修士学位論文

題目

構文情報を用いた機械学習に基づくリファクタリング箇所の推薦 –メソッド抽出リファクタリングへの適用–

指導教員

楠本 真二 教授

報告者

今里 文香

平成 27 年 2 月 6 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

平成 26 年度 修士学位論文

構文情報を用いた機械学習に基づくリファクタリング箇所の推薦
–メソッド抽出リファクタリングへの適用–

今里 文香

内容梗概

リファクタリングとは，ソフトウェアの外部的振る舞いを保ちながら，その内部構造を変更する技術である．リファクタリングの主な目的は，ソースコードを整理することにより，ソフトウェアの保守性を向上させることである．リファクタリングは，ソフトウェアの開発過程における様々な場面で用いられる重要な技術であり，これまでに，リファクタリングに関する多くの研究が行われている．その中には，リファクタリングすべき箇所を開発者に推薦する手法を提案している研究がいくつか存在する．リファクタリング箇所の推薦は，開発者が膨大なソースコードの中からリファクタリングすべき箇所を特定する手間や時間を削減することに役立つ．しかし，リファクタリング箇所には明確な基準が存在しない．したがって，ある開発者やプロジェクトにとってはリファクタリング対象となる箇所が，他の開発者やプロジェクトにとってはリファクタリング対象とならない可能性がある．

そこで本研究では，機械学習を用いて，ソースコード中のリファクタリングすべき箇所を推薦する手法を提案する．機械学習とは，既存のデータを学習することにより，未知のデータの性質を予測する技術のことである．機械学習を用いることで，プロジェクトや開発者の特色に柔軟に対応してリファクタリング箇所を推薦することが可能になる．提案手法では，過去に行われたリファクタリングの情報を学習する．そして，学習した情報をもとに，与えられたソースコードにおいてリファクタリングすべき箇所を開発者に推薦する．

また，提案手法を評価するために，5 つのプロジェクトを対象として実験を行った．その結果，提案手法により，高い精度でリファクタリング箇所を推薦できていることを確認した．

主な用語

リファクタリング
機械学習
リポジトリマイニング
ソフトウェア開発
ソフトウェア保守
品質管理

目 次

1　まえがき

近年，ソフトウェアの保守性を向上させる技術として，リファクタリングが注目されている [1]．リファクタリングとは，ソースコードの外部的なふるまいを保ちつつ，その内部構造を変更・改善する技術である [2]．ソフトウェア開発の中では，様々な種類のリファクタリングが行われている．頻度の高いものとしては，例えば，*Rename*（変数名やメソッド名といった識別子名を変更する），*Move*（変数やメソッドなどを現在の場所から別の場所に移動する），*Extract Method*（メソッドの一部を新たなメソッドとして抽出する）などが挙げられる [3]．既存研究により，ソフトウェアは開発期間が長くなるほどソースコードが複雑になり，保守性が低下する傾向があるということが明らかとなっている [4]．そのため，特に長期間に渡り開発が行われているソフトウェアは，リファクタリングによるソースコードの整理が必要となる．このようなリファクタリングによるソースコードの整理は，ソフトウェアの保守性を向上させ，バグの防止に役立つ．

しかし，多くのプロジェクトはソースコードの量が膨大であり，リファクタリングすべき箇所を手動で特定することは，非常にコストのかかる作業となる．また，手動でリファクタリング箇所を特定する場合，本来リファクタリングすべきであった箇所を開発者が見落としてしまう可能性もある．そこで，効率的にリファクタリングを行うために，ソースコード中から自動でリファクタリングすべき箇所を特定し，開発者に推薦する必要がある．実際に，開発者にリファクタリング箇所を推薦する既存研究がいくつか存在する [5]，[6]，[7]，[8]．また，これらのようなリファクタリング箇所を推薦する手法の多くは，あらかじめ設定した基準をもとに，リファクタリング箇所の特定を行っている．しかし，リファクタリングを行う箇所には明確な基準が存在せず，プロジェクトや開発者によって異なる可能性がある．したがって，ある一定の基準に沿ってリファクタリング箇所を推薦した場合，その推薦した箇所が一部のユーザーにとっては有益であっても，他のユーザーにとっては有益でないかもしれない．そのため，それぞれのプロジェクトや開発者の特色に柔軟に対応してリファクタリング箇所を推薦することが重要となる．

