修士論文

コードの時系列変化を考慮した
機械学習に基づく欠陥混入の低減手法

2410064　笹川 尋翔
主指導教員　鈴木 正人

北陸先端科学技術大学院大学
先端科学技術専攻
情報科学

令和 8 年 3 月

Abstract

In modern software engineering, development processes have transitioned from traditional waterfall models to agile and continuous integration/
continuous delivery (CI/CD) frameworks. This shift allows for rapid responses to market needs. However, it also requires frequent code changes. Continuous modifications often compromise the consistency between new and existing code. Consequently, these changes increase the risk of introducing software defects. The cost of fixing such defects increases exponentially as they move toward later stages of production. Therefore, identifying potential defects early is a critical challenge for quality assurance.

Historically, defect prediction research has been based on structural metrics, such as cyclomatic complexity and lines of code (LOC). These metrics analyze the state of the code at a specific point in time. However, structural metrics cannot capture the dynamic characteristics of the development process. Recent studies have begun to utilize change metrics derived from version control systems (VCS). These approaches show a stronger correlation with defect occurrences than structural metrics. Nevertheless, existing change-related methods face two issues. First, they do not consider the irregular timing of code commits. Second, they focus on a single component, such as methods or commits, without combining local and global perspectives.

This work investigates a defect prediction method that incorporates three approaches to address these issues. First, the method treats commits as irregularly occurring events. Rather than measuring data at fixed intervals, it focuses on the differences between consecutive commits to reflect the actual changes of software development. This enables the machine learning model to track changes in complexity that may not be apparent in structural metrics.

Second, the proposed method combines multi-level metrics to capture both local and global trends. We extract per-method metrics to reflect local implementation errors. These include changes in lines of code, token counts, and cyclomatic complexity. Simultaneously, we extract per-commit metrics to represent global impact.These include the number of modified files, lines added or deleted, and "change scattering" measured by entropy. By combining these perspectives, the machine learning model can identify risks where a minor local change might have a significant global impact.

Third, we introduce a review prioritization method based on "effort-aware" defect prediction. In real-world development, resources for code review are limited. It is

often impossible to review every single change in detail. conventional methods often assume that review effort is uniform across all components. However, the actual effort depends on the scale and complexity of the changes. Our method defines a corrected review effort using the scale and complexity of the changes. Next, we apply combinatorial optimization algorithms to detect more defects with less effort.

To validate the effectiveness of our approach, we conducted experiments on five open-source projects: Elasticsearch, Hazelcast, Netty, OrientDB, and Neo4j . These projects cover diverse domains, including distributed search engines, network frameworks, and multi-model databases, ensuring that the findings are applicable to various software architectures. We used the BugHunter dataset, which provides information about defect-introducing and defect-fixing commits. The defect prediction model was built using a random forest, a machine learning algorithm capable of handling nonlinear relationships and providing feature importance. Performance was evaluated using F1 scores and Area Under the Curve (AUC) through 10-fold cross-validation.

We confirmed that the proposed method consistently improves the accuracy of the prediction in all target projects. Compared to conventional models using only structural metrics, the inclusion of change-related features led to an increase in the F1 score. In particular, for projects where the prediction performance was low using conventional methods, the proposed approach significantly improved the prediction performance. The AUC values are high for all projects, and this reflects a strong ability to distinguish between defect-introducing commits and defect-fixing commits. Statistical analysis using the McNemar test indicated that these performance gains were consistent and not attributable to random variation.

Furthermore, we found that the review prioritization method is effective under actual resource constraints. Even with a little review effort, the proposed method was able to identify a large majority of defects. For a given effort, the proposed method detected more defects than conventional methods, suggesting that this model effectively prioritizes changes with a high risk of defect introduction. The Wilcoxon signed-rank test supported the significance of this improvement. The proposed prioritization method achieved an efficiency gain, identifying approximately 70 to 75% of all defects within only 20% of the total review effort. These findings suggest that developers can find most potential issues by their a little review effort.

Analysis of feature importance revealed that commit-level metrics, such as the num-

ber of added lines and modified files, served as the strongest predictors of defect risk in most projects. However, method-level metrics also provided critical information; for example, the change in token counts was identified as a significant predictor in the Netty project. Interestingly, Partial Dependence Plot (PDP) analysis showed a trend where smaller changes often had a higher probability of containing defects. This suggests that while large commits indicate broad impact, subtle modifications at the method level can effectively signal defect risks. Additionally, small changes are easier to review, meaning defects within them are more likely to be identified and recorded in the dataset.

In conclusion, this research demonstrates that using the time-series data of code changes and combining multi-level metrics improve the defect prediction performance. By providing a model that considers the review effort, we offer a practical decision-support tool for developers. Future work should focus on identifying the specific intent behind changes, such as whether a change is for a new feature or refactoring. Understanding the purpose of the change makes it easier to detect the cause of defects and helps create more targeted advice.

目次

図目次

表目次

第 1 章

はじめに

1.1　背景

現代のソフトウェア開発プロセスは、従来のウォーターフォール型から、アジャイル開発や CI/CD（継続的インテグレーション/継続的デリバリー）を軸とした迅速かつ反復的なプロセスへと変化している。これは、市場の不確実性が高まり、ユーザーニーズの多様化に対応し続けることがビジネスの成否を分ける要素となったためである。このような環境下では、リリース後もソースコードの継続的な変更が不可欠となる。

しかし、頻繁な変更は新しいコードと既存のコードとの整合性を損なう可能性があり、副作用として新たな欠陥が混入する要因となる。ソフトウェアの欠陥の修正コストは、発見が遅れるほど指数的に増大することが知られている。開発の初期段階での修正に比べ、テスト工程や本番環境での発見は、修正作業のみならず、影響範囲の特定、再テスト、デプロイの再実行を伴うため、多くの工数を消費し、ユーザー体験の低下を招く。したがって、欠陥を早期に特定して対処することは品質保証における重要な課題となっている。

これまで、データ分析に基づく欠陥予測の研究では、ソースコードの構造的な特徴に着目していた。しかし、コードが絶えず変化する現代の開発現場においては、ソフトウェアの構造のみを分析するだけでは構造の変化に関する特徴を十分に捉えきれず、予測精度の向上に寄与しにくい。また、プロジェクトごとに異なる変更単位や変更頻度を考慮した分析手法も確立されていない。

さらに、レビュー工数のの配分も課題の 1 つである。全ての変更に対して詳細なレビューを行うことは労力の制約上困難であるため、開発現場では「どの変更を優先的に確認すべきか」という意思決定を支援するための指標が求められている。

1.2　目的

本研究の目的は、コード変更の特徴を考慮した欠陥予測手法を構築し、レビューに費やせる労力をリスクの高い変更箇所に効率的に配分する意思決定支援モデルを提示することである。具体的には、以下の 3 つの目的を達成することを目指す。

1 つ目は、異なる構成要素からなる変更メトリクスの活用である。本研究では、メソッド単位での局所的な変化と、コミット単位での全体的な変化を考慮した特徴量を検討する。これにより、単一の構成要素では見落とされがちな「局所的な修正が全体に及ぼすリスク」を捉える。

2 つ目は、不規則な時系列変化を考慮した欠陥予測モデルの構築である。コード変更を等間隔なデータではなく、不規則なタイミングで発生するイベントとして扱い、直前の状態からの差分を学習することで、変更の特徴に基づいて欠陥混入リスクを検出する。

3 つ目は、レビュー労力の測定と欠陥発見率の向上である。単なる予測精度の向上に留まらず、変更の規模と複雑さに基づくレビュー労力を定義し、少ないレビュー労力で欠陥発見率を改善する。これにより、同一のレビュー労力において、従来手法よりも高い欠陥発見率を達成可能であることを示す。

1.3　構成

本論文の構成は以下の通りである。

第 2 章では、ソフトウェア品質と欠陥予測に関する関連研究を示す。構造的メトリクス、変更メトリクス、および機械学習を用いた欠陥予測手法の動向を整理し、既存手法における時系列情報の扱いとレビュー労力の推定における課題を明確にする。

第 3 章では、本研究が提案する欠陥予測およびレビュー優先度付け手法について述べる。変更の不規則性を考慮した特徴量設計、メソッド単位とコミット単位の特徴量の活用方法、およびレビュー労力の推定モデルの構築について述べる。

第 4 章では、提案手法の有効性を検証するための実験環境について述べる。データセットの選定理由、前処理の手順、および評価指標の設定について説明する。

第 5 章では、実験結果を報告する。段階的な評価を通じて、各メトリクス群が予測性能に与える寄与度を明らかにするとともに、レビュー労力に対する欠陥発見率の累積曲線を用いて、提案手法がレビューに費やせる労力の配分の効率化にどの程度貢献するかを検証する。

第 6 章では、実験結果に基づき、変更の規模と欠陥混入確率の関係性や、プロジェクト間での特性の相違について考察する。また、本研究で採用した手法やデータセットに関連する制約についても議論する。

第 7 章では、本研究の結論をまとめ、今後の展望を述べる。

第 2 章

コード品質と関連研究

2.1　欠陥予測の研究動向

ソフトウェア開発において、保守に関わる作業はライフサイクル全体のコストの大部分を占める。JIS X 0161 では、保守作業を是正保守、適応保守、完全化保守、予防保守の 4 つに分類しており [1]、特に不具合を修正する是正保守の効率化がコスト削減の鍵となる。Microsoft Research の調査では、76% の開発者がリファクタリングによる欠陥混入を懸念していると報告されており [2]、変更に伴う欠陥混入リスクの管理が保守活動における重要な課題である。Ostrand らは、欠陥密度が最も高い構成要素の 20% に平均 83% の欠陥が集中することを示し [3]、レビュー工数を効果的に配分する戦略の重要性を示唆した。

ソフトウェア欠陥予測の初期研究では、主に構造的メトリクスが活用されてきた。1970 年代の McCabe による循環的複雑度の提案 [4] 以降、コードの論理構造と欠陥の関係が注目され、様々な指標が開発された。

2000 年代以降、バージョン管理システム（VCS）の普及により、コードの変更履歴を活用した研究が盛んになった。なぜなら、プロジェクトの規模が大きくなるにつれて、新たな変更が以前に行われた変更と競合する状況が多発するようになったためである。この問題を解決するためには、ソースコードが変更された日時、その変更を加えた開発者、変更における差分などの情報を VCS で記録することが不可欠であった。

Hassan[5] らは、変更ファイル数に基づく複雑さが欠陥予測精度の向上に貢献することを示した。近年では、機械学習技術の発展により、構造的メトリクスと VCS から得られる変更履歴を組み合わせた高精度な予測モデルが構築されている。

2.2　構造的メトリクスに基づく欠陥予測

構造的メトリクスは、ソースコードの構造的な特徴を測るためのものである。これらは主に、複雑度、規模、コンポーネント間の関係性の 3 つの観点から構成される。第一に、論理構造の複雑さを測る指標として、McCabe[4] が提案した循環的複雑度が広く用いられる。これは制御フローグラフにおける独立したパスの数として定義され、以下の式で計算される。

$$V(G) = E - N + 2P$$

ここで、E はグラフにおけるエッジの数、N はグラフにおけるノードの数、P は連結成分の数（通常、単一のメソッドを対象とする場合は $P = 1$）である。循環的複雑度が高いほど、コードの分岐やループが複雑であることを示し、欠陥混入リスクが高まる傾向にある。

第二に、規模を測る指標は、コードの量的な大きさを測定する。最も基本的なものは LOC（Lines of Code）であり、多くの研究でコードサイズと欠陥数の正の相関が報告されている [6]。

第三に、コード内の要素間の関係性を測る指標として、結合度や凝集度に関するメトリクスがある。Chidamber と Kemerer[7] が提案したメトリクス群がその代表例である。CBO（Coupling Between Objects）は要素間の結合度を、RFC（Response For Class）は要素が呼び出すメソッドの範囲を、LCOM（Lack of Cohesion of Methods）は内部要素間の凝集度をそれぞれ測定する。

これらの構造的メトリクスは、ある時点でのコードの状態を分析するのに適しているが、コードの一時的な構造のみを反映するため、変更の規模や頻度などの構造の変化に関する特徴を捉えられない。変更の特徴は構造の特徴よりも欠陥混入との関連性が高いことが指摘されており [8]、構造的メトリクスのみでは十分な予測精度を得られない。

2.3　変更メトリクスに基づく欠陥予測

コードの変更履歴を活用した欠陥予測の研究は、バージョン管理システムの普及とともに発展してきた。これらの研究では、VCS 上のコミットログから「以前の状態と今の状態」を比較した差分を抽出し、それを特徴量に変換する。これにより、ソースコードの構造だけでなく、開発者が「いつ、どの程度の頻度で、どのような規模の修正を試みたか」

という開発プロセスに潜む欠陥混入リスクをモデル化することが可能になる。Graves らは、大規模な電話交換システムを対象とした研究で、変更の頻度が欠陥の強い予測因子であることを示した [8]。

Nagappan と Ball は、変更の規模（コードの追加・削除行数）や変更ファイル数に基づく欠陥予測手法を考え、システムの欠陥密度の推定における有効性を示した [9]。

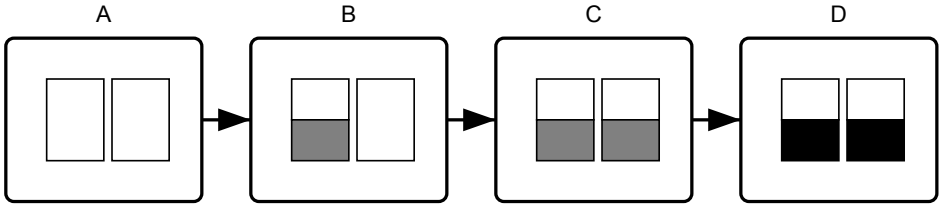
Hassan は、エントロピーの概念を用いて変更の複雑さを測定した [5]。エントロピーが高い変更ほど欠陥を含みやすいことが示された。

Kamei らは、Just-In-Time（JIT）欠陥予測モデルを提案した [10]。JIT モデルは、コミット時に欠陥混入の有無を予測し、即座にレビューやテストの対象とする。彼らは 6 つのオープンソースプロジェクトと 5 つの商用プロジェクトを用いた実証研究により、平均正解率 68%、平均再現率 64% で欠陥混入コミットを予測可能であることを示し、20% のレビュー労力で欠陥混入コミットの 35% を特定可能であることを示した。

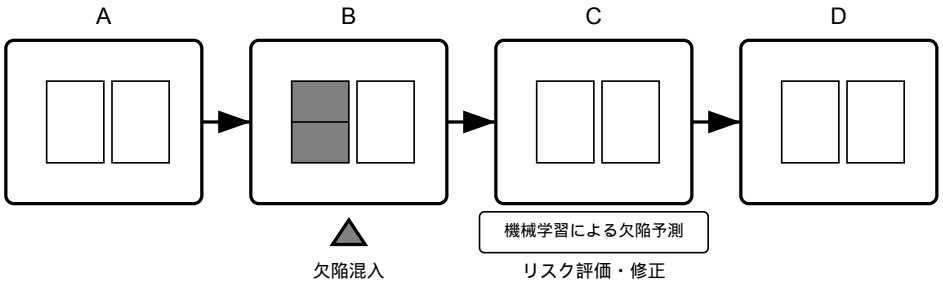
しかし、既存の変更メトリクス研究では、一連の変更データから全体の変化量を算出しており、コミットのタイミングや間隔といった時系列構造を明示的に考慮していない。ソフトウェアの変更要求やその承認は不規則なタイミングで発生する活動であり、この特徴を考慮した手法は十分に探求されていない。

2.4　機械学習を用いた欠陥予測

ソフトウェアの品質に影響を与える主な要因としては、コードの構造と開発プロセスがある。これらと欠陥の間に存在する複雑な関係をモデル化するために、機械学習技術が活用されている。先行研究では、多様なアルゴリズムの比較を通じた適切なモデル選定によって予測精度を向上させられることが示されており、人間が定義するルールでは捉えきれない潜在的な欠陥混入リスクを自動的に抽出する重要性が論じられている [11]。また、決定木を用いた機械学習モデルが、過学習を抑制しながら安定した高い分類性能を発揮することも確認されており、変数間の非線形関係を正確に捉える能力が、ソフトウェアの欠陥の判別に役立つことが示されている [12]。さらに、予測結果を実際の品質保証活動に繋げるためには、モデルのクラス分類を可視化し、開発者に納得がいく客観的な根拠を提示することが必要である。そのため、Partial Dependence Plot などの手法を用いて、予測基準を可視化することが求められる [13]。



(a) 欠陥予測を行わない場合の欠陥の広がり



(b) 欠陥予測を用いた場合の早期対応

図 2.1: 欠陥予測の有無による欠陥の広がりの比較

2.5 レビュー効率化とコスト分析

ソフトウェア開発において、レビューは品質保証活動の中心となる重要な活動であるが、レビューに費やせる労力が少ないため、全てのコードを詳細にレビューすることは困難である。一方で、欠陥の事後修正は開発者にとっては大きな負担である。開発後期の修正コストは初期段階の数倍に達することもあり、本番環境で欠陥が見つかるとユーザー体験を損なう可能性がある。これは、欠陥が他のモジュールに波及し、テストの再実行やデプロイの繰り返しが必要となるためである。

