しより高スケーラビリティかつ高速でありながら，ほぼ同程度の正答率を達成することを確かめた．

6.1 今後の課題

今後の課題として以下の項目がある．

- パラメータの変化による CombBL の性能に対する各構成要素の寄与率の調査．
- 完全に同じデータセットと実験手順による検証．
- スケーラビリティと速度の検証．

謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本論文の作成に至るまで，全ての面で丁寧なご指導を頂きました，本学情報工学部門水野修准教授に厚く御礼申し上げます．また，国際会議への共著を通じて研究の具体化をご支援頂きました，国立研究開発法人産業技術総合研究所崔銀惠氏に厚く御礼申し上げます．本論文執筆にあたり貴重な助言を多数頂きました，本学情報工学専攻河端駿也君，山田晃久君，藤原剛史君，森啓太君，Fruy Nicolas 氏，Anais Tournois 氏，本学情報工学課程黒田翔太君，田中健太郎君，西浦生成君，原田禎之君，学生生活を通じて筆者の支えとなった家族や友人に深く感謝致します．

参考文献

[1] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V.C. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," Software Engineering, IEEE Transactions on, vol.33, no.6, pp.420–432, 2007.

[2] S.K. Lukins, N.A. Kraft, and L.H. Etzkorn, "Bug localization using latent dirichlet allocation," Information and Software Technology, vol.52, no.9, pp.972–990, 2010.

[3] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," Software Engineering (ICSE), 2012 34th International Conference on, pp.14–24, IEEE, 2012.

[4] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," Software Engineering, IEEE Transactions on, vol.39, no.11, pp.1597–1610, 2013.

[5] A.T. Nguyen, T.T. Nguyen, J. Al-Kofahi, H.V. Nguyen, and T.N. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on, pp.263–272, IEEE, 2011.

[6] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.689–699, ACM, 2014.

[7] A.N. Lam, A.T. Nguyen, H.A. Nguyen, and T.N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, pp.476–481, IEEE, 2015.

[8] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model.," INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010, pp.1045–1048, 2010.

[9] T. Mikolov, word2vec - Tool for computing continuous distributed representations of words., Google Code（オンライン）, 入手先〈https://code.google.com/archive/p/word2vec/〉（参照 2016-2-6）.

[10] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," Advances in neural information processing systems, pp.3111–3119, 2013.

[11] I. Grigorik, Bugspots - Bug Prediction Heuristic, GitHub（オンライン）, 入手先〈https://github.com/igrigorik/bugspots〉（参照 2016-2-6）.

[12] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds, p.23, ACM, 2008.

[13] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes? (on Fridays.)," Proc. of 2nd International workshop on Mining software repositories, pp.24–28, 2005.

[14] M. Porter, The Porter Stemming Algorithm, tartarus.org（オンライン）, 入手先〈http://tartarus.org/martin/PorterStemmer/〉（参照 2016-2-7）.

[15] C.D. Manning and H. Schütze, Foundations of statistical natural language processing, vol.999, MIT Press, 1999.

[16] I.J. Good, "The population frequencies of species and the estimation of population parameters," Biometrika, vol.40, no.3-4, pp.237–264, 1953.

[17] I.H. Witten and T.C. Bell, "The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression," Information Theory, IEEE Transactions on, vol.37, no.4, pp.1085–1094, 1991.

[18] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," Proceedings of the 8th Working Conference on Mining Software Repositories, pp.43–52, ACM, 2011.

[19] D.M. Blei, A.Y. Ng, and M.I. Jordan, "Latent dirichlet allocation," the Journal of machine Learning research, vol.3, pp.993–1022, 2003.

[20] S. Deerwester, "Improving information retrieval with latent semantic indexing," Proceedings of the 51st Annual Meeting of the American Society for Information Science 25, pp.36–40, 1988.

[21] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Combining probabilistic ranking and latent semantic indexing for feature identification," Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on, pp.137–148, IEEE, 2006.

[22] C.D. Manning, P. Raghavan, H. Schütze, et al., Introduction to information retrieval, vol.1, no.1, Cambridge university press Cambridge, 2008.