この問題を克服するため，後藤らは機械学習を用いてリファクタリングを行うべき箇所を推薦する手法を提案している [9]．後藤らの提案手法では，*Extract Method* リファクタリングに着目し，過去に行われた *Extract Method* リファクタリングを機械学習によって学習することで，現在のソースコード中に含まれるメソッドのうち，*Extract Method* リファクタリングを行うべきメソッドを開発者に推薦する．このように，機械学習では，プロジェクトごとに実際に行われたリファクタリングの事例を学習するため，そのプロジェクトの特色に柔軟に対応したリファクタリング箇所の推薦を行うことができるようになる．評価実験の結果，後藤らは *Precision* が 60〜85 %程度，*Recall* が 60〜95 %程度の精度でリファクタリング箇所の予測に成功した．後藤らの提案手法では，一部のプロジェクトに対しては 90 %以上と高い予測精度を達成している．しかし一方で，予測精度が 60 %程度に留まっているプロジェクトも存在する．

そこで本研究では，後藤らの提案手法よりもさらに高い精度でリファクタリング箇所を推薦する

ことを目的として，後藤らの手法を改良した．後藤らの提案手法では，メソッドから取得することのできる 21 種類のメトリクス（メソッドのサイズ，複雑度，凝集度など）をもとに学習を行っている．しかし，それらの多くはメソッドの抽象的な情報であり，具体的にメソッドがどのような構文から成り立っているかまでは考慮していない．そこで本研究では，メソッドの構造をより詳細に表現することのできる 84 種類の構文情報に基づく特徴を用いた．また，後藤らの提案手法では，学習に用いるメソッドのうち，リファクタリングが行われていないメソッドの選択をランダムで行っている．しかし，ランダムでメソッドを選択した場合，本来リファクタリングすべきであったにも関わらず，開発者によって見落とされているメソッドを選択してしまう可能性がある．そこで本研究では，リファクタリングが行われたメソッドと同じファイルに含まれるメソッドから，リファクタリングされていないメソッドを選択するようにした．

また，提案手法によるリファクタリング箇所の推薦の精度を評価するために，5 つのオープンソースプロジェクトを対象とした評価実験を行った．その結果，高い精度でリファクタリング箇所の推薦に成功していることを確認した．以下に，実験によって得られた結果をまとめる．

- 提案手法により，*Precision* が 86〜93 %，*Recall* が 79〜97 %と高い精度でリファクタリング箇所の予測を行えることを確認した
- ほとんどの場合において，後藤らの提案手法よりも，本研究における提案手法の方が高い精度で予測を行えることを確認した
- あるプロジェクトから構築したモデルを別のプロジェクトに適応することは，必ずしも有効であるとは限らない

以降，2 章で準備について述べる．3 章で提案手法の詳細について説明し，4 章で提案手法による推薦の精度を調査するための評価実験について説明する．5 章で評価実験の結果について考察を行い，6 章で結果の妥当性について述べる．7 章で関連研究について紹介し，最後に 8 章で本研究のまとめについて述べる．

2 準備

2.1 *Extract Method* リファクタリング

Extract Method リファクタリングとは，既存のメソッドの一部を，新たなメソッドとして抽出するリファクタリング技術である．あるメソッドに対して *Extract Method* リファクタリングを行う場合，まず，そのメソッドの一部を別の新たなメソッドとして抽出する．その後，ソフトウェアの外部的な振る舞いを保つために，必要に応じて抽出したメソッドを修正する．最後に，抽出元となったメソッドから，新しいメソッドとして抽出した箇所を削除し，代わりに抽出したメソッドを呼び出す文を追加する．

図 1 は jEdit の開発過程で行われた *Extract Method* リファクタリングの例である．図 1(a) は *Extract Method* リファクタリングを適用する前のソースコードを表しており，図 1(b) はリファクタリングの適用後のソースコードを表している．この例では，メソッド *insert* 中の 151〜156 行目が，新たなメソッド *prepareGapForInsertion* として抽出されている．そして，メソッド *insert* において抽出された箇所が削除され，代わりに同じ箇所にメソッド *prepareGapForInsertion* を呼び出す文が追加されている．この例では，リファクタリングを適用する前のソースコードにおいて，メソッド *insert* は以下の 2 つの機能を有している．

- バッファ *text* において，文字列 *str* を挿入する位置を調整
- バッファ *text* に文字列 *str* を挿入

そして，メソッド抽出により，メソッド *insert* 中における挿入位置の調整を行う機能がメソッド *prepareGapForInsertion* として抽出されている．この操作により，それぞれの機能がメソッドごとに分割され，ソースコードの理解性が増している．

ソフトウェア開発におけるソースコードの追加や変更により，メソッドはしばしば冗長となることや，複数の機能を有するようになることがある．冗長なメソッドや複数の機能を有するメソッドはソースコードの理解を妨げ，バグの混入を誘発しかねない．*Extract Method* リファクタリングは，そういった複雑なメソッドを分割し，ソースコードの理解性を向上させる．また，*Extract Method* リファクタリングは，複数の機能を有するメソッドを分割することによって各機能に 1 つずつメソッドを割り当てることで，それぞれのメソッドの役割を明確にすることにも役立つ．このような点から，*Extract Method* リファクタリングによるソースコードの整理はソフトウェア開発において非常に効果的である．