図 2.1 に示すように、欠陥予測手法を用いて早い段階で欠陥を見つけることで、欠陥の波及を防ぎ、修正コストを削減可能である。Mende と Koschke は、欠陥予測を労力を考

慮した順位付け問題として捉え直し、Effort-Aware Defect Prediction（レビュー労力に基づいた欠陥予測）の概念を提案した [14]。従来手法は各構成要素のレビュー労力が同一であると仮定していたが、実際にはそれぞれの規模や複雑度によってレビュー労力が異なる。彼らは循環的複雑度を用いてレビュー労力を計算し、少ない労力でレビューでき、かつ欠陥混入率が高い構成要素を優先的にレビューする戦略を提案した。また、Kamei らは、この手法を Just-In-Time 欠陥予測に適用した [10]。

第 3 章

時系列データに基づく分析手法

3.1　コミットの不規則性を考慮した時系列情報の活用

ソフトウェア開発におけるコミットは、不規則なタイミングで発生する。開発者の作業ペースやプロジェクトの状況により、頻繁に更新される時期もあれば、長期間動きがない時期もある。この不規則性は、従来の時系列分析手法が前提とする等間隔データとは性質が異なる。

株価や気温の予測に用いられる従来の時系列分析手法（ARIMA や LSTM など）は、データが一定間隔で測定されることを前提としている。しかし、コミットのような不規則に発生するイベントの記録は点過程データと呼ばれ、イベントの発生タイミング自体が重要な情報を持つ。本研究では、コミットを点過程データとして扱い、発生タイミングの不規則性が開発のリズムやリスクを反映すると考える。

点過程データの特性を活用するため、本研究では各コミット時点での絶対値ではなく、直前のコミットからの変化に着目する。具体的には、メソッドが書き換えられた際、現在の行数ではなく前回の状態から「何行増えたか」「構造がどれほど複雑になったか」を数値化する。この差分ベースのアプローチにより、短期間での急激な複雑化といった構造的メトリクスでは捉えられない欠陥の予兆を検出可能である。変化量や変化率が重要な理由は、変更が既存コードとの整合性を崩すリスクを伴うためである。また、急激な変化は開発者の理解が不完全な状態での作業を示唆するため、欠陥混入リスクを高める可能性がある。

本研究では欠陥予測を「将来の数値を当てる回帰問題」ではなく「その変更に欠陥が含まれるか否かの二値分類問題」として扱う。これは、レビューに費やせる労力をリスクの高い箇所に集中させることを目的としているためである。回帰問題として捉えた場合、欠

陥数の予測値を得られるが、レビュー対象の優先順位付けには欠陥混入確率の方がより有用である。

研究の全体像を図 3.1 に示す。不規則なコミットを点過程データとして整理し、直前のコミットからの変化を特徴量として抽出する。これらの特徴量から機械学習により欠陥混入確率を予測し、レビュー対象コミットの優先順位付けを行う。

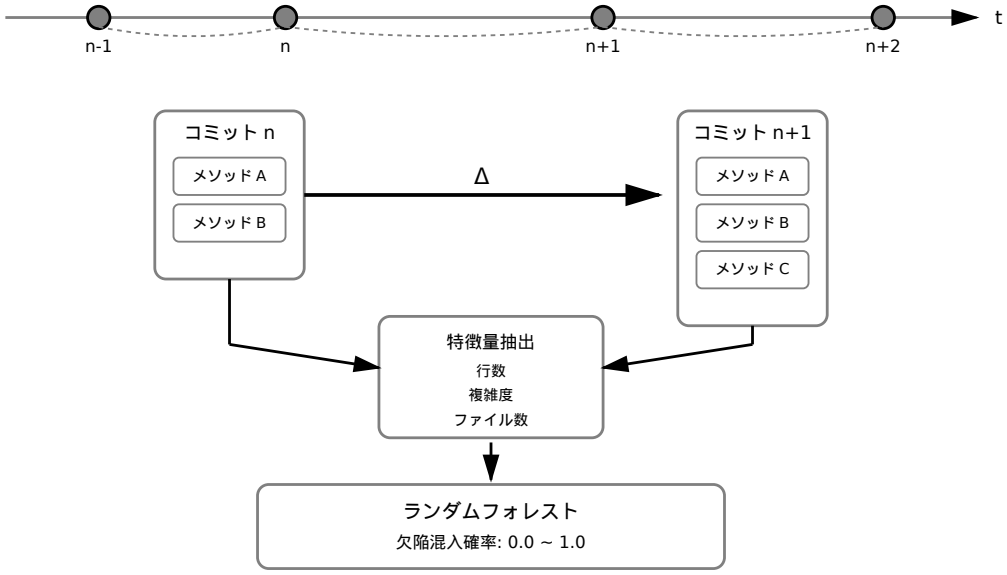


図 3.1: 時系列情報の活用

3.2 構成要素ごとの変更特性の分析

欠陥予測精度を向上させるため、コード変更を複数の視点から捉える必要がある。本研究では、メソッド単位とコミット単位という 2 つの構成要素を用いて特徴量を設計する。

この 2 つの構成要素を用いる理由は、欠陥は局所的な実装ミスとシステム全体の不整合の両方から生じるためである。メソッド単位のメトリクスは前者を、コミット単位のメトリクスは後者を捉える。また、既存研究において、単一の構成要素のメトリクスを用いるだけでは十分な予測精度を得られないことが示されている。異なる構成要素のメトリクスを組み合わせることで、互いの欠点を補完し合うことが可能である。

■メソッド単位の変更メトリクス　メソッド単位では、個々のメソッドの局所的な変化を捉えるため、直前のコミットからの変化量を特徴量として用いる。本研究では、コード変更の異なる側面を捉えるため、以下の 3 つのメトリクスを用いる。

- コード行数の変化量: 変更の規模を表し、レビュー労力と相関がある。
- トークン数の変化量: 意味のあるコード要素（変数名、演算子、キーワードなど）の変化を捉え、空白行やコメントのみの変更と論理的な変更を区別する。
- 循環的複雑度の変化量: 制御フローの複雑さの変化を測定し、条件分岐やループの追加による論理構造の変化を捉える。

■コミット単位の変更メトリクス　コミット単位では、特定のコミットに関連付いた変更の影響範囲を捉えるため、以下のメトリクスを用いる。

- 変更されたファイル数: 変更範囲の広さを表す。複数ファイルにまたがる変更を加えると、モジュール間の整合性確保が困難になり、欠陥混入リスクが高まる。
- コードの追加行数: 変更前のコードに対して追加されたコードの行数。新たなコードの導入が既存のコードに影響する場合、特に欠陥が生まれやすい。
- コードの削除行数: 変更前のコードから削除されたコードの行数。大規模な削除は、依存関係の破壊につながる可能性がある。
- 変更の広がり: 各ファイルの変更行数から算出されるエントロピーを用いて、変更の広がりを定量化する。変更が複数のファイルに分散しているほど値が大きくなり、それに伴ってレビュー労力が増加する。詳しくは 3.4 節で述べる。

メソッド単位とコミット単位のメトリクスで変化量を採用する理由は、欠陥混入リスクが絶対的な変化に比例するためである。一例として、変化率を用いると小規模なメソッドやコミットにおいて極端な値をとる。例えば、10 行のメソッドに対して 2 行追加した場合の変化率は 20% であり、100 行のメソッドに対して 2 行追加した場合の変化率は 2% であるが、この変化率の相違は欠陥混入リスクとは無関係である。

これらのメトリクスを組み合わせることで、変更の規模と複雑さの両方を捉える。

■異なる構成要素に基づく特徴量の活用　メソッド単位のメトリクスが変更の局所的な影響を捉え、コミット単位のメトリクスが変更の全体的な影響を捉えることで、欠陥予測精度が向上する。例えば、あるメソッドの循環的複雑度が大きく増加した場合（メソッド単位)、それが単一ファイルの変更なのか、複数ファイルにまたがる変更の一部なのか（コ

ミット単位）によって、欠陥混入リスクは異なる。異なる構成要素の特徴を考慮することにより、単一の構成要素では捉えられない欠陥の特徴を識別可能になる。具体的な活用方法については第 4 章で述べる。

3.3　欠陥の特徴の学習とモデル評価

本研究では、3.1 節と 3.2 節で設計した、時系列変化を考慮したメトリクスを用いて欠陥予測モデルを構築する。

■機械学習手法の選定　ソフトウェア欠陥予測に適用可能な手法には、時系列分析に特化した深層学習モデルやニューラルネットワークなど、様々なアルゴリズムが存在する。しかし、本研究の目的とデータの性質を考慮し、以下の理由からランダムフォレストを採用する。

まず、LSTM や ARIMA といった代表的な時系列分析手法は通常、データが一定間隔で測定されることを前提としている。しかし、ソフトウェア開発におけるコミットは開発者のペースや状況に応じて不規則なタイミングで発生するため、等間隔データを前提とする既存の時系列モデルをそのまま適用することは困難である。

次に、ニューラルネットワークは複数の層を通じて複雑な特徴表現を自動的に学習し、高い予測精度を達成する可能性がある一方で、その内部構造がブラックボックス化しやすいという課題がある。本研究の目的は、単に数値を予測するだけでなく、どのようなコード変更が欠陥を招くのかという予測の根拠を明らかにし、開発者の意思決定を支援することである。ランダムフォレストは、特徴量重要度を用いて各メトリクスの寄与度を定量化でき、さらに Partial Dependence Plot（PDP）により特定の特徴量と予測結果の関係を可視化することが可能である。

さらに、ランダムフォレストはブートストラップサンプリングによるアンサンブル学習を用いるため、単一の決定木に比べて過学習に強く、安定した予測性能を示す。また、ソフトウェアメトリクス間に存在する「変更行数が少ない場合であっても変更ファイル数が多い場合に欠陥混入リスクが高まる」といった非線形な相互作用を予測できるという長所もある。

このように、時系列データの不規則性への対応、現場での運用に不可欠な説明可能性、予測性能の安定性を考慮し、機械学習手法としてランダムフォレストを選定する。

■評価指標　モデルの性能を評価するための指標として、不均衡データに適した F1 スコアおよび AUC を用いる。

F1 スコアは、適合率と再現率の調和平均として以下のように定義される。

$$\text{適合率} = \frac{TP}{TP + FP}, \quad \text{再現率} = \frac{TP}{TP + FN} \tag{3.1}$$

$$F_1 = 2 \times \frac{\text{適合率} \times \text{再現率}}{\text{適合率} + \text{再現率}} \tag{3.2}$$

ここで，各記号の定義は以下の通りである。

- TP (True Positive): 欠陥と予測され、実際に欠陥がある箇所の数
- FP (False Positive): 欠陥と予測されたが、実際には欠陥がない箇所の数
- FN (False Negative): 欠陥がないと予測されたが、実際には欠陥がある箇所の数
- TN (True Negative): 欠陥がないと予測され、実際に欠陥がない箇所の数

適合率は欠陥を含むと予測したデータのうち実際に欠陥を含むデータの割合、再現率は実際に欠陥を含むデータのうち欠陥を含むと予測できた割合を表す。F1 スコアを採用する理由は、欠陥予測のために使用するデータが不均衡であるためである。一般的に、欠陥を含むメソッドやコミットの数は、欠陥を含まないメソッドやコミットの数よりも少ない。このような不均衡データでは、正解率のみでは適切な評価ができない。例えば、全てのデータを欠陥なしと予測しても、高い正確率が得られる可能性がある。F1 スコアは両者のバランスを評価するため、不均衡データに関しても適切に評価可能である。

一方、AUC（Area Under the Curve）は、分類器が陽性クラス（欠陥を含むデータ）と陰性クラス（欠陥を含まないデータ）をどの程度正確に識別可能かを示す指標である。分類のしきい値を変化させた際の真陽性率と偽陽性率の関係をプロットしたものは ROC（Receiver Operating Characteristic）曲線と呼ばれ、AUC はその曲線の下側の面積を表す。AUC の値は 0.5 から 1 の範囲をとり、1 に近いほど高い識別性能を、0.5 に近いほどランダムな予測と同等の性能であることを意味する。AUC は特定のしきい値に依存することなくモデルの全体的な識別能力を評価可能であるため、F1 スコアと併せてモデルの有効性を多角的に検証するために用いられる。

また、データの不均衡に対処するため、本研究では学習プロセスにおいてサンプリングを行う。サンプリングとは、データの不均衡を調整し、モデルの正しい学習を可能にすることである。サンプリング手法としては、多数派クラスのサンプル数を削減する手法（アンダーサンプリング）や、少数派クラスのサンプル数を増加させる手法（オーバーサンプリング）がある。これらの技術を導入する理由は、多数派クラスである欠陥を含まないデータの特徴によって、欠陥を含むデータの特徴が埋もれてしまうのを防ぐためである。

サンプリングによってクラス間の比率を調整し、モデルに欠陥の特徴を学習する機会を適切に与えることで、正解率という表面的な数値に依存しない、識別能力の高い予測モデルを構築することを目指す。

■統計的仮説検定　提案手法が既存手法と比較して統計的に有意な改善をもたらしているかを検証するため、McNemar 検定を実施する。McNemar 検定は、同じデータセットに対する 2 つの分類器の予測結果を比較するための検定手法である。この検定により、提案手法による予測性能の改善が偶然ではなく、統計的に有意であることを確認可能である。

■交差検証　モデルの汎化性能を評価するため、交差検証（Cross-Validation）を用いる。この手法はデータを k 個のサブセットに分割して評価を行うものであるが、本研究の評価においては 10 分割交差検証を採用する。交差検証の手順は以下の通りである。

1. データセットを 10 個のグループに分割し、#1 から#10 の番号を付与する。
2. 偏りの影響を軽減するため、以下の処理を $k = 1$ から 10 まで繰り返す。
   （a）グループ#k を検証データ、それ以外の 9 個を訓練データとしてモデルを構築・評価する。

これにより、全てのデータが一度ずつテストに使用され、データ分割の偏りに依存しない頑健な性能評価が可能になる。具体的な実装と評価手順については第 4 章で述べる。

3.4　レビュー労力を考慮した欠陥予測の改善

実際の開発現場では、レビューに費やせる労力は少ない。本節では、少ないレビュー労力でより多くの欠陥を検出するための手法を提案する。

■従来手法の問題点　従来のレビュー優先度付け手法では、欠陥予測モデルの予測値（0 あるいは 1）と変更行数に基づいてレビュー対象のコミットを順位付けし、上位のコミットから順にレビューを行う。[10]。しかし、この手法には課題が 2 つある。

まず、レビュー労力の推定方法が不正確である。既存手法では変更行数のみからレビュー労力を計算しているが、実際のレビュー労力は変更の複雑度にも依存する。例えば、10 個のファイルに分散した 100 行の変更は、1 個のファイルに集中した 100 行の変更よりもレビュー労力が大きい。なぜなら、変更が複数のファイルにまたがる場合、ファイル間の整合性を確認したり、モジュール同士の関係性を理解したりするために、より多くの労力が必要になるためである。

次に、レビュー労力の制約の設定が不適切である。従来手法では全てのコミットのレビュー労力の和が総労力として設定されているが、数千行の変更を含む巨大なコミットが存在する場合、この設定はレビューの実態を反映しない。なぜなら、実際の開発現場では、レビューに使える労力には実質的な上限があり、巨大なコミットは分割されることが多いためである。

■レビュー対象の優先順位付け　本研究では、レビュー対象の選択を組み合わせ最適化問題として扱う。これにより、従来の順位付け手法では扱えない制約条件を組み込める。

レビュー対象コミットの選択は、以下の特性を持つ。

- レビューに費やせる労力（制約条件）の範囲内で
- 各コミットのレビューに必要な労力（コスト）を考慮しながら
- 欠陥発見期待値（価値）の合計を最大化する

したがって、レビュー対象コミットの選択は、「容量上限のある袋に価値と重さを持つ複数のアイテムを入れるとき、総重量が容量を超えないように総価値を最大化すること」であり、組み合わせ最適化問題に置き換えられる。このことから、レビュー対象コミットの選択を、限られたリソース（総労力）の中で欠陥発見の期待値を最大化する問題として考える。

具体的には、レビューを待つ各コミットに対して、レビューを実施するか否かの二値の意思決定を行う。この際、選択された全てのコミットのレビュー労力の合計が、あらかじめ設定した総労力の上限を超えないという制約条件を設ける。この制約の下で、選択されたコミットに含まれる欠陥混入確率（期待値）の総和が最大となるような組み合わせを特定することを目指す。

これにより、単純な順位付けでは考慮が難しい「少ない労力でいかに多くの欠陥混入リスクをカバーするか」というリソース配分の問題を、組み合わせ最適化問題として扱うことが可能になる。