[23] H.U. Asuncion, A.U. Asuncion, and R.N. Taylor, "Software traceability with topic modeling," Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, pp.95–104, ACM, 2010.

[24] Y. Bengio, "Learning deep architectures for ai," Foundations and trends® in Machine Learning, vol.2, no.1, pp.1–127, 2009.

[25] E. Arisoy, T.N. Sainath, B. Kingsbury, and B. Ramabhadran, "Deep neural network language models," Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT, pp.20–28, Association for Computational Linguistics, 2012.

[26] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," Proceedings of the 22nd International Conference on Program Comprehension, pp.53–63, ACM, 2014.

[27] R.K. Saha, M. Lease, S. Khurshid, and D.E. Perry, "Improving bug localization using structured information retrieval," Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pp.345–355, IEEE, 2013.

[28] B. Sisman and A.C. Kak, "Incorporating version histories in information retrieval based bug localization," Proceedings of the 9th IEEE Working Conference on Mining

Software Repositories, pp.50–59, IEEE Press, 2012.

[29] S. Wang, D. Lo, and J. Lawall, "Compositional vector space models for improved bug localization," 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp.171–180, IEEE, 2014.

[30] T.-D.B. Le, R.J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: better together," Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp.579–590, ACM, 2015.

[31] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," Proceedings of Workshop at the International Conference on Learning Representations (ICLR), 2013.

[32] T. Mikolov, W. tauYih, and G. Zweig, "Linguistic regularities in continuous space word representations," Proceedings of NAACL HLT, 2013.

[33] A. Zhila, W.-t. Yih, C. Meek, G. Zweig, and T. Mikolov, "Combining heterogeneous models for measuring relational similarity.," HLT-NAACL, pp.1000–1009, 2013.

[34] Y. Bengio, H. Schwenk, J.-S. Senécal, F. Morin, and J.-L. Gauvain, "Neural probabilistic language models," Innovations in Machine Learning, pp.137–186, Springer, 2006.

[35] Y. Bengio and Y. LeCun, "Scaling learning algorithms towards ai, ‖ in large-scale kernel machines, l. bottou, o. chapelle, d. decoste and j. weston, eds," 2007.

[36] T. Mikolov, "Language modeling for speech recognition in czech," PhD thesis, Masters thesis, Brno University of Technology, 2007.

[37] T. Mikolov, J. Kopeckỳ, L. Burget, O. Glembek, and J.H. Černockỳ, "Neural network based language models for highly inflective languages," Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on, pp.4725–4728, IEEE, 2009.

[38] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning internal representations by back-propagating errors," Nature, vol.323, pp.533–536, 1986.

[39] J.L. Elman, “Finding structure in time,” Cognitive science, vol.14, no.2, pp.179–211, 1990.

[40] R. Socher, C.C. Lin, C. Manning, and A.Y. Ng, “Parsing natural scenes and natural language with recursive neural networks,” Proceedings of the 28th international conference on machine learning (ICML-11), pp.129–136, 2011.

[41] Q. Le and T. Mikolov, “Distributed representations of sentences and documents,” Proceedings of the 31st International Conference on Machine Learning (ICML-14), pp.1188–1196, 2014.

[42] S.W. Thomas, lscp: A lightweight source code preprocesser, GitHub（オンライン）, 入手先〈https://github.com/doofuslarge/lscp〉（参照 2016-2-6）.

[43] J. Richardson, B. Franz, and F. Technologies, Lingua::Stem::En - Porter’s stemming algorithm for ’generic’ English, CPAN（オンライン）, 入手先〈http://search.cpan.org/ snowhare/Lingua-Stem/lib/Lingua/Stem/En.pm〉（参照 2016-2-6）.

[44] S. Bird, E. Klein, and E. Loper, Natural language processing with Python, ” O’Reilly Media, Inc.”, 2009.

[45] S. Bird, E. Loper, and E. Klein, Natural Language Toolkit (NLTK), GitHub（オンライン）, 入手先〈https://github.com/nltk/nltk〉（参照 2016-2-7）.