2.2 *Extract Method* リファクタリングの推薦手法

機械学習に基づく *Extract Method* リファクタリングの推薦手法

後藤らは，機械学習を用いて *Extract Method* リファクタリングを行うべき箇所を特定する手法を提案している [9]．後藤らは，リファクタリングを行う箇所はプロジェクトや開発者によって異なる

```
148 public void insert(int start, CharSequence str)
149 {
150    int len = str.length();
151    moveGapStart(start);
152    if(gapEnd - gapStart < len)
153    {
154       ensureCapacity(length + len + 1024);
155       moveGapEnd(start + len + 1024);
156    }
157
158    for (int i = 0; i < len; i++)
159    {
160       text[start+i] = str.charAt(i);
161    }
162    gapStart += len;
163    length += len;
164 }
```

(a) 適用前

```
142 public void insert(int start, CharSequence str)
143 {
144    int len = str.length();
145    prepareGapForInsertion(start, len);
146    for (int i = 0; i < len; i++)
147    {
148       text[start+i] = str.charAt(i);
149    }
150    gapStart += len;
151    length += len;
152 }
 ⋮
227 private void prepareGapForInsertion(int start, int len)
228 {
229    moveGapStart(start);
230    if(gapEnd - gapStart < len)
231    {
232       int gapSize = len + 1024;
233       ensureCapacity(length + gapSize);
234       moveGapEnd(start + gapSize);
235    }
236 }
```

(b) 適用後

図 1: *Extract Method* リファクタリングの例

可能性があるとし，プロジェクトや開発者の特色に柔軟に対応してリファクタリング箇所を特定し，開発者に推薦する必要があると主張している．そこで後藤らは，過去の事例にもとづき未来の事象を予測することのできる機械学習に着目している．後藤らの提案手法では，過去に行われたリファクタ

リングについて，21 種類のメトリクスを取得し，機械学習によってそれらの情報を学習する．そして，学習した情報に基づき，ソースコード中においてリファクタリングすべき箇所を開発者に推薦する．後藤らの提案手法で用いられているメトリクスの一覧を表 1 に示す．

後藤らは，提案手法の精度を評価するため，評価実験を行った．その結果，後藤らは，後藤らの提案手法によって *Precision* が 61〜93 %，*Recall* が 57〜95 %の精度でリファクタリング箇所の予測を行えることを確認した．一方で，後藤らの提案手法によるリファクタリング箇所の予測では，対象としたすべてのプロジェクトについて高い精度での予測に成功したわけではない．jEdit を対象とした場合は，*Precision*，*Recall* ともに 60 %であり，予測精度はそれほど高くはない．このように，後藤らの提案手法では，多くの場合においては高精度でリファクタリング箇所を予測することができるが，その一方でプロジェクトごとに精度にばらつきが見られるといった問題がある．

その他の *Extract Method* リファクタリングの推薦手法

Tsantalis らは制御フローグラフを利用して *Extract Method* リファクタリングを行うべき箇所を特定する手法を提案している [13]．彼らの提案手法では，ブロック構造をベースとしたスライシング技術 [14] を採用，拡張しており，メソッド中のプログラム文を制御フローグラフに基づいてブロックに分割している．彼らの提案手法では，まず入力として与えられたメソッドの中から，その中で宣言されているすべての変数を特定する．そして，それぞれの変数について，制御フローグラフに基づいて，プログラム文をスライスの抽出が可能なブロックに分割する．分割したブロックは，*Extract Method* リファクタリングの候補となる．彼らの提案手法では，それぞれの候補について，*Extract Method* リファクタリングによってプログラムの挙動が変化しないようにするために，あらかじめ基準を設け，その基準を満たした候補のみを *Extract Method* リファクタリングの対象箇所として開発

表 1: 後藤らの提案手法で用いられているメソッドのメトリクス

カテゴリ	メソッドの特徴
メソッドのサイズ	メソッド中の文の数（NOS）
メソッドのシグネチャ	メソッドの引数の数（ARG），返り血の有無（RET），アクセスレベル（ACCESS）
複雑度 [10]	サイクロマチック数（CYCLO）
凝集度 [11]	Tightness，Coverage，Overlap
メソッドの構文	ループ数（LOOP），if 文数（IF），casa 文数（CASE），ブロック数（BLOCK），ネストの深さ（NEST）
メソッド内の変数	ローカル変数の数（VAR）
CK メトリクス [12]	WMC，DIT，CBO，NOC，RFC，LCOM
コードクローン	コードクローンの有無（CLONE）