■レビュー労力の計算　変更の規模と複雑度を考慮した上でレビュー労力を計算する。

まず、コミット i におけるコードの変更行数 C_i を、追加行数 LA_i と削除行数 LD_i の合計として定義する。次に、変更の広がり H_i を、以下の情報理論のエントロピーにより定量化する。

$$H_i = -\sum_{k=1}^{n_i} p_k \log_2 p_k$$

ここで、n_i はコミット i で変更されたファイル数であり、p_k は全ての変更行数に対する各ファイル k の変更行数の割合である。このエントロピーを $\log_2 n_i$ で割ることで正規化し、正規化された変更の広がり $\bar{H}_i$ を得る。

これらを用いて、補正済みレビュー労力 W_i を次のように算出する。

$$W_i = \log_2(C_i \times N_i^{\bar{H}_i} + 1)$$

この対数変換により、巨大なコミットの影響が緩和され、中小規模のコミットも適切に評価することが可能になる。

最後に、各コミットのレビューの優先順位を決定する指標として、単位労力あたりの欠陥混入確率を表す密度 D_i を求める。

$$D_i = \frac{\hat{y}_i}{W_i}$$

■貪欲法による求解　組み合わせ最適化問題を解くため、本研究では貪欲法を採用する。動的計画法は最適解を求められるが、計算量が $O(NC_{total})$ であり、容量 C_{total} が大きい場合は実用的ではない。一方、貪欲法は計算量が $O(N \log N)$ であり、容量に依存せず比較的短い時間で解を得られる。

貪欲法では、各コミット i の密度 D_i（単位労力あたりの欠陥発見期待値）を計算し、密度の高い順にコミットを選択する。アルゴリズムは以下の通りである。

1. 全コミットについて密度 D_i を計算する
2. 密度の降順にコミットをソートする
3. 累積労力 $W_{\text{total}} = 0$ とする
4. ソートされた順にコミットを選択し、以下を実行する：
   - $W_{\text{total}} + W_i \leq C_{total}$ であれば、コミット i をレビュー対象に追加し、$W_{\text{total}} \leftarrow W_{\text{total}} + W_i$ とする
   - そうでなければ、コミット i をスキップする
5. 累積労力が容量を超えるまで、または全てのコミットを検討するまで繰り返す

■レビュー労力の上限設定　実際にレビューに費やせる労力を考慮した上で、容量 C_{total} を設定する必要がある。本研究では、労力が小さい順に並べたときの上位 80% のコミットの労力の和を C_{total} とする。

1. 全コミットをレビュー労力の昇順にソートする

2. 上位 80% のコミットを選択する
3. 選択されたコミットのレビュー労力の和を C_{total} とする

この設定により、レビュー労力が非常に大きいコミットを除外することで、実際の開発現場におけるレビュー労力の制約を反映する。80% というしきい値は実験的に決定したものであり、プロジェクトの特性に応じて変更可能である。

■評価方法　レビュー労力に対する欠陥発見数の累積曲線を用いて、提案手法の効果を評価する。この累積曲線は、横軸に「レビューに費やした労力」を配置し、縦軸に「発見した欠陥の数」を配置したものである。

提案手法では、密度の高い順にコミットを選択してレビューする。各コミットをレビューするごとに、レビューに費やした労力の合計と発見した欠陥の数の合計を記録し曲線を描く。比較対象として、既存手法のモデル（変更メトリクスを追加する前のモデル）の予測確率を用いて同様に貪欲法を適用する。

提案手法が既存手法よりも左上に位置する曲線を描く場合、同じレビュー労力でより多くの欠陥を発見可能であることを意味し、実際のレビューにおいて、提案した特徴量が欠陥発見に役立つ可能性があると示唆される。

第 4 章

実験環境

4.1　汎用性検証のための対象選定

■欠陥予測モデル構築に求められるデータの要件　本研究が目的とする、コミット間の不規則な変化を捉える欠陥予測モデルの構築には、従来の静的なデータ構造とは異なる、時系列的な変化を表現できるデータが求められる。本研究における欠陥予測モデル構築に求められるデータの要件は大きく分けて 3 つある。データの要件を図 4.1 に示す。

1 つ目は、ソフトウェアの一時的な構造を表すデータではなく、ソフトウェアの変更プロセスを追跡できるデータ構造であることである。例えば、特定の時点におけるメトリクスのみを提供する形式では、コードが絶えず変化する現代の開発現場における「構造の変

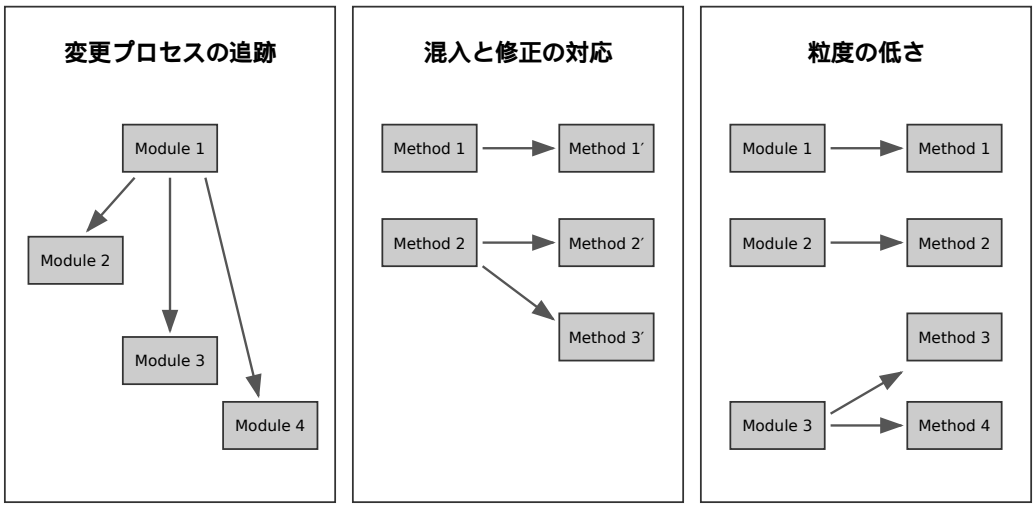


図 4.1: データ構造の決定に使用する構成変更の分類

化」という特徴を十分に捉えられない。

2 つ目は、欠陥の混入から修正までの対応関係が明確であることである。単なる欠陥の有無に加えて、欠陥が混入・修正されたコミットの位置（コミット ID）が分からなければ、直前の状態からの差分を学習して欠陥混入リスクを検出するモデルを構築することは困難である。

3 つ目は、データの粒度が低いことである。例えば、ファイル単位ではなくメソッド単位のメトリクスを保持していれば、局所的な変更が全体に及ぼす影響をより具体的に評価できる。

■データセットの選定　前述したデータ要件に照らし合わせると、欠陥予測の研究で古くから用いられてきた NASA MDP や PROMISE データセットは、特定の時点における構造的メトリクスのみを提供しており、欠陥混入のタイミングという時系列情報が欠落している。また、近年広く利用されている Defects4J[15] は、欠陥修正の直前・直後の状態に特化しており、欠陥がどのコミットで混入したかを遡って分析する用途には適さない。そこで本研究では、これらの課題を解決し、欠陥混入と欠陥修正のコミットをペアで特定している BugHunter データセット [16] を採用する。このデータセットは本研究の分析対象であるメソッド単位のメトリクスをあらかじめ保持しており、前処理のコストを抑えた詳細な分析が可能であるだけでなく、GitHub 上の実際の Java プロジェクトに基づいているため、CI/CD などの現代的な開発プロセスを反映している。

BugHunter データセットは、その公開以降、多くの欠陥予測研究で活用されている。Modanlou らは、BugHunter データセットの全 15 プロジェクトをメソッドレベルで使用し、ディープニューラルネットワークを提案することで、平均 F1 スコア 84.08% を達成した [17]。Jász は、メソッドレベルでの隠れ依存関係メトリクスの有効性を検証し、従来のソフトウェアメトリクスと比較して F1 スコアが 11% 改善されることを示した [18]。これらの先行研究は、BugHunter データセットがメソッドレベルでの欠陥予測研究において有用であることを示している。

■対象プロジェクトの選定　BugHunter データセットに含まれる 15 のプロジェクトの中から、統計的検定力を確保するため、データセットのレコード数が多いプロジェクトを優先的に選定する。

データセットの構築には SZZ アルゴリズム [19] が用いられているため、解決済みのバグレポートの数とデータセットのレコード数の大きさには相関関係がある。そのため、解決済みのバグレポートの数を基準として、データセットの相対的な大きさを推測可能である。この基準に基づき、解決済みのバグレポートの数が多い上位 5 つのプロジェクトを対

象として選定した。

選定するプロジェクトの数を 5 つとした理由は、統計的検定力の確保とデータの多様性の維持を両立するためである。プロジェクト数が極めて限定的である場合、プロジェクト間の特性の差異を十分に網羅できず、提案手法の汎化性能を客観的に評価することが困難になる。本研究では、選定数を 5 つとすることで、複数の異なるドメインをカバーしつつ、McNemar 検定などの統計的仮説検定において十分なサンプルサイズを確保する。これにより、得られた結果の統計的な有意性が担保され、手法の有効性について、より信頼性の高い議論を行えるようになる。

表 4.1: 対象プロジェクトとバグレポート数

プロジェクト	ドメイン	バグレポート数
Elasticsearch[20]	分散検索・分析エンジン	4,287
Hazelcast[21]	分散コンピューティング	3,762
Netty[22]	ネットワークフレームワーク	2,207
OrientDB[23]	マルチモデル DB	1,272
Neo4j[24]	グラフ DB	1,152

Elasticsearch は、分散検索・分析エンジンであり、大規模なログデータやテキストデータの検索に広く使用されている。Hazelcast は、インメモリデータグリッドを提供する分散コンピューティングプラットフォームである。Netty は、高性能な非同期イベント駆動型のネットワークアプリケーションフレームワークである。OrientDB は、マルチモデルデータベースであり、グラフデータベースとドキュメントデータベースの機能を併せ持つ。Neo4j は、グラフデータベースの代表的な実装の一つであり、ソーシャルネットワーク分析や推薦システムなど、関係性を重視するアプリケーションで広く使用されている。

これらは、いずれも Java で記述されている活発な OSS プロジェクトである。ドメインは検索エンジン、分散システム、ネットワークフレームワーク、データベースと多岐にわたり、それぞれのプロジェクトが異なる開発特性を持つため、提案手法が様々なプロジェクトにおいて有効であることを検証可能である。

■正解ラベルと構成要素の相違　本研究では、メソッド単位での欠陥予測を行う。なぜなら、メソッド単位の予測では構成要素の大きさが比較的小さく、欠陥の特徴を捉えやすいためである。実際に、BugHunter データセットを用いた先行研究において、メソッド単位のデータセットを用いた場合に最も予測精度が高くなることが示されている。

メソッド単位での欠陥予測を行うため、各メソッドに含まれる欠陥の数を二値化して正解ラベルとする。具体的には、欠陥の数が 0 のメソッドを「欠陥を含まないデータ」、欠陥の数が 1 以上のメソッドを「欠陥を含むデータ」に分類する。この二値分類により、ランダムフォレストのような分類アルゴリズムを適用可能な形式にデータを整形する。

■データの前処理と使用レコード数　データの前処理として、以下の処理を実施する。まず、値が全て同じであるカラムを削除する。なぜなら、これらのカラムは予測に寄与しないためである。次に、メソッドの識別子をベクトルに変換する。なぜなら、メソッド名やクラス名といった識別子をそのまま用いると、各識別子を独立したカテゴリーとして扱うことになり、未知の識別子に対する汎化性能が低下するためである。メソッドの識別子をいくつかの要素に分解することで、それらの識別子を要素の類似性を表す特徴量に変換し、複数のメソッドが特定の共通要素を持つことを機械学習モデルに認識させることが可能である。

交差検証時の評価指標の安定化を図るため、各プロジェクトのデータセットから抽出するレコード数の上限を 5,000 行とする。10 分割交差検証においては、5,000 行のレコードを各サブセット（約 500 行）に分配することで、不均衡データであっても少数クラスのサンプルを十分に確保することが可能である。これにより、交差検証の各フェーズにおける F1 スコアの変動を抑制し、評価指標の安定性を高めることが可能である。

データ抽出の際、利用可能なデータが 5,000 行以下の場合は全てのデータを使用し、5,000 行を超える場合は最初の 5,000 行を使用する。データセット内のレコードは、必ずしも時系列順に並んでいるわけではないが、同一メソッドの複数のバージョン（欠陥混入時と修正時）が含まれており、これらの情報を用いて変化量を計算することが可能である。

4.2　コミット間変化の機械学習可能な形式への変換

3.2 節で定義した変更メトリクスを実際のデータセットに適用するため、以下の実装を行う。

■メソッド単位の変更メトリクスの計算　メソッド単位の変更メトリクスとして、コード行数の変化量、トークン数の変化量、循環的複雑度の変化量を算出する。これらは、同一メソッドの欠陥混入時あるいは欠陥修正時とその直前のコミットの値の差分から計算される。

具体的には、BugHunter データセットに含まれる各メソッドについて、欠陥混入コミットまたは欠陥修正コミットにおける値と、その直前のコミットにおける値を取得し、以下

の計算を行う。

- コード行数の変化量 = 現在のコミットにおけるコード行数 - 直前のコミットにおけるコード行数
- トークン数の変化量 = 現在のコミットにおけるトークン数 - 直前のコミットにおけるトークン数
- 循環的複雑度の変化量 = 現在のコミットにおける循環的複雑度 - 直前のコミットにおける循環的複雑度

これらのメソッド単位の変更メトリクスは、メソッドがどの程度変更されたかを表現する指標となる。

■コミット単位の変更メトリクスの計算　コミット単位の変更メトリクスとして、変更されたファイル数、コードの追加行数、コードの削除行数、変更の広がりを生成する。これらは、BugHunter データセットに含まれるコミット情報から以下のように計算される。

- 変更されたファイル数: 現在のコミットと直前のコミットの間の変更されたファイル数
- コードの追加行数: 現在のコミットと直前のコミットの間のコードの追加行数
- コードの削除行数: 現在のコミットと直前のコミットの間のコードの削除行数
- 変更の広がり: $\bar{H}_i$

これらのコミット単位の変更メトリクスは、コミット全体における変更の規模や複雑度を捉える。

■メソッドの操作タイプのラベル付与　メソッドの操作タイプ（追加、変更、削除）は、欠陥混入リスクと密接に関係している。新たに追加されたメソッドは既存のコードとの間で予期しない相互作用を引き起こす可能性があり、既存のメソッドの変更は意図しない副作用を生む可能性がある。先行研究において、新たに追加されたコードは既存のコードと比較して欠陥密度が高い傾向があることが示されている。

各メソッドに対して、追加、変更、削除のいずれの操作が行われたかを示すラベルを付与する。これらの操作タイプは、欠陥混入コミットあるいは欠陥修正コミットと、その直前のコミットを比較することで判定される。

- 追加: 直前のコミットには存在せず、そのコミットで新たに追加されたメソッド
- 変更: 直前のコミットとそのコミットの両方に存在し、内容が変更されたメソッド

- 削除: 直前のコミットには存在したが、そのコミットで削除されたメソッド

これらのラベルを One-Hot エンコーディングを用いて数値ベクトルに変換し、カテゴリカル変数として扱う。例えば、「追加」は [1, 0, 0]、「変更」は [0, 1, 0]、「削除」は [0, 0, 1] のようにエンコードされる。

■メソッドの識別子の処理　メソッドの識別子を特徴量として用いる理由は、識別子に含まれる、類似性に関する特徴から欠陥混入リスクを推測するためである。メソッド名やクラス名には通常、機能を示す単語が含まれる。例えば”parse”、”serialize”、”validate” といった単語を含むメソッドは、入力検証やデータ変換に関わるため、エッジケースの考慮不足による欠陥が発生しやすい。また、パッケージ構造も重要な情報を提供する。特定のパッケージ（例えば、ネットワーク処理やデータベース接続）に属するメソッドは、外部モジュールとの相互作用に起因する欠陥が発生しやすい。

メソッドの識別子をそのまま特徴量として扱うと、メソッド間のテキストの類似性を捉えられない。例えば、‘getUserName()‘と ‘getUserAge()‘は、どちらも「get」と「user」という共通要素を持ち、ユーザー情報を取得するという類似した機能を持つが、識別子全体を独立したカテゴリーとして扱うと、この類似性を認識できない。さらに、訓練データに存在しない新しいメソッド名に対しては、全く予測ができなくなる。

この問題を解決するため、メソッドの識別子をいくつかの要素に分解する。具体的には、トークン分割により、‘getUserName()‘は [”user”, ”name”] に、‘getUserAge()‘は [”user”, ”age”] に分解され、両者が「user」という共通要素を持つことを認識可能になる。これにより、モデルは「user に関連するメソッド」という抽象的な特徴を学習でき、訓練データに存在しない ‘getUserEmail()‘のような新しいメソッドに対しても、[”user”, ”email”] というトークンから適切に予測することが可能になる。このトークン分割では以下の処理を実行する。

1. 正規表現を用いてメソッドの完全修飾名からパッケージ名、クラス名、メソッド名を抽出する
2. パッケージ名を. で分割し、各部分をトークンとする
3. クラス名を$で分割し、各部分をキャメルケース分割する（例: ”UserManager” → [”User”, ”Manager”]）
4. メソッド名をスネークケース（_や-）で分割した後、さらにキャメルケース分割する（例: ”get_user_name” → [”get”, ”user”, ”name”]）
5. <init>や<clinit>などの特殊なメソッド名は”constructor” というトークンに変換

する

6. 全てのトークンを小文字に変換する
7. 高頻度で出現するトークンと指定文字数未満のトークンを除外する

表 4.2 に、実際のメソッドの完全修飾名に対するトークン分割の例を示す。

表 4.2: メソッド識別子のトークン分割の例

処理ステップ	結果
元のメソッドの完全修飾名	com.example.user.UserManager .getUserName()Ljava/lang/String;
パッケージ分割	[”com”, ”example”, ”user”]
クラス名のキャメルケース分割	[”User”, ”Manager”]
メソッド名のキャメルケース分割	[”get”, ”User”, ”Name”]
小文字化	[”com”, ”example”, ”user”, ”user”, ”manager”, ”get”, ”user”, ”name”]
高頻度で出現するトークンの除去	[”example”, ”user”, ”manager”, ”user”, ”name”]
重複するトークンの除去	[”example”, ”user”, ”manager”, ”name”]

この例では、”com” と”get” が高頻度で出現するトークンとして除外され、”user” は複数回出現するが意味的に重要な単語として保持される。最終的に [”example”, ”user”, ”manager”, ”name”] という、メソッドの機能を表す有用なトークンが抽出される。

表 4.3 に、異なる種類のメソッドに対するトークン分割の比較例を示す。

表 4.3: 異なるメソッドタイプのトークン分割比較

メソッドタイプ	元のメソッド名	抽出されるトークン
データ検証	validateEmail()	[”validate”, ”email”]
JSON 解析	parseJsonData()	[”parse”, ”json”, ”data”]
データベース接続	connectToDatabase()	[”connect”, ”database”]
HTTP 通信	sendHttpRequest()	[”send”, ”http”, ”request”]
コンストラクター	<init>()	[”constructor”]

頻繁に出現するトークンとして除外される単語は、以下の通りである。

- Java パッケージ名: ”java”, ”util”, ”lang”, ”io”, ”net”, ”org”, ”com”, ”javax”
- メソッドの接頭辞: ”get”, ”set”, ”is”, ”has”
- 動詞: ”create”, ”build”, ”make”, ”run”, ”execute”
- Java の予約語や型名: ”class”, ”interface”, ”void”, ”int”, ”string”
- 修飾語: ”impl”, ”default”, ”base”, ”simple”, ”empty”

これらのトークンは、多くのプロジェクトで頻出するため識別能力が低く、予測モデルにノイズをもたらす。一方、”validate”、”parser”、”socket” といった単語は、メソッドの具体的な機能を示し、欠陥予測に有用である。

抽出されたトークンは、TF-IDF（Term Frequency-Inverse Document Frequency）を用いて数値ベクトルに変換される。TF-IDF は、各トークンの重要度を、文書内での出現頻度と全文書での出現頻度の逆数の積として計算する。これにより、特定のメソッドに特徴的なトークンに高い重みが付与され、多くのメソッドに共通するトークンの重みは抑制される。この処理により、メソッドが属するパッケージやクラスの名前、メソッド名自体が持つ類似性に基づく情報を特徴量として活用可能である。

■異なる構成要素の特徴量の活用　最終的に、以下の特徴量を組み合わせてモデルの入力とする。

- BugHunter データセットに元々含まれる構造的メトリクス
- メソッド単位の変更メトリクス
- コミット単位の変更メトリクス
- メソッドの操作タイプ
- メソッドの識別子に含まれる特徴的な要素

■データ前処理とメトリクス計算の実装手順　前述した特徴量を実際のデータセットに適用するため、データの前処理とメトリクスの計算を実施する。データセット構築のために実行した一連のプログラムとその処理内容を表 4.4 に示す。

まず、BugHunter データセットから取得した生データに対して前処理を行う。drop_columns.py を用いて全ての値が 0 であるカラムを削除し、drop_rows.py を用いて分析対象外のレコード（例: 外部ライブラリに属するメソッド）を削除する。

前処理後のデータに対して、メソッド単位の変更メトリクスを追加する。add_method_level_metrics.py を実行し、各メソッドについて欠陥混入コミットあるいは欠陥修正コミットとその直前のコミットにおけるコード行数、トークン数、循環

表 4.4: データセット構築に使用したプログラムとその処理内容

プログラム名	処理内容
drop_columns.py	全ての値が 0 であるカラムを削除する
drop_rows.py	分析対象外のレコード（外部ライブラリに属するメソッドなど）を削除する
add_method_level_metrics.py	メソッド単位のメトリクスを取得し、直前のコミットとの差分を計算する
add_commit_level_metrics.py	コミット単位のメトリクスを取得し、直前のコミットとの差分を計算する

的複雑度を取得し、これらの差分を計算する。

続いて、コミット単位の変更メトリクスを追加する。add_commit_level_metrics.py を実行し、各コミットについて変更されたファイル数、コードの追加行数、コードの削除行数、変更の広がりを取得し、これらの差分を計算する。これらのメトリクスは、Git リポジトリから GitPython ライブラリを用いて計算される。変更の広がりについては、Hassan の手法 [5] に基づき、情報理論のエントロピーを用いて変更の分散度を求める。

これらのプログラムは順次実行され、各ステップの出力が次のステップの入力となる。具体的には、drop_columns.py → drop_rows.py → add_method_level_metrics.py → add_commit_level_metrics.py の順に実行することで、最終的なデータセットが構築される。これらのプログラムを含むリポジトリは参考文献に記載されている [25]。

4.3 提案手法の効果を検証する比較モデルの準備

3.3 節で述べたランダムフォレストを用いて、欠陥予測モデルを構築する。本節では、ランダムフォレストの実装とモデル構築に必要な準備について述べる。

■ランダムフォレストの実装 ランダムフォレストは、複数の決定木を構築し、それらの予測結果を集約することで高い予測精度と汎化性能を実現するアンサンブル学習手法である。

図 4.2 に決定木の構造を示す。決定木は、各ノードにおける特徴量のしきい値に基づいてデータを分割し、葉ノードで最終的な予測クラスを出力する。しかし、単一の決定木は訓練データを過学習しやすく、汎化性能が低下する傾向がある。

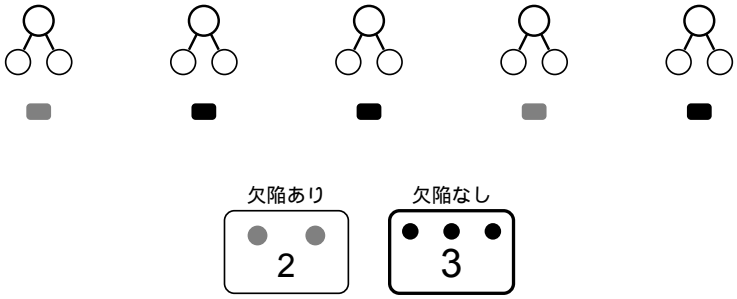


図 4.2: 決定木の構造

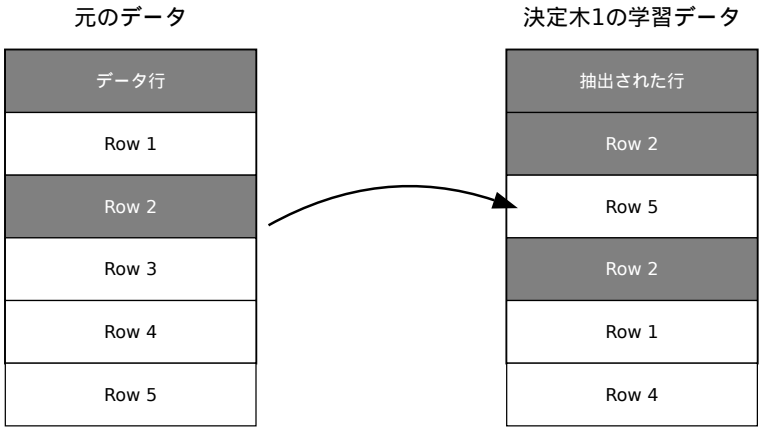


図 4.3: ブートストラップサンプリングによる訓練データの生成

ランダムフォレストでは、この問題を解決するためにブートストラップサンプリングを用いる。図 4.3 に示すように、元の訓練データセットから復元抽出により複数の異なるサンプルを生成し、各サンプルに対して独立に決定木を学習する。

図 4.4 に、ランダムフォレストによる分類の流れを示す。各決定木は独立に予測を行い、分類問題では多数決により最終的な予測クラスが決定される。また、各分岐点ではランダムに選択された特徴量の部分集合のみが使用されるため、決定木間の相関が低減され、モデルの汎化性能が向上する。

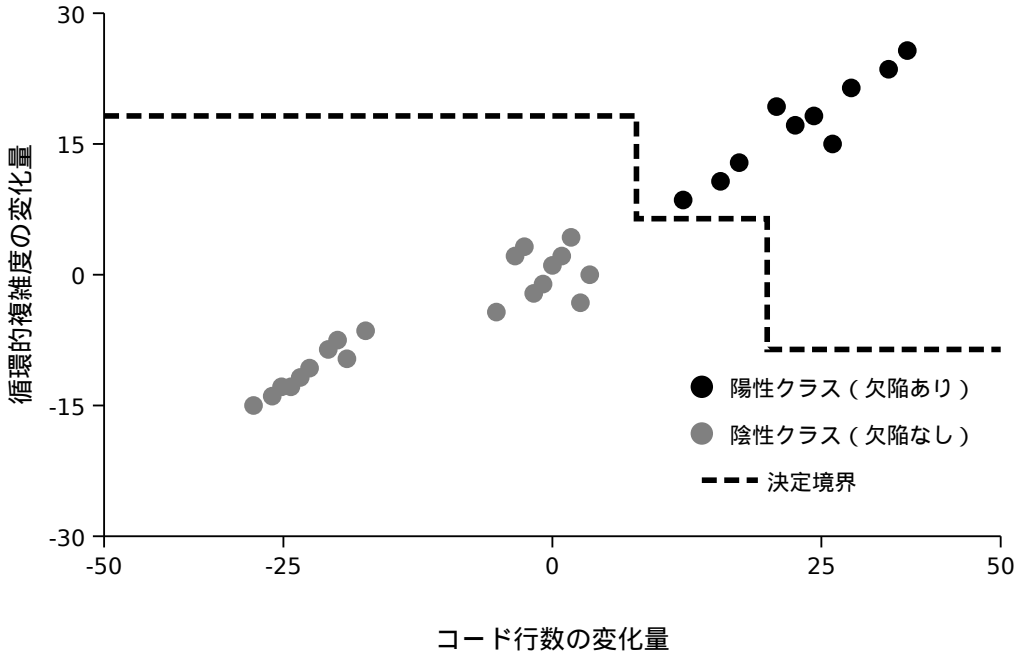


図 4.4: ランダムフォレストによる分類

■モデル構築の段階的アプローチ　提案する特徴量の有効性を検証するため、段階的にモデルを構築する。各ステップで特徴量を追加することにより、その特徴量群が予測性能にどの程度寄与するかを定量的に評価可能である。

以下の 3 段階でモデルを構築する。

ステップ 1（既存手法）：BugHunter データセットに元々含まれている構造的メトリクスのみを特徴量として使用する。このモデルを既存手法のモデル（後続のモデルとの比較基準）とする。

ステップ 2: 既存手法のモデルに対して、メソッド単位の変更メトリクスを追加する。これにより、メソッド単位の時系列変化を考慮することの効果を評価する。

ステップ 3（提案手法）：ステップ 2 のモデルに対して、さらにコミット単位の変更メトリクスを追加する。これにより、局所的な観点と全体的な観点の分析を組み合わせた場合の効果を評価する。

この段階的評価により、1）時系列変化を考慮することで予測性能は向上するか、2）メソッド単位とコミット単位のメトリクスを活用することで、さらなる改善が得られるか、3）各特徴量群の相対的な重要性はどの程度かが明らかになる

x1	x2	x3	予測確率
10	5	A	0.20
10	8	B	0.35
10	2	A	0.15

平均: 0.23

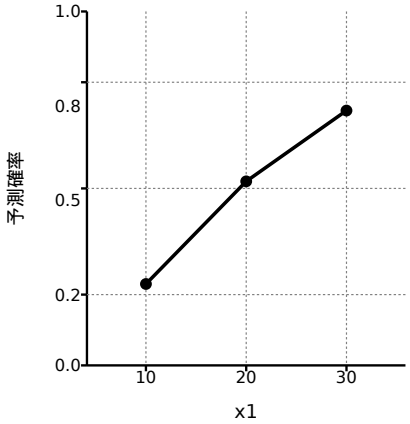


図 4.5: Partial Dependence Plot の計算方法

■モデルのクラス分類の分析手法　モデルのクラス分類を理解するため、以下の分析手法を準備する。

特徴量重要度: ランダムフォレストが提供する特徴量重要度は、各特徴量が決定木の分岐においてどの程度情報利得をもたらしたかを示す指標である。この値により、どの特徴量が欠陥予測に重要であるかを定量的に評価可能である。

Partial Dependence Plot（PDP）: PDP は、特定の特徴量の値を変化させたときのモデルの予測値の平均的な変化を可視化する手法である。図 4.5 に示すように、特徴量 x_s に対する Partial Dependence 関数は、他の特徴量の値を固定した状態で x_s を変化させた際の予測値の平均として計算される。

特徴量 x_s に対する Partial Dependence 関数 $f_{\mathrm{PD}}(x_s)$ は以下のように定義される。

$$f_{\mathrm{PD}}(x_s) = \frac{1}{n}\sum_{i=1}^{n} \hat{f}(x_s, x_{-s}^{(i)})$$

ここで、$\hat{f}$ は学習されたモデル、$x_{-s}^{(i)}$ は i 番目のサンプルにおける特徴量 x_s 以外の特徴量の値を表す。

決定木の可視化: ランダムフォレストを構成する決定木の一つを可視化することで、どのような分類条件で欠陥の有無を判断しているかを確認することが可能である。

4.4　労力制約下での評価指標の設計

3.4 節で議論した手法に基づいて、レビュー労力を計算する。本節では、実装の詳細とデータセットへの適用について述べる。

■レビュー労力の計算　各コミット i のレビュー労力を算出するため、以下の手順を実行する。

まず、コミットの基本情報として、追加されたコード行数 LA_i、削除されたコード行数 LD_i、および変更されたファイル数 N_i を取得する。レビュー労力の基本単位となるコードの全ての変更行数 C_i は、これらを用いて $C_i = LA_i + LD_i$ として算出される。

次に、変更の広がり H_i を、以下の情報理論のエントロピーを用いて計算する。

$$H_i = -\sum_{k=1}^{n_i} p_k \log_2 p_k$$

ここで、n_i はコミット i で変更されたファイル数であり、p_k は全変更行数に対する各ファイル k の変更行数の割合を表す。この H_i を $\log_2 n_i$ で割ることで正規化した値を $\bar{H}_i$ とし、これを変更の複雑度の重みとして用いる。

最終的なレビュー労力は、変更行数 C_i に変更の複雑度を反映させた値を対数変換することで算出する。本研究では、エントロピーの計算（ビット単位の評価）と理論的な一貫性を持たせるため、底に 2 を用いた二進対数（$\log_2$）を採用する。

$$W_i = \log_2(C_i \times N_i^{\bar{H}_i} + 1)$$

この対数変換により、数千行を超えるような巨大なコミットの影響を緩和しつつ、中小規模のコミット間の相対的な労力の相違を適切に評価することが可能になる。

■レビュー総労力の設定　3.4 節で定義した方法に従い、レビューに使える総労力 C_{total} を設定する。全コミットをレビュー労力の昇順にソートし、上位 80% のコミットの労力の和を C_{total} とする。

■密度の計算　3.4 節で定義した貪欲法による優先度付けを行うため、以下の式により密度 D_i を計算する。

$$D_i = \frac{\hat{y}_i}{W_i}$$

ここで、$\hat{y}_i$ はモデルが予測したコミット i の欠陥混入確率であり、W_i は前述の手順で算出した補正済みレビュー労力である。この密度 D_i は、単位労力あたりの欠陥発見期待値を表し、レビュー対象コミットの優先順位付けに用いられる。

■データセットへの適用　BugHunter データセットに含まれる各コミットについて、上記の手順に従ってレビュー労力と密度 D_i を算出する。

■統計的仮説検定　提案手法の有効性を統計的に検証するため、以下の 2 つの検定を用いる。

McNemar 検定: 二値分類の予測性能（F1 スコア）について、提案手法と既存手法の間に統計的に有意な差があるかを検証する。McNemar 検定は、対応のある 2 つの分類器の性能を比較する際に広く用いられ、各サンプルに対する 2 つのモデルの予測結果の不一致度を分析する。

Wilcoxon の符号順位検定: レビュー労力 20% と 40% の時点における欠陥発見率について、提案手法が既存手法より統計的に有意に優れているかを検証する。5 つのプロジェクトを独立したサンプルとして扱い、各プロジェクトにおけるそれぞれの手法の欠陥発見率のペアを比較する。サンプルサイズが 5 と小さく、データの正規性が保証されないため、ノンパラメトリック検定である Wilcoxon の符号順位検定が適切である。本研究の目的は「提案手法が既存手法より優れている」ことの検証であるため、片側検定を採用する。

すべての統計的仮説検定において、有意水準は 0.05 とする。

第 5 章

実験結果

5.1　評価手順

本研究では、提案する変更メトリクスの各要素が予測性能に与える影響を明確にするため、段階的な評価を実施する。本節では、評価手順を順に述べる。

■データ分割　まず、各プロジェクトのデータセットを訓練データ 80%、テストデータ 20% に分割する。訓練データはモデルの学習に使用し、テストデータは最終的な性能評価に使用する。テストデータは訓練過程で一切使用しないため、未知データに対する汎化性能を測定可能である。

■10 分割交差検証　次に、訓練データに対して 10 分割交差検証を実施し、モデルの安定性を評価する。交差検証は、データセットを複数のサブセットに分割し、それぞれのサブセットを訓練データあるいは検証データとして交互に用いる手法である。具体的な手順は以下の通りである。

1. データセットを 10 個のグループに分割し、#1 から#10 の番号を付与する。
2. 偏りの影響を軽減するため、以下の処理を $k = 1$ から 10 まで繰り返す。
   (a) グループ#k を検証データ、それ以外の 9 個を訓練データとしてモデルを構築し、F1 スコア、適合率、再現率を計算する。

交差検証で得られた全ての評価結果の平均値と標準偏差を算出する。標準偏差が小さいほど、データの分割方法に依存しにくい安定したモデルであることを示す。

■評価指標　評価指標として、3.3 節で定義した適合率、再現率、F1 スコア、および AUC（Area Under the Curve）を用いる。適合率は欠陥を含むと予測したコミットのうち実際に欠陥を含む割合、再現率は実際に欠陥を含むコミットのうち正しく予測できた割合を示す。F1 スコアは適合率と再現率の調和平均であり、陽性クラス（欠陥を含むコミット）と陰性クラス（欠陥を含まないコミット）の不均衡が大きいデータセットにおいて、モデルの総合的な予測性能を評価するのに適している。AUC は分類器の性能をしきい値によらず評価する指標であり、1 に近いほど性能が高い。

■最終評価　交差検証により性能が確認されたモデルを、訓練データ全体で再学習する。その後、このモデルをテストデータに適用し、最終的な性能指標を算出する。

■段階的評価の実施　評価は以下の 3 段階で実施する。第 1 段階（既存手法、ステップ 1）では BugHunter データセットの元のメトリクスのみを使用する。これは構造的メトリクスを中心とした特徴量である。第 2 段階（ステップ 2）ではメソッド単位の変更メトリクスを追加する。これには、メソッドの変更タイプ（追加、削除、修正）やメソッド単位のコード行数の変化量などが含まれる。第 3 段階（提案手法、ステップ 3）ではコミット単位の変更メトリクスをさらに追加する。これには、コミットの変更ファイル数、追加・削除コード行数などが含まれる。各段階の評価結果を比較することで、メソッド単位とコミット単位の変更メトリクスがそれぞれどの程度性能向上に寄与するかを明らかにする。

■統計的有意性の検証　提案手法（ステップ 3）と既存手法（ステップ 1）の機械学習モデルの性能差が統計的に有意であることを、McNemar 検定により検証する。McNemar 検定は、同じテストデータに対する 2 つの分類器の予測結果を比較する検定手法である。この検定では、まず 2 つのモデルの予測結果から分割表を作成する。分割表では、既存手法で正分類であり提案手法で誤分類であったサンプル数を n_{12}、提案手法で正分類であり既存手法で誤分類であったサンプル数を n_{21} とする。検定統計量は以下で計算される。

$$\chi^2 = \frac{(n_{12} - n_{21})^2}{n_{12} + n_{21}}$$

有意水準 0.05 で、$p < 0.05$ の場合に性能差が統計的に有意であると判断する。

■レビュー労力削減効果の評価　提案手法によるレビュー労力削減効果を評価するため、レビュー労力に対する欠陥発見数の累積曲線を作成する。累積曲線は、横軸に投入したレビュー労力、縦軸に発見した欠陥数をプロットしたグラフである。提案手法の曲線が既存手法より左上に位置する場合、同じ労力でより多くの欠陥を発見可能であることを意味

する。

■貪欲法によるレビュー対象の選択　累積曲線を作成するため、各モデル（ステップ 1、ステップ 2、ステップ 3）について、貪欲法によりレビュー対象コミットを選択する。貪欲法は、各段階で局所的に最適な選択を行うアルゴリズムである。このアルゴリズムでは、まず各コミット i のレビュー労力を 4.4 節の方法で計算する。次に、全コミットをレビュー労力の昇順にソートし、上位 80% のコミットの労力の和を総労力 C_{total} として設定する。各コミット i について、モデルが予測した欠陥混入確率 $\hat{y}_i$ と補正済み労力 W_i から密度を計算する。

$$D_i = \frac{\hat{y}_i}{W_i}$$

密度 D_i の降順にコミットをソートする。累積労力 $W_{\text{total}} = 0$ とする。ソートされた順にコミットを検討し、$W_{\text{total}} + W_i \leq C_{total}$ であればコミット i をレビュー対象に追加し、$W_{\text{total}} \leftarrow W_{\text{total}} + W_i$ とする。そうでなければスキップする。累積労力が容量を超えるまで、または全コミットを検討するまで繰り返す。各コミットをレビューするごとに、累積レビュー労力と累積欠陥発見数を記録する。この手順により、少ないレビュー労力で欠陥発見期待値を最大化するレビュー対象を選択することが可能である。

■累積曲線の作成　累積曲線は以下の手順で作成する。各モデルについて、貪欲法によりレビュー対象コミットを選択する。各コミットをレビューする順に、累積レビュー労力と累積欠陥発見数を記録する。横軸を累積レビュー労力、縦軸を累積欠陥発見数として、各モデルの曲線を同一グラフ上に描画する。総労力の 20%, 40% 時点での欠陥発見数を比較する。

■評価指標の定義　評価指標として、欠陥発見率と改善幅を用いる。欠陥発見率は、特定労力時点までに発見した欠陥の数を全欠陥数で割った値であり、以下の式で定義される。

$$\text{欠陥発見率} = \frac{\text{発見した欠陥の数}}{\text{全欠陥数}} \times 100$$

改善幅は、提案手法（ステップ 3）と既存手法（ステップ 1）の欠陥発見率の差であり、以下の式で定義される。

$$\text{改善幅} = \text{提案手法の欠陥発見率} - \text{既存手法の欠陥発見率}$$

■特徴量重要度の算出　ランダムフォレストが提供する特徴量重要度を用いて、各特徴量の予測への寄与度を定量化する。特徴量重要度は、各特徴量が決定木の分岐においてどの程度情報利得をもたらしたかを示す指標である。情報利得は、ある特徴量で分岐することにより、データの不純度（通常、ジニ不純度で表される）がどの程度減少するかを測る指標である。ランダムフォレストでは、全ての決定木における各特徴量の情報利得の平均を計算することで特徴量重要度を算出する。本研究では、この指標を用いて、訓練済みの機械学習モデルから各特徴量の重要度を取得する。さらに、各特徴量を特徴量重要度の降順にソートする。上位の特徴量を抽出し、棒グラフで可視化する。これにより、メソッド単位、コミット単位、元のメトリクスという各カテゴリーの特徴量がどの程度重要であるかを分析する。

■Partial Dependence Plot の生成　各特徴量と欠陥混入確率の関係を可視化するため、Partial Dependence Plot（PDP）を生成する。PDP は、特定の特徴量の値を変化させたときに、モデルの予測値がどのように変化するかを示すグラフである。特徴量 x_s に対する Partial Dependence 関数 $f_{\mathrm{PD}}(x_s)$ を求めるには、まず特徴量 x_s の値を固定値に設定する。他の全ての特徴量 x_{-s} は、各サンプルの実際の値を使用する。全サンプルについて予測値を計算し、その平均を取る。

$$f_{\mathrm{PD}}(x_s) = \frac{1}{n}\sum_{i=1}^{n} \hat{f}(x_s, x_{-s}^{(i)})$$

特徴量 x_s の値を変化させながらこの計算を繰り返す。この結果、横軸を特徴量 x_s の値、縦軸を予測確率とするグラフが得られる。このグラフを用いることで、特定の特徴量が増加した際に欠陥混入確率がどのように変化するかを視覚的に理解することが可能になる。このグラフを生成するために、まず特徴量重要度が上位の特徴量を選択する。各特徴量の値の範囲を等間隔に分割する。各点において上記の計算方法で Partial Dependence 値を計算する。横軸を特徴量の値、縦軸を予測確率として、各特徴量の PDP をグラフ化する。

■決定木の可視化　ランダムフォレストを構成する決定木の構造を可視化し、どのような分類条件で欠陥の有無を判断しているかを確認する。決定木の構造を可視化するには、まずランダムフォレストから代表的な決定木を 1 つ選択する。決定木の各ノードにおける分岐条件（特徴量としきい値）を抽出する。各ノードのサンプル数とクラス分布を取得する。グラフ描画ライブラリを用いて、決定木を描画する。

5.2 提案手法による予測性能の向上

提案手法の予測性能を 3 段階で評価した。表 5.1 に、5 つのプロジェクトにおけるテストデータでの評価結果を示す。

表 5.1: テストデータによる評価

プロジェクト	モデル	F1 スコア	AUC
Elasticsearch	ステップ 1	0.575	0.775
	ステップ 2	0.708	0.880
	ステップ 3	**0.767**	**0.925**
Hazelcast	ステップ 1	0.678	0.875
	ステップ 2	0.733	0.900
	ステップ 3	**0.790**	**0.932**
Neo4j	ステップ 1	0.478	0.713
	ステップ 2	0.616	0.8300
	ステップ 3	**0.742**	**0.938**
Netty	ステップ 1	0.455	0.705
	ステップ 2	0.661	0.882
	ステップ 3	**0.747**	**0.934**
OrientDB	ステップ 1	0.483	0.738
	ステップ 2	0.524	0.765
	ステップ 3	**0.701**	**0.914**

全プロジェクトでステップを経るごとに性能が一貫して向上した。既存手法（ステップ 1）から提案手法（ステップ 3）への F1 スコアの改善幅は、Hazelcast の 0.11 から Netty の 0.29 まで分布し、平均改善幅は 0.21 であった。AUC は全プロジェクトで 0.91 を超え、高い識別性能を達成した。特に Neo4j と Netty では、既存手法の F1 スコアが 0.48 程度と低かったが、提案手法で 0.74 程度まで向上し、欠陥予測の実用性が大きく改善された。

ステップ 2 からステップ 3 への改善も全プロジェクトで確認され、コミット単位の変更メトリクスが予測に役立つ情報を提供することが示された。改善幅はプロジェクトによって異なり、Neo4j では 0.13、Netty では 0.09 であった一方、OrientDB では 0.18 と最大の改善を示した。これは、プロジェクトの特性によって、メソッド単位とコミット単位の

各メトリクスの価値が異なることを示唆している。

McNemar 検定により、提案手法と既存手法の性能差の統計的有意性を検証した。全プロジェクトで $p < 0.05$ となり、性能向上は統計的に有意であることが確認された。

■交差検証による安定性　10 分割交差検証により、データの分割に依存しない安定性を評価した。表 5.2 に結果を示す。

表 5.2: 10 分割交差検証の結果（平均値 ± 標準偏差）

プロジェクト	モデル	F1 スコア	AUC
Elasticsearch	ステップ 3	0.791 ± 0.024	0.931 ± 0.016
Hazelcast	ステップ 3	0.783 ± 0.019	0.931 ± 0.011
Neo4j	ステップ 3	0.739 ± 0.033	0.936 ± 0.011
Netty	ステップ 3	0.708 ± 0.028	0.925 ± 0.016
OrientDB	ステップ 3	0.664 ± 0.025	0.899 ± 0.019

各プロジェクトで F1 スコアの標準偏差は 0.02 から 0.03 程度、AUC の標準偏差は 0.01 から 0.02 程度に収まっており、提案手法はデータの分割方法に大きく依存せず安定した性能を発揮することが確認された。

5.3　提案したメトリクスの予測寄与度の確認

特徴量重要度を分析した結果、提案した変更メトリクスが予測に大きく寄与することが確認された。全プロジェクトで共通して、コミット単位の変更メトリクスである `lines_added`（追加行数）と `num_files`（変更ファイル数）が上位を占めた。これらは変更規模を表す指標であり、欠陥混入リスクの主要な予測因子となっている。また、`entropy`（変更の広がり）も多くのプロジェクトで重要な特徴量として機能し、変更が特定箇所に集中しているか、複数箇所に分散しているかが予測に影響することが示された。

メソッド単位の変更メトリクスも重要な予測情報を提供する。特に Netty では `tokens_change`（トークン数の変化量）が最も重要な特徴量となった。一方、Elasticsearch や Hazelcast では `lines_added` や `num_files` といったコミット単位のメトリクスが上位を占める傾向が見られた。この相違は、プロジェクトのドメインや開発特性を反映していると考えられる。

図 5.1 に、代表例として Neo4j プロジェクトにおける特徴量重要度を示す。Neo4j

では lines_added（追加行数）が最も重要な特徴量であり、次いで num_files（変更ファイル数）、entropy（変更の広がり）が続いている。メソッド単位のメトリクスでは tokens_change（トークン数の変化量）が上位に位置し、局所的な変化の情報も予測に寄与している。この結果から、コミット単位とメソッド単位の変更メトリクスを組み合わせることで、変更の全体像と局所的な変化の両面を捉えることが可能となり、予測精度の向上につながることが確認された。

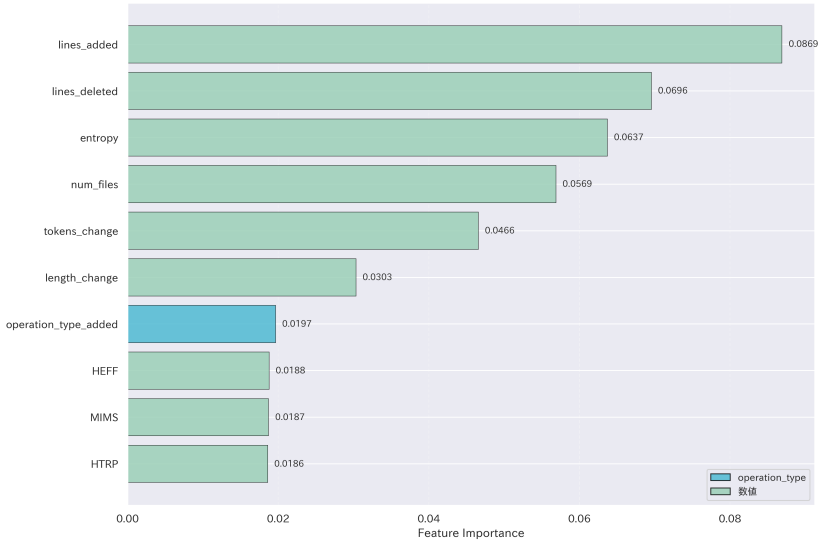


図 5.1: 特徴量重要度（代表例: Neo4j）

5.4 提案メトリクスの分布特性と予測への影響

重要度の高い特徴量の分布特性を分析した結果、コミット単位とメソッド単位のメトリクスで対照的な分布が観察された。

コミット単位のメトリクスは、右裾が長い分布を示した。大半のコミットは小規模な変更であるが、一部に大規模な変更が含まれる。この分布の偏りは、日常的な変更と大規模な機能追加やリファクタリングという異なる性質の変更が混在していることを反映している。

メソッド単位のメトリクスは異なる特性を示した。tokens_change（トークン数の変化量）や lines_change（行数の変化量）の中央値は 0 であり、欠陥混入や欠陥修正にかかわるメソッドが変更される場合は、小さな修正に留まることが確認された。

この分布特性は、提案手法の予測性能に影響を与える。コミット単位やメソッド単位の

メトリクスの偏った分布により、ランダムフォレストは、主に小規模な変更を含む訓練データから欠陥の傾向を学習する。その結果、実際の開発で頻繁に発生する小規模な変更による欠陥混入を効果的に捉えることが可能となる。

5.5 特徴量と予測確率の関係性

Partial Dependence Plot（PDP）を用いて、重要な特徴量と欠陥混入確率の関係を分析した。PDP は特定の特徴量の値を変化させたときに予測確率がどのように変化するかを示す。

コミット単位の変更メトリクスにおいて、値が増加するにつれて欠陥混入確率が低下する傾向が全プロジェクトで一貫して観察された。メソッド単位のメトリクスでは、`tokens_change`（トークン数の変化量）が 0 付近に近づくにつれて、欠陥混入確率が上昇する傾向が確認された。

これらの分析から、コミット全体の規模が小さく、欠陥にかかわるメソッドへの変更も小さいという組み合わせが、最も高い欠陥混入リスクを示すことが明らかになった。

5.6 欠陥予測基準の抽出

ランダムフォレストを構成する決定木を可視化し、モデルがどのように欠陥の有無を予測しているかを分析した。決定木の可視化により、特徴量重要度だけでは把握できない具体的な分類条件を抽出可能である。

提案手法（ステップ 3）の決定木では、ルートノード付近で `lines_added`（追加行数）や `tokens_change`（トークン数の変化量）といった変更メトリクスが頻繁に使用されていた。これらは、コミットの規模やメソッドの変化量という時系列的な情報を表す特徴量である。決定木の浅い階層でこれらの特徴量が選択されることは、コードの構造的な変化が欠陥予測において重要な特徴となっていることを示している。

5.7 同一労力での欠陥発見数増加の実証

提案手法の有効性を評価するため、レビュー労力に対する欠陥発見数を分析した。表 5.3 に、レビュー労力を 20% および 40% に制限した場合の欠陥発見率を示す。

全プロジェクトで提案手法が既存手法を上回り、同一のレビュー労力における欠陥発見率の増加が確認された。レビュー労力 20% の時点では、既存手法が平均 56.7% の欠陥を

表 5.3: レビュー労力 20% および 40% 時点での欠陥発見率

	レビュー労力 20% 時		レビュー労力 40% 時	
プロジェクト	既存手法	**提案手法**	既存手法	**提案手法**
Elasticsearch	64.7%	**68.1%**	78.8%	**86.6%**
Hazelcast	56.1%	**56.1%**	77.6%	**80.0%**
Neo4j	53.0%	**68.8%**	72.5%	**89.5%**
Netty	55.8%	**75.6%**	77.4%	**93.5%**
OrientDB	54.0%	**65.8%**	72.6%	**85.7%**

発見するのに対し、提案手法では 66.9% を発見し、10.2 ポイントの改善を達成した。レビュー労力 40% では、既存手法の 75.8% に対し提案手法は 87.0% となり、11.3 ポイントの改善が見られた。

特に Neo4j と Netty では顕著な効果が確認された。Neo4j ではレビュー労力 20% 時点で 15.8%、40% 時点で 17.0% の改善を示した。Netty では 20% 時点で 19.8%、40% 時点で 16.1% の改善となり、わずか 20% の労力で全体の 75% 以上の欠陥を検出することが可能になった。一方、Hazelcast では 20% 時点で改善が見られなかったものの、40% 時点では 2.4% の改善が確認された。

図 5.2 に、Neo4j におけるレビュー労力に対する欠陥発見数の累積曲線を示す。提案手法の曲線が既存手法より左上に位置しており、同一労力でより多くの欠陥を発見可能であることが分かる。

■統計的有意性の検証　Wilcoxon の符号順位検定により、改善の統計的有意性を検証した。表 5.4 に結果を示す。

表 5.4: 欠陥発見率に対する Wilcoxon の符号順位検定の結果

労力	既存手法	提案手法	平均改善幅	p 値
20%	56.7%	66.9%	+10.2%	0.0625
40%	75.8%	87.0%	+11.3%	0.0313

レビュー労力 40% の時点で統計的に有意な改善が確認された ($p = 0.0313$)。一方、20% の時点では有意差は認められなかった ($p = 0.0625$)。これは、20% 時点ではプロジェクト間のばらつきが大きく、Hazelcast で改善が見られなかった一方で Netty や Neo4j では大幅な改善が見られたためである。

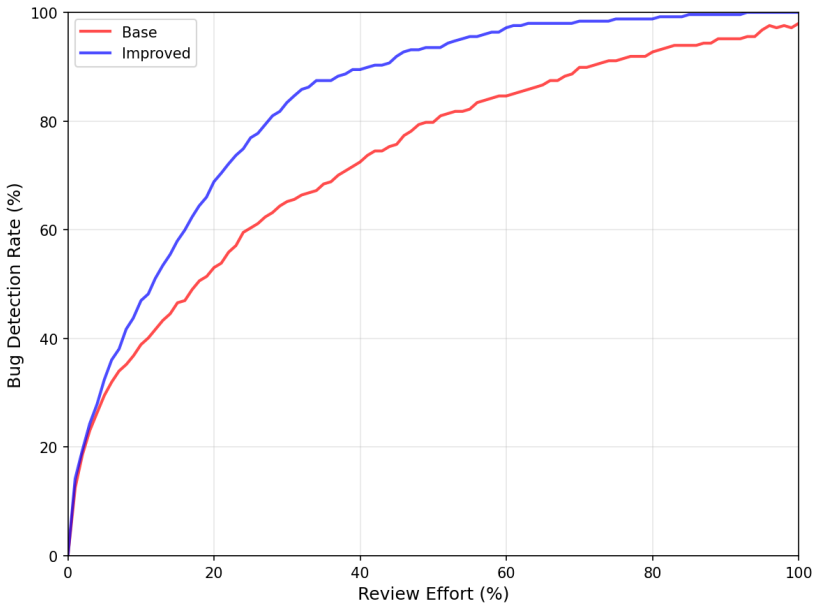


図 5.2: レビュー労力に対する欠陥発見数の比較（代表例: Neo4j）

第 6 章

考察

6.1　実験結果の性質と要因分析

第 5 章で得られた主要な結果を簡潔に要約し、本章で考察すべき疑問を明確化する。提案手法（ステップ 3）は、5 つのプロジェクトすべてで既存手法と比較して F1 スコアが向上した。レビュー労力に対する欠陥発見数の分析では、20% のレビュー労力で全欠陥の 70〜75% を検出可能であることが示された。しかし、これらの結果からいくつかの疑問が生じる。

■なぜソフトウェアの変更サイズが小さく変更ファイル数が少ないほど欠陥混入確率が高くなるのか？　Partial Dependence Plot 分析により、追加行数や変更ファイル数などの変更規模を表すメトリクスの値が増加するにつれて、欠陥混入確率が低下する傾向が全プロジェクトで一貫して観察された。これは直感に反する結果である。

■なぜプロジェクトごとに欠陥予測に影響を与える特徴量が異なるのか？　特徴量重要度を分析した結果、Netty ではトークン数の変化量が最重要特徴量であったのに対し、他のプロジェクトでは追加行数や変更ファイル数が上位を占めた。プロジェクト間で重要な特徴量が異なる理由は何か。

■なぜプロジェクト間で欠陥予測の精度が異なるのか？　F1 スコアの改善幅は Netty で 0.29、Hazelcast で 0.11 と最大 2.6 倍の差が見られた。この性能差は何に起因するのか。

■本研究の評価方法とデータセットは、これらの疑問に答えるために妥当か？　SZZ アルゴリズムや BugHunter データセットの制約が結果にどのように影響しているか。

本章では、これらの疑問に対して既存研究の知見と本研究のデータを用いて考察する。

6.2　変更規模と欠陥混入確率の関係

第 5 章の Partial Dependence Plot 分析で観察された「変更規模が小さいほど欠陥混入確率が高い」という傾向について考察する。この現象は全 5 プロジェクトで一貫して確認されており、偶然ではなく何らかの原因が存在すると考えられる。

本現象を説明する可能性として、以下の仮説が考えられる。

■欠陥密度とサンプル分布の影響　この結果は、一見すると直感に反するように思われる。一般的に、ソフトウェア工学の研究では、変更規模が大きいほど欠陥密度（単位コード量あたりの欠陥数）が高くなることが知られている。しかし、本研究で構築した機械学習モデルが予測しているのは欠陥密度ではなく、各コミットが欠陥を含む確率である。

この 2 つの概念の相違を理解する鍵は、実際のソフトウェア開発におけるコミットサイズの分布にある。ソフトウェア開発では、変更の大部分が小規模であり、大規模な変更は相対的に少数である。機械学習モデルは訓練データにおける欠陥の出現傾向から学習するため、サンプル数が多い小規模変更の領域において、より多くの欠陥例とその特徴を学習することになる。したがって、個々のコミットの欠陥密度が低くても、サンプル数が圧倒的に多ければ、その領域で観察される欠陥の絶対数は多くなり、結果として予測モデルはこの領域で高い欠陥混入確率を出力する可能性がある。

言い換えれば、大規模変更は欠陥密度が高い可能性があるものの、その絶対数が少ないため、モデルの予測において支配的な影響を持たない。一方、小規模変更は密度こそ低いが、その数の多さゆえに多くの欠陥を含んでおり、モデルはこの特徴を学習していると考えられる。

■変更の性質と目的の相違　小規模な変更と大規模な変更では、その目的や性質が異なる可能性がある。Hindle らは、大規模なコミットをカテゴリーごとに分類することでその性質を分析し、変更の規模によって種類に顕著な相違があることを示した [26]。

彼らの調査では、変更の目的を「適応的（Adaptive）」「改良的（Perfective）」「修正的（Corrective）」の 3 つに分類している。その結果、いずれの規模においても適応的な変更が頻繁に見られたが、規模の比較において興味深い逆転現象が確認された。小規模なコミットでは、改良的な変更よりも修正的な変更（バグ修正）が行われる傾向が強かったのに対し、大規模なコミットでは、修正よりも改良的な変更（機能追加やリファクタリング）が多く含まれていた。

Hindle らによれば、不具合の修正（Corrective）は「外科手術的」で局所的な処置に留

まることが多く、結果として小規模な変更になりやすい。対して、システムの改善や再構築を目的とした改良的（Perfective）な変更はその影響範囲が広く、多くのファイルに波及するため、大規模な変更となる傾向がある。

一見すると、変更規模の小さい修正的な変更の方が欠陥の混入確率は低いように思われる。しかし、システムの結合度が高い場合は、たとえ局所的な修正であってもシステム全体の整合性を損なう恐れがあり、小規模な変更が必ずしも安全であるとは限らない。

■開発者の認識の相違　変更の規模によって、開発者とレビュアーが払う注意の程度が異なる可能性がある。Bosu らは、レビューの有効性は変更のサイズとともに低下することを発見しており [27]、大規模な変更では詳細な検査が困難になることを示唆している。同様に、Kononenko らは、変更サイズがレビュー時間に大きく影響すると報告しており [28]、レビュアーがより多くの時間を費やす必要があることを示している。このレビューの困難さの相違は、欠陥発見率に影響を与える可能性がある。小規模な変更はレビューが容易であるため、潜在的な欠陥が発見されやすく、後続のコミットで修正される可能性が高い。一方、大規模な変更では詳細な検査が困難なため、欠陥が見逃されやすく、修正されないまま残存する可能性がある。本研究で使用する BugHunter データセットは SZZ アルゴリズムに基づいて欠陥混入コミットを特定しているため、実際に修正された欠陥のみが「欠陥あり」として検出される。したがって、小規模な変更で観察される高い欠陥混入確率は、変更が小規模であるほど欠陥密度が高いことを示すのではなく、レビューが容易であるため欠陥発見率が高いことを示している可能性がある。

■開発者の経験と役割分担　小規模な変更は経験の浅い開発者が担当することが多い可能性がある。Ju ら [29] は、オンボーディング戦略に関する調査において、新人開発者に対しては Simple-Complex 戦略が広く採用されていることを報告している。この戦略では、マネージャーは最初の 1 週間にバグ修正のタスクを割り当て、その後の 2〜3 週間でバグの発見に取り組ませることで、段階的にコードベースへの理解を深めさせる。一方、経験豊富な開発者に対しては、Exploration-Based 戦略が用いられ、定義が曖昧で不確実性の高いタスクが割り当てられる傾向にある。このように、開発者の経験レベルに応じてタスクの複雑さや性質が調整されることが示されている。すなわち、経験の浅い開発者が、局所的に正しく見える変更を加えた結果、他の部分への影響を見落とし、欠陥を混入させてしまうことがある。

6.3 プロジェクト間の特徴量重要度の相違

特徴量重要度を分析した結果、プロジェクトによって重要な特徴量が異なることが確認された。例えば、Netty ではトークン数の変化量が最も重要な特徴量であったのに対し、他の多くのプロジェクトでは追加行数や変更ファイル数が上位を占めた。本節では、この差異が生じる理由について考察する。

■プロジェクトドメインとアーキテクチャの影響　プロジェクトのドメインやアーキテクチャ特性が、重要な特徴量に影響を与えている可能性がある。Netty は非同期ネットワークフレームワークであり、責務が明確で焦点が絞られている。ネットワークプロトコルの実装では、わずかなトークンレベルの変更（例: バイトオーダーの処理、フラグビットの操作）が重大な欠陥を引き起こす可能性がある。そのため、トークン数の変化量のような詳細な変更メトリクスが重要になると考えられる。一方、Elasticsearch や Hazelcast のような分散システム基盤は、広範な機能セットを持ち、変更の影響範囲が広い。このようなプロジェクトでは、コミット全体の規模が、変更の複雑性や影響範囲をより適切に捉える指標となる。Neo4j や OrientDB のようなデータベースシステムでは、トランザクション処理や状態管理の複雑性が高い。これらのプロジェクトでは、変更の広がりが重要な指標となる傾向が見られた。

■開発プロセスとツールの影響　プロジェクトの開発プロセスや使用しているツールも、重要な特徴量に影響を与える可能性がある。高いテストカバレッジを持つプロジェクトでは、自動テストによって構造的メトリクスだけでもある程度の欠陥検出が可能である。このような場合、変更メトリクスによる追加的な情報の価値が相対的に低くなる。また、厳格なコードレビュープロセスを持つプロジェクトでは、小規模な変更でも慎重にレビューされるため、変更サイズと欠陥リスクの関係が弱まる可能性がある。

■欠陥の種類による影響　プロジェクトで発生する欠陥の種類によって、重要な特徴量が異なる可能性がある。ロジックバグは、条件分岐の複雑さや変更の頻度と関連が深く、コードの理解しやすさや保守性を測る特徴量が重要になる。セキュリティバグは、特定のセキュリティ問題（例: 入力検証の欠如、認証処理の不備）を持つコード箇所で発生しやすく、コードの複雑性や外部ライブラリとの連携に関する特徴量が重要になると考えられる。本研究では欠陥の種類を区別せず一律に扱っているが、欠陥の種類ごとに異なる予測モデルを構築することで、より高精度な欠陥予測が可能になると考えられる。これは今後

の重要な研究課題である。

6.4 プロジェクト間の予測性能の差異

本研究では、5 つのプロジェクトに提案手法を適用した結果、F1 スコアの改善幅に最大 2.6 倍の差（Hazelcast: 0.11 vs Netty: 0.29）が観察された。本節では、この性能差が何に起因するかを体系的に分析する。

■欠陥とコミットの特性　ステップ 1（構造的メトリクスのみ）での予測精度と、ステップ 3（変更メトリクス追加後）での改善幅の間に明確な関係が観察された。構造的メトリクスのみを使ったモデル（ステップ 1）の予測精度が比較的高いプロジェクトでは、変更メトリクス追加後のモデル（ステップ 3）でもあまり予測精度が改善しない一方で、ステップ 1 の予測精度が比較的低いプロジェクトでは、ステップ 3 で予測精度が改善しやすいことが分かる。具体的には、Hazelcast はステップ 1 で既に F1 スコア 0.68 と比較的高い性能を達成しており、改善の余地が限定的である。一方、Netty はステップ 1 で F1 スコア 0.45 と低い性能であったが、ステップ 3 で 0.75 まで向上し、0.29 という最大の改善幅を示した。

この関係は、構造的メトリクスと変更メトリクスの相補性を示している。構造的メトリクスのみで予測が困難なプロジェクトでは、変更メトリクスが提供する時系列情報が特に有効に機能する。逆に、構造的メトリクスである程度予測可能なプロジェクトでは、変更メトリクスの影響力が小さくなる。

重要なのは、改善幅の差異にかかわらず、すべてのプロジェクトでステップ 3 の最終的な F1 スコアが 0.70 以上に達していることである。これは、構造的メトリクスと変更メトリクスの組み合わせにより、プロジェクト特性の相違を超えて、より安定した予測が可能になることを示している。ステップ 1 の F1 スコアは 0.45 から 0.68 と 1.5 倍の範囲でばらついていたが、ステップ 3 では 0.70 から 0.79 と 1.1 倍程度に収束している。このばらつきの減少は、提案手法が多様なプロジェクトに対して一貫した性能を提供することが可能であることを意味する。

さらに、既存手法の性能と改善幅の関係を詳しく分析するため、各プロジェクトのデータセットにおける陽性クラス（欠陥が含まれるデータ）の割合を算出し、F1 スコアの改善幅との相関を調査した。表 6.1 に、各プロジェクトの陽性クラスの割合と F1 改善幅を示す。

この結果から、陽性クラスの割合と F1 改善幅の間に強い負の相関（Pearson 相関係数

表 6.1: データセットの陽性クラス割合と F1 改善幅の関係

プロジェクト	陽性クラスの割合	F1 改善幅
Netty	0.22	0.29
Neo4j	0.26	0.26
OrientDB	0.26	0.22
Elasticsearch	0.35	0.19
Hazelcast	0.37	0.11

$r = -0.93$, $p = 0.020$）が観察された。すなわち、データセット中の欠陥を含むメソッドの割合が低いほど（データの不均衡が大きいほど）、変更メトリクスの追加による性能改善効果が大きい傾向がある。

この関係は以下のように解釈することが可能である。データが比較的バランスの取れているプロジェクト（Hazelcast: 陽性クラス 37.5%、Elasticsearch: 35.0%）では、構造的メトリクスのみでも陽性クラスと陰性クラスの判別が比較的容易であり、既存手法の性能が高い。一方、データの不均衡が大きいプロジェクト（Netty: 陽性クラス 21.8%、Neo4j: 25.8%）では、陽性クラスが少数派であるため構造的メトリクスのみでは識別が困難であり、既存手法の性能が低くなる。このような不均衡データに対して、変更メトリクスが提供する時系列的な変化の特徴は、少数派クラスを識別するための追加的な判別情報として特に有効に機能すると考えられる。

データの不均衡は機械学習における一般的な課題であるが、本研究の結果は、異なる構成要素に基づく変更メトリクスの組み合わせが不均衡データに対する有効な対処法の一つとなりうることを示唆している。

6.5 カテゴリーごとの制約

■データセット BugHunter データセットでは、欠陥混入コミットの特定に SZZ アルゴリズムを採用している。SZZ は、バグ修正コミットから変更履歴を遡って欠陥を混入したコミットを特定する手法である。

SZZ アルゴリズムの長所は、大規模なデータセットを自動的に構築可能である点と、明確なアルゴリズムに基づくため再現性が高い点である。数千から数万のコミットを含むプロジェクトにおいて、人手によるラベル付けは現実的でないが、SZZ は VCS と課題管理システムの情報を活用することで効率的にラベル付きデータを生成可能である。

一方で、SZZ には精度に関する課題も存在する。Herbold らの調査によれば、SZZ は欠陥修正コミットの約 5 分の 1 を見逃し、SZZ が欠陥修正として識別したコミットのうち実際に欠陥修正であるのは約半分である [30]。これは、SZZ が課題管理システムに記録されたバグのみを追跡するため、軽微な修正や開発中に発見された問題を検出できないことに起因する。

本研究では、これらのトレードオフを考慮した上で、以下の理由から BugHunter データセットを採用した。SZZ の短所は既存手法と提案手法に等しく影響するため、相対的な性能比較には大きな影響を与えない。ただし、本研究で得られたモデルの絶対的な予測精度は、実際の JIT 品質保証における真の予測精度と乖離している可能性があることに留意する必要がある。

さらに、本研究では各プロジェクトのデータセットから抽出するレコード数の上限を 5,000 行としているが、この制限が欠陥予測精度を低下させ、レビュー対象のコミットの選択を誤らせる恐れがある。より多くのデータを使用すれば性能がさらに向上する可能性がある。

■学習手法　本研究では機械学習アルゴリズムとしてランダムフォレストを採用したが、近年の関連研究では、ディープニューラルネットワーク（DNN）がより高い F1 スコアを達成することが報告されている。DNN が優れた性能を示す理由として、複数の隠れ層を通じて非線形な特徴表現を自動的に学習可能である点と、特徴間の高次の相互作用を効果的にモデル化可能である点が挙げられる。例えば、変更の規模、開発者の経験、コードの結合度が複雑に組み合わさって欠陥混入リスクが上昇するといった非線形な関係を、DNN はより柔軟に捉えられる可能性がある。

一方、本研究がランダムフォレストを選択した理由は、解釈可能性を重視したためである。本研究の目的は予測精度の最大化だけでなく、どの特徴が欠陥予測に寄与するかを理解することにある。ランダムフォレストは特徴量重要度や Partial Dependence Plot といった分析手法を提供し、これらは 6.2 節の変更規模と予測確率の関係や、6.3 節のプロジェクト間の特徴量重要度の相違といった考察に不可欠である。DNN は予測精度が高い一方で、内部表現がブラックボックス化しやすく、このような詳細な分析が困難である。

また、本研究では、ランダムフォレストのハイパーパラメーター（決定木の数、深さ、最小サンプル数など）の最適化を実施していない。ハイパーパラメーターチューニングにより、性能がさらに向上する可能性がある。

次に、特徴量エンジニアリングにおいてメソッドの変更タイプを導入したが、変更の意図が見逃されている。例えば、機能追加やリファクタリングという目的を特徴量として加

えることで、性能が向上する可能性がある。

最後に、レビュー労力に対する欠陥発見数の分析におけるレビュー労力の計算方法は実際のレビュー労力と完全には一致しない。なぜなら、レビュー労力は変更の規模や複雑度だけでなく、レビュアーの経験やコードベースの知識にも依存するためである。

■ケーススタディ　ケーススタディで使用したソフトウェア開発プロジェクトは、すべてJava で実装されている。本研究の手法で得られた知見の汎用性を検討するため、言語特性を「型システムと安全性」「メモリ管理方式」「例外処理とエラー伝播モデル」の 3 つのカテゴリーに分類し、Java と他言語の比較を通じて、本データセットにおける欠陥の性質を考察する。

1 つ目は、型システムと安全性である。Java は静的型付けを採用しており、コンパイル時に多くの型不整合が検出される。これに対し、Python や JavaScript のような動的型付け言語では、実行時に初めて型エラーが顕在化することが多い。この違いにより、本研究のデータセットに含まれる欠陥は、型不整合のような基本的な記述ミス以外の論理構造の誤りに集中していると推測される。また、C 言語で見られるようなポインター演算によるメモリへの直接アクセスが制限されているため、バッファオーバーフローに起因する脆弱性や欠陥も、Java プロジェクトでは比較的少ない。

2 つ目は、メモリ管理方式である。Java はガベージコレクションによる自動メモリ管理を備えている。これは、C++ のように開発者が手動でメモリの確保と解放を管理する言語と比較して、メモリリークや不正なメモリ参照による欠陥の発生を抑える。本研究で抽出されたメトリクスは、こうしたメモリ管理の不備よりも、メソッド間の呼び出し関係や制御フローの複雑化に起因する論理的な欠陥をより強く反映している。

3 つ目は、例外処理とエラー伝播モデルである。Java は検査例外（Checked Exception）という、呼び出し側に例外処理を強制する仕組みを言語仕様として持つ。これは、Python のような非検査例外が主流の言語や、Go のように戻り値でエラーを扱う言語とは対照的である。

この仕組みは、コードの制御フローグラフに大きな影響を与える。たとえば、外部リソースへのアクセスを 1 行追加するだけで、それに付随して複数の例外処理（try-catch ブロック）の記述が必要になる。これが循環的複雑度やトークン数の変化量を増加させ、例え小規模な変更であっても、分岐を複雑化させる要因となる。本研究で示された小規模な変更における高い欠陥混入確率の背景には、こうした Java 特有の厳格なエラーハンドリングが、変更の「見かけの小ささ」に対して「論理的な複雑さ」を増大させていると考えられる。

以上の比較から、本研究で得られた知見は、Java 特有の「静的型付けに保護された環境下でののコード変更」において顕著に現れる特性である可能性がある。したがって、異なる言語特性を持つプロジェクトに本手法を適用する際には、対象となる言語の安全性やエラーハンドリングの慣習に応じたメトリクスの再選定が必要となる。

6.6 欠陥混入と欠陥修正における多対多関係への対応

本研究では、欠陥予測を「その変更に欠陥が含まれるかどうか」という二値分類問題として扱った。しかし、ソフトウェア開発においては、欠陥の混入と修正は必ずしも 1 対 1 ではなく、多対多の対応関係を持つ。本節では、この多対多関係が予測モデルに与える影響と、その解決策について考察する。

■多対多関係の実態と課題　実際のプロジェクトでは、1 つの大きな変更が複数の独立した欠陥を誘発するケース（1:N）や、複数のコミットにまたがる変更が積み重なって初めて 1 つの欠陥が顕在化するケース（N:1）が発生する。現在の二値分類による欠陥予測では、これらの依存関係を「欠陥の有無」という単一のラベルに集約することになり、各コミットが持つ欠陥混入への寄与度を詳細に評価できないという課題がある。

■回帰モデルによる寄与度の予測　この多対多関係を適切に扱うための解決策として、二値分類から回帰モデルへの移行が有効である。回帰モデルを導入することで、特定のコミットが「いくつの欠陥に関与したか」あるいは「欠陥全体に対してどの程度の責任（寄与度）を持つか」を連続値として予測可能になる。これにより、3.4 節で定義した密度 D_i において、「予測された欠陥の影響度」を $\hat{y}_i$ として活用でき、レビューの優先順位付けをより正確に行える。

■SZZ アルゴリズムのノイズと相対的リスクの識別　回帰モデルの導入にあたっては、正解ラベルの生成に用いる SZZ アルゴリズムの精度が課題となる。SZZ は欠陥修正の見落としや誤検出を含むため、欠陥混入への寄与度を厳密な連続値として定義しようとすると、学習データに大きなノイズが生じる懸念がある。しかし、開発現場の意思決定支援においては、欠陥混入数の予測よりも、リソース配分のための「相対的な欠陥混入リスクの差異」を正しく捉えることがより重要である。したがって、欠陥混入数そのものではなく、プロジェクト内での相対的な影響度を予測することで、SZZ の不完全性を許容しつつ、多対多の環境下でも高リスクな変更を効果的に識別できると考えられる。

■説明可能性を考慮したアルゴリズム選定　多対多の関係性をモデル化する場合であっても、本研究が重視している「予測の根拠」の提示は不可欠である。しかし、分類と回帰の両方に対応している機械学習アルゴリズムは多い。一例として、ランダムフォレストを用いた回帰モデルでは、特徴量重要度や決定木の可視化を通じた分析が可能である。これにより、複雑な依存関係においてどのメトリクスが欠陥混入の主な原因となっているかを、開発者が直感的に理解できる形でフィードバックすることが可能となる。

6.7　欠陥混入の原因の特定

本節では、1.2 節で設定した 3 つの研究目的に対する達成状況を振り返る。さらに、それを踏まえて今後の研究課題を考察する。

1 つ目の目的「異なる構成要素からなる変更メトリクスの活用」については、メソッド単位の局所的な変化とコミット単位の全体的な変化を組み合わせた特徴量を設計し、全プロジェクトで予測精度の向上を確認できた。5.2 節で示したように、両者を組み合わせることで単独使用時よりも高い F1 スコアを達成しており、複数の構成要素の変化を同時に捉えることの有効性を示すことができた。

2 つ目の目的「不規則な時系列変化を考慮した欠陥予測モデルの構築」については、コミットを点過程データとして扱い、直前の状態からの差分を用いて特徴量を抽出する手法を提案した。5.3 節の結果から、この差分ベースのアプローチが構造的メトリクスのみの手法と比較して一貫した性能向上をもたらすことが示された。

3 つ目の目的「レビュー労力の測定と欠陥発見率の向上」については、変更の規模と複雑度に基づくレビュー労力指標を定義し、5.4 節で示したように、同一のレビュー労力で既存手法よりも多くの欠陥を発見可能であることを確認した。

一方で、変更の目的を変更の特徴と関連付け、欠陥の原因を特定することはできていない。6.2 節で議論したように、小規模な変更と大規模な変更ではその目的が異なる。変更の目的の相違は、欠陥混入リスクに大きく影響を与える可能性があり、欠陥の原因とも関連性が高いが、本研究ではこの変更の意図を抽出し、活用することができなかった。

変更の目的を特定することの利点はいくつかある。例えば、6.2 節で観察された変更の規模と欠陥混入確率の逆転現象について、変更の目的に応じた説明が可能になる。具体的には、「欠陥修正を目的とした小規模な変更」と「機能追加を目的とした小規模な変更」を区別して欠陥混入確率を算出することで、変更規模と欠陥混入確率の関係が変更の目的によってどのように異なるかを明らかにすることが可能である。Hindle らの研究 [26] が示

唆するように、小規模な変更は修正を意図したものであることが多いという傾向が確認されれば、小規模な変更に見られる高い欠陥混入確率は、変更の目的そのもの（過去の欠陥を修正する過程で新たな欠陥を混入させる）に起因すると結論付けることが可能である。逆に、機能追加を目的とした小規模な変更の欠陥混入確率が低ければ、変更規模が小さいこと自体が安全性を保証するわけではなく、変更の目的が重要であることが明確になる。

このように、変更の目的を明らかにするための手段を考えることは、欠陥が混入した原因を突き止めるのに役立つ。しかし、変更の目的を短時間で特定するのは容易ではない。例えば、ソフトウェア欠陥の分類手法として広く用いられている Orthogonal Defect Classification（ODC）を適用した研究 [31] では、NoSQL データベースの 4096 件の欠陥を手動で分類するために、まず 300 件の欠陥を用いて研究者をトレーニングし、その後、各欠陥に対して複数の属性（Activity、Trigger、Impact、Target、Defect Type、Qualifier）を手作業で追加している。

しかし、このような詳細な欠陥分類を実施するには、専門的な知識と訓練が必要であり、大量のコミットに対して手動で適用する場合はさらに多くの時間が必要になる。欠陥の詳細な分類には多大な労力が必要であり、本研究で扱う数千から数万のコミットに対して同様のアプローチを適用することは困難である。さらに、ODC のような既存の欠陥分類手法は、欠陥が発見された後に分析が行われることを前提としている。一方、本研究で必要なのは、コミット時点での変更の目的の特定であり、欠陥がまだ顕在化していない可能性のある変更の意図を推定する必要がある。

これらの課題を解決するために、欠陥に対するラベル付けの負担を最小限に抑えつつ、変更が加えられたときにその目的を推定する手法が求められる。これまでは分析対象となるソフトウェアについての深い知識を持つ開発者が欠陥を手動で分類していた。これは、彼らが経験の蓄積によって、特定の変更の背景を推測することに長けていたからであると考えられる。このことから、これまでに行われた変更の目的を理解した上で、新しい変更の目的を推測することが、欠陥分類の精度を担保する上で重要であるといえる。

この知見から、変更目的の推定を効率的かつ迅速に行うためには、熟練した開発者が暗黙的に参照している情報を明らかにし、いくつかの要素に整理する必要がある。変更目的の推定に必要な要素には、以下のようなものがある。

- コミットに関する情報
  - これには、コミットメッセージのテキスト、コミットに紐付けられた課題管理システムの課題番号とそのラベル（“bug”, “enhancement”, “refactoring” など）、レビューでのコメントや議論の内容が含まれる。これらの情報は、開発

者が変更の目的を明示的に記述したものであり、最も直接的な情報源である。ただし、プロジェクトや開発者によって記述の詳細度が大きく異なるため、これらの情報が実際にどの程度役立つのかを事前に評価する必要がある。

- コード変更の構造に関する情報
  - これには、変更された構成要素の機能（テストコード、ドキュメント、設定ファイルなど）、変更を構成する処理（エラーハンドリングなど）が含まれる。例えば、テストファイルのみが変更されている場合は「テスト追加」である可能性が高く、エラーハンドリングのコードが追加されている場合は「欠陥修正」である可能性が高い。これらの情報は、静的コード解析ツールや抽象構文木（AST）の比較により自動的に抽出可能であるため、コミットメッセージが不十分な場合でも有効な判断材料となる。
- 変更の文脈に関する情報
  - これには、変更された構成要素の過去の変更履歴（過去にどのような目的の変更が多かったか）、その構成要素における過去の欠陥報告の有無と頻度、変更箇所の周辺コードの特性（複雑度、結合度など）、プロジェクト全体における最近の変更傾向（リリース前の欠陥修正フェーズか、新機能開発フェーズかなど）が含まれる。熟練した開発者は、これらの文脈情報を参照することで、同じコード変更でも異なる解釈を行う。例えば、過去に頻繁に欠陥が報告されている構成要素への変更は「欠陥修正」である可能性が高いが、新規に作成されたファイルへの変更は「機能追加」である可能性が高い。
- 開発者に関する情報
  - これには、変更を行った開発者のプロジェクトへの貢献履歴（貢献期間、過去のコミット数）、その開発者が過去に行った変更の目的の傾向（主に欠陥修正を担当しているか、新機能開発を担当しているか）、その開発者の専門領域（特定のモジュールやコンポーネントに精通しているか）が含まれる。6.2 節で議論したように、経験の浅い開発者は小規模な欠陥修正を担当することが多いという仮説があり、開発者の特性は変更目的の推定において重要な手がかりとなる。

これらの情報を用いることで、効率的かつ迅速に変更目的を推定可能になると考えられる。変更目的を推定する際は、利用可能な情報の質と量がプロジェクトによって大きく異なることを認識し、各プロジェクトにおいて最も信頼性の高い情報源を優先的に活用することが重要である。信頼性の高い情報はプロジェクトごとに異なるため、利用可能な情報

を適切に重み付けする工夫が求められる。

6.8 開発現場における適用例

本研究で提案した手法が、実際のソフトウェア開発現場でどのように貢献し、開発者の意思決定を支援するかについて、具体的なユースケースを用いて説明する。

■迅速なリリースが求められる大規模 Web サービスの開発　ある大規模 Web サービスを開発しているチームを想定する。このチームでは CI/CD パイプラインを導入しており、1 日に数十件のコミットが発生している。チームが抱える人員に対してシステムの規模が大きいため、すべての変更を詳細にレビューすることは現実的ではなく、多くの場合、変更行数の少ない小規模な変更のレビューが簡略化されがちである。ここで本手法を適用すると、以下のようなプロセスで品質向上が図られる。

初めに、開発者が「特定のメソッド内の数トークンのみを変更した小規模なコミット」をプッシュした際、本モデルがそれを「高リスク」と判定し、レビューツール上に警告を表示する。従来の変更規模に基づく評価では見逃されていた、局所的な変更が全体に及ぼすリスクを数値として提示する。

続いて、プロジェクトリーダーは、本手法が出力する「密度 D_i（単位労力あたりの欠陥発見期待値）」に基づき、その日のレビュー対象を優先順位付けする。実験結果（5.7 節）で示した通り、総労力のわずか 20% を「密度の高いコミット」に集中させるだけで、潜在的な欠陥の約 7 割を早期に特定できる。

また、特徴量重要度の分析により、なぜその変更が「高リスク」と判定されたのか（例: 特定の複雑なメソッドのトークン変化が激しいため）を開発者にフィードバックできる。これにより、開発者は自身の修正が既存コードの整合性にどう影響したかを再認識し、修正の質を自発的に向上させることが可能となる。

本手法の最大の長所は、人間の直感に依存しないレビュー労力の分配を可能にする点にある。従来、「小規模な変更は安全である」と過信されていた領域に潜むリスクを、時系列変化と複数の構成要素からなるメトリクスを組み合わせることで明らかにした。これにより、開発チームは、どこを重点的に見るべきか分からないという悩みから解放され、新機能の開発やリファクタリングに注力できるようになる。

第 7 章

おわりに

本研究では、変更履歴を活用し、欠陥予測とレビューの優先度付けの効率化を目指した。現代のソフトウェア開発では頻繁なコード変更が不可欠である一方、変更に伴う欠陥混入リスクが品質保証上の課題となっている。従来の構造的メトリクスのみでは変更の特徴を十分に捉えられず、レビューに費やされる労力を効率的に配分する手法も改善が必要であった。本研究では、メソッド単位とコミット単位という異なる構成要素からなる変更メトリクスの活用、不規則な時系列変化を考慮した欠陥予測モデルの構築、レビュー労力の測定と欠陥発見率の向上という 3 つの研究目的を設定した。

これらの目的を達成するため、コミットを点過程データとして扱い、直前の状態からの差分を特徴量に変換する手法を提案した。5 つのオープンソースプロジェクトを用いた実験により、提案手法が従来の構造的メトリクスのみを用いる手法と比較して予測精度を向上させることを示した。また、レビュー労力を考慮した優先度付け手法により、同一の労力でより多くの欠陥を発見可能であることを確認した。最終的に、欠陥予測モデルのクラス分類の詳細とレビューの優先度付け手法を関連付けることで、開発者の意思決定を支援するモデルを構築した。

本研究では、設定した 3 つの研究目的を達成することができた。メソッド単位とコミット単位の変更メトリクスを組み合わせることで、全プロジェクトで予測精度の向上を確認した。不規則な時系列変化を考慮した差分ベースのアプローチにより、全プロジェクトで F1 スコアが向上した。レビュー労力を考慮した優先度付け手法により、レビュー労力を効果的に配分し、欠陥発見率を向上させられることを示した。第 6 章では、変更規模と欠陥混入確率の関係性、プロジェクト間の特徴量重要度の相違、予測性能の差異とその要因について考察した。

ただし、変更の目的を変更の特徴と関連付け、欠陥の原因を特定することは十分に達成

できていない。変更の目的と欠陥の種類の関係を明らかにすることで、開発者に対して目的別の注意点を提示するガイドラインを作成し、欠陥の予防に繋げられる可能性がある。

今後の研究課題としては、欠陥の混入と修正における多対多関係の解明と、変更目的の推定による欠陥混入原因の特定がある。

まず、欠陥の混入と修正における多対多関係の解明については、従来の二値分類モデルを拡張し、各変更が持つ「欠陥混入への寄与度」を予測可能な回帰モデルに移行することが求められる。本研究では欠陥の有無を予測対象としたが、実際には 1 つの変更が複数の欠陥を誘発したり（1:N）、複数の変更が累積して 1 つの欠陥を生んだりする（N:1）といった複雑な依存関係が存在する。これを分析するためには、欠陥混入確率ではなく「予測される欠陥の影響度」を連続値として示し、それに基づいた密度による優先順位付けを行う必要がある。

次に、変更目的の推定による欠陥混入原因の特定については、熟練した開発者が暗黙的に参照している情報を体系化し、自動的に抽出する手法の構築が必要である。具体的には、コミットメッセージや課題管理システムのラベルといった直接的な情報に加え、抽象構文木（AST）の比較による変更構造の解析、過去の修正履歴や開発者の経験といった文脈情報を活用することが重要である。これにより、開発者に対する具体的なフィードバックの提供が可能になる。

参考文献

[1] 日本産業標準調査会. *JIS X 0161: 2008* **ソフトウェア技術－ソフトウェアライフサイクルプロセス－保守**. 日本規格協会, 2008.

[2] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014.

[3] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[4] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[5] Ahmed E. Hassan. Predicting faults using the complexity of code changes. *Proc. of IEEE 31st International Conference on Software Engineering (ICSE)*, pages 78–88, 2009.

[6] Norman E. Fenton and Martin Neil. Software metrics: Successes, failures and new directions. *Journal of Systems and Software*, 47(2):149–157, 1999.

[7] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[8] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[9] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. *Proc. of 27th International Conference on Software Engineering (ICSE)*, pages 284–292, 2005.

[10] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris

Mockus, and Anand Sinha. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[11] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[12] Baljinder Ghotra, Shane McIntosh, and Ahmed E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. *Proc. of IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, 1:789–800, 2015.

[13] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

[14] Thilo Mende and Rainer Koschke. Effort-aware defect prediction models. *Proc. of 14th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 107–116, 2010.

[15] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for Java programs. *Proc. of International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, 2014.

[16] Rudolf Ferenc, Péter Gyimesi, Gábor Gyimesi, Zoltán Tóth, and Tibor Gyimóthy. An automatically created novel bug dataset and its validation in bug prediction. *Journal of Systems and Software*, 169:110691, 2020.

[17] Mehrasa Modanlou Jouybari, Alireza Tajary, Mansoor Fateh, and Vahid Abolghasemi. A novel deep neural network structure for software fault prediction. *PeerJ Computer Science*, 10:e2270, 2024.

[18] Judit Jász. The effectiveness of hidden dependence metrics in bug prediction. *IEEE Access*, 12:77214–77225, 2024.

[19] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *Proc. of International Workshop on Mining Software Repositories (MSR)*, pages 1–5, 2005.

[20] Elasticsearch. elastic/elasticsearch: Free and open source, distributed, RESTful search engine. `https://github.com/elastic/elasticsearch`. (2025-12-28 参照).

[21] Hazelcast. hazelcast/hazelcast: Hazelcast is a unified real-time data platform

combining stream processing with a fast data store, allowing customers to act instantly on data-in-motion for real-time insights. https://github.com/hazelcast/hazelcast. (2025-12-28 参照).

[22] The Netty Project. netty/netty: Netty project - an event-driven asynchronous network application framework. https://github.com/netty/netty. (2025-12-28 参照).

[23] OrientDB. orientechnologies/orientdb: Orientdb is the most versatile dbms supporting graph, document, reactive, full-text and geospatial models in one multi-model product. orientdb can run distributed (multi-master), supports SQL, ACID transactions, full-text indexing and reactive queries. https://github.com/orientechnologies/orientdb. (2025-12-28 参照).

[24] Neo4j. neo4j/neo4j: Graphs for everyone. https://github.com/neo4j/neo4j. (2025-12-28 参照).

[25] Hiroto Sasagawa. nagutabby/bug-hunter. https://github.com/nagutabby/bug-hunter. (2026-01-04 参照).

[26] Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us? A taxonomical study of large commits. *Proc. of International Working Conference on Mining Software Repositories (MSR)*, pages 99–108, 2008.

[27] Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at Microsoft. *Proc. of IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR)*, pages 146–156, 2015.

[28] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. Code review quality: How developers see it. *Proc. of 38th International Conference on Software Engineering (ICSE)*, pages 1028–1038, 2016.

[29] An Ju, Hitesh Sajnani, Scot Kelly, and Kim Herzig. A case study of onboarding in software teams: Tasks and strategies. *Proc. of 43rd International Conference on Software Engineering (ICSE)*, pages 613–623, 2021.

[30] S. Herbold, A. Trautsch, F. Trautsch, et al. Problems with SZZ and features: An empirical study of the state of practice of defect prediction data collection. *Empirical Software Engineering*, 27(42), 2022.

[31] João Agnelo, Nuno Laranjeiro, and Jorge Bernardino. Using orthogonal defect classification to characterize NoSQL database defects. *Journal of Systems and*

Software, 159:110451, 2020.

付録 A

ソースコード

本章では、第 3 章および第 4 章で説明した処理のソースコードを示す。

A.1　メソッド単位の変更メトリクスの抽出

メソッド単位の変更メトリクスを抽出する処理を以下に示す。この処理では、連続するコミット間でのメソッドの変更を検出し、変更タイプ（追加、削除、修正）および循環的複雑度・コード行数・トークン数の変化量を計算する。

```
def calculate_complexity_changes(complexity_data):
    if len(complexity_data) == 0:
        return {
            'current_commit': None,
            'ccn_change': None,
            'length_change': None,
            'tokens_change': None,
            'operation_type': 'NaN'
        }
    elif len(complexity_data) == 1:
        single_data = complexity_data[0]

        if single_data['commit_order'] == 1:
            return {
                'current_commit': single_data['commit_hash'],
                'ccn_change': None,
                'length_change': None,
                'tokens_change': None,
```

```
                'operation_type': 'deleted'
            }
        else:
            return {
                'current_commit': single_data['commit_hash'],
                'ccn_change': None,
                'length_change': None,
                'tokens_change': None,
                'operation_type': 'added'
            }
    elif len(complexity_data) == 2:
        previous_data = complexity_data[0]
        current_data = complexity_data[1]

        ccn_change = current_data['ccn'] - previous_data['ccn']
        length_change = current_data['length'] - previous_data['
            length']
        tokens_change = current_data['tokens'] - previous_data['
            tokens']

        return {
            'current_commit': current_data['commit_hash'],
            'ccn_change': ccn_change,
            'length_change': length_change,
            'tokens_change': tokens_change,
            'operation_type': 'modified'
        }
    else:
        return {
            'current_commit': None,
            'ccn_change': None,
            'length_change': None,
            'tokens_change': None,
            'operation_type': 'NaN'
        }
```

A.2　コミット単位の変更メトリクスの抽出

コミット単位の変更メトリクスを抽出する処理を以下に示す。この処理では、各コミットにおける全体的な変更の特徴（変更ファイル数、コードの追加・削除行数、変更の広がり）を計算する。

A.2.1　変更ファイル数とコードの追加・削除行数の計算

```
def get_commit_change_stats(repo_path, commit_hash):
    try:
        repo = Repo(repo_path)
        commit = repo.commit(commit_hash)

        if not commit.parents:
            return None

        parent = commit.parents[0]

        files_changed = 0
        lines_added = 0
        lines_deleted = 0
        total_lines_before = 0
        lines_per_file = []

        diff_index = parent.diff(commit)

        for diff_item in diff_index:
            if diff_item.change_type in ['A', 'M', 'D', 'R']:
                files_changed += 1

            file_lines_changed = 0

            if diff_item.change_type == 'M':
                try:
                    a_content = diff_item.a_blob.data_stream.read().
                        decode('utf-8', errors='ignore')
                    b_content = diff_item.b_blob.data_stream.read().
                        decode('utf-8', errors='ignore')
```

```
29
30                  a_lines = a_content.splitlines()
31                  b_lines = b_content.splitlines()
32
33                  total_lines_before += len(a_lines)
34
35                  import difflib
36                  diff = difflib.unified_diff(a_lines, b_lines,
                        lineterm='')
37
38                  for line in diff:
39                      if line.startswith('+') and not line.
                            startswith('+++'):
40                          lines_added += 1
41                          file_lines_changed += 1
42                      elif line.startswith('-') and not line.
                            startswith('---'):
43                          lines_deleted += 1
44                          file_lines_changed += 1
45              except Exception as e:
46                  pass
47
48          elif diff_item.change_type == 'A':
49              try:
50                  b_content = diff_item.b_blob.data_stream.read().
                        decode('utf-8', errors='ignore')
51                  b_lines = b_content.splitlines()
52                  lines_added += len(b_lines)
53                  file_lines_changed += len(b_lines)
54              except:
55                  pass
56
57          elif diff_item.change_type == 'D':
58              try:
59                  a_content = diff_item.a_blob.data_stream.read().
                        decode('utf-8', errors='ignore')
60                  a_lines = a_content.splitlines()
61                  total_lines_before += len(a_lines)
62                  lines_deleted += len(a_lines)
63                  file_lines_changed += len(a_lines)
```

```
                except:
                    pass

            if file_lines_changed > 0:
                lines_per_file.append(file_lines_changed)

        return {
            'files_changed': files_changed,
            'lines_added': lines_added,
            'lines_deleted': lines_deleted,
            'total_lines_before': total_lines_before,
            'lines_per_file': lines_per_file
        }

    except Exception as e:
        return None
```

A.2.2 変更の広がりの計算

```
def calculate_entropy(lines_per_file):
    if not lines_per_file or len(lines_per_file) == 0:
        return None

    if len(lines_per_file) == 1:
        return 0.0

    total_lines = sum(lines_per_file)
    if total_lines == 0:
        return None

    entropy = 0.0
    for lines in lines_per_file:
        if lines > 0:
            pk = lines / total_lines
            entropy -= pk * np.log2(pk)

    normalized_entropy = entropy / np.log2(len(lines_per_file))
    return normalized_entropy
```

A.2.3 コミット単位の変更メトリクスの計算

```
def calculate_vcs_metrics(stats):
    if stats is None:
        return {
            'num_files': None,
            'lines_added': None,
            'lines_deleted': None,
            'entropy': None
        }

    nf = stats['files_changed']
    la = stats['lines_added']
    ld = stats['lines_deleted']

    entropy = calculate_entropy(stats['lines_per_file'])

    return {
        'num_files': nf,
        'lines_added': la,
        'lines_deleted': ld,
        'entropy': entropy
    }
```

A.3　レビュー労力の計算

レビュー労力を計算する処理を以下に示す。この処理では、コードの変更行数、変更ファイル数、および変更の広がりに基づいてレビューに必要な労力を推定する。

A.3.1　レビュー労力の推定

```
def calculate_effort(code_churn, num_files, entropy):
    if num_files == 0 or code_churn == 0:
        raw_effort = 1.0
    elif num_files == 1:
        raw_effort = float(code_churn)
    elif entropy is None or entropy == 0 or np.isnan(entropy):
        raw_effort = float(code_churn)
    else:
        raw_effort = code_churn * (num_files ** entropy)

    effort = np.log(raw_effort + 1)

    return effort
```

A.3.2　貪欲法によるレビュー対象コミットの選択

少ないレビュー労力で高い欠陥発見率を達成するために、貪欲法を用いてレビュー対象コミットを選択する。

```
def knapsack_greedy(weights, values, capacity):
    n = len(weights)

    items = []
    for i in range(n):
        if weights[i] > 0:
            ratio = values[i] / weights[i]
            items.append((i, ratio, weights[i], values[i]))
        else:
            items.append((i, 0, weights[i], values[i]))

```

```
    items.sort(key=lambda x: x[1], reverse=True)

    selected_indices = []
    total_weight = 0.0
    total_value = 0.0

    for idx, ratio, weight, value in items:
        if total_weight + weight <= capacity:
            selected_indices.append(idx)
            total_weight += weight
            total_value += value

    selected_indices.sort()

    return selected_indices, total_value, total_weight
```

A.3.3 欠陥発見率の計算

レビュー労力に対する欠陥発見率を計算する関数を以下に示す。

```
def calculate_cost_benefit_greedy(y_true, y_pred_proba, efforts,
    capacity_ratios):
    total_effort = np.sum(efforts)
    total_bugs = np.sum(y_true)

    results = []

    results.append({
        'capacity_ratio': 0.0,
        'effort_used': 0.0,
        'commits_reviewed': 0,
        'bugs_found': 0.0,
        'review_effort_ratio': 0.0,
        'bug_detection_ratio': 0.0
    })

    for capacity_ratio in capacity_ratios:
        capacity = total_effort * capacity_ratio

        selected_indices, total_value, total_weight =
```

```
            knapsack_greedy(
20              weights=efforts,
21              values=y_pred_proba,
22              capacity=capacity
23          )
24
25          bugs_found = np.sum(y_true[selected_indices])
26
27          results.append({
28              'capacity_ratio': capacity_ratio,
29              'effort_used': total_weight,
30              'commits_reviewed': len(selected_indices),
31              'bugs_found': bugs_found,
32              'review_effort_ratio': total_weight / total_effort if
                    total_effort > 0 else 0,
33              'bug_detection_ratio': bugs_found / total_bugs if
                    total_bugs > 0 else 0
34          })
35
36      return pd.DataFrame(results)
```