者に提示する．

Silva らは IDE ベースのリファクタリングツールを用いて自動的に *Extract Method* リファクタリングを適用することが可能な箇所を特定する手法を提案している [15]．彼らの提案手法では，メソッド中におけるソースコードのブロック構造を表す階層モデルに基づき，*Extract Method* リファクタリングを行うことができる箇所をすべて特定し，候補とする．そして，3 つの条件（構文条件，プログラム動作の保持に関する条件，品質条件）に基づいて候補をフィルタリングし，フィルタリングによって残った候補をリファクタリング箇所として開発者に推薦する．また，彼らはプログラム理解の向上を目的として，リファクタリングされる可能性の高い順に候補をランク付けした．その結果，Silva らは，彼らの提案手法を用いることで，*Precision*，*Recall* ともに 48 %の精度でリファクタリング箇所のランク付けを行うことに成功した．

これらの手法には，2 つの問題点がある．1 つ目は，プロジェクトに柔軟に対応してリファクタリング箇所を推薦することができない点である．これらの既存研究では，あらかじめ設定した基準に基づいてリファクタリング箇所を特定している．しかし，リファクタリング箇所には本来明確な基準は存在せず，リファクタリング箇所はプロジェクトごとに異なる可能性がある．したがって，既存研究によって推薦したリファクタリング箇所は，あるプロジェクトにとっては有用であっても，他のプロジェクトにとっては有用でないかもしれない．一方で，本手法や後藤らの手法のように機械学習を用いた場合，実際に過去に行われたリファクタリングの事例を学習するため，そのプロジェクトに依存したリファクタリングの特定が可能である．さらに，既存手法では，あらかじめ設定した基準を満たさない箇所は，例えリファクタリングすべき箇所であったとしてもリファクタリング箇所として検出することができない．例えば，Tsantalis らの手法では，*break* 文や *continue* 文，*return* 文を含む箇所はリファクタリング箇所として検出することができない．また，Silva らの手法では，メソッドの長さを基準の 1 つとしてリファクタリング箇所の検出を行うため，長すぎるメソッドや短すぎるメソッドは検出することができない．

2 つ目は，リファクタリングすべき箇所だけでなく，リファクタリングする必要のない箇所も推薦する点である．Tsantalis らの手法や Silva らの手法はいずれも，リファクタリングを行うべきメソッドではなく，リファクタリングを行うことができるすべての箇所を候補として取得し，開発者に提示する．そして，それらの中から開発者がリファクタリングしたい任意の候補を選択するようになっている．したがって，これらの手法では，開発者に推薦されるリファクタリング候補の数が膨大となる．こういった事態を防ぐため，既存研究では候補をフィルタリングしているが，フィルタリング後に推薦される箇所が実際にそのプロジェクトにとって有用であるとは限らない．さらに，フィルタリングによって，本来リファクタリングすべきであった箇所を候補から除去してしまう可能性もある．また，Silva らの手法では特定したリファクタリング箇所について，リファクタリングされる可能性の高い順にランキングを行っている．しかし，その精度は *Precision*，*Recall* ともに 48 %とそれほど高くはない．一方で，本手法や後藤らの手法のように機械学習を用いた場合，過去に行われたリファクタリングを学習することによって，ソースコードの中からリファクタリングすべき箇所のみを

予測することができる．

2.3 *Extract Method* リファクタリングの検出

藤原らは，*Extract Method* リファクタリングを対象とした検出ツール Kenja を公開している [16]．Kenja は，バージョン管理システムの一種である Git[17] で管理されているプロジェクトの開発履歴を入力とし，そのプロジェクトの開発過程において行われた *Extract Method* リファクタリングを検出するツールである．本節では，Kenja を使用する上で必要となるバージョン管理システム について紹介し，その後，Kenja の詳細を説明する．

2.3.1 バージョン管理システム

バージョン管理システムとは，主にプログラム開発においてソースコードやその他のデータを管理するために用いるシステムである．バージョン管理システムの主な機能は以下の 2 つである．

- 開発情報の共有

  バージョン管理システムでは，ソフトウェアの開発情報 (ソースコードなど) はリポジトリと呼ばれるデータ格納庫に保存される．開発者はソフトウェアの開発を行う際，リポジトリから開発情報を手元にコピーし (この動作をチェックアウトと呼ぶ)，修正，変更を加えた後，リポジトリに反映させる (この動作をコミット，あるいはチェックインと呼ぶ)．これにより，同一のプロジェクトを複数の開発者で共有することが可能になる．

- 開発履歴の保持

  バージョン管理システムでは，開発履歴情報をリビジョンと呼ばれる単位で保持している．リビジョンとは開発の状態を表す単位で，リポジトリがコミットを受け付けるたびに生成され，それぞれのリビジョンにはユニークな値（リビジョン番号）が割り当てられる．開発履歴情報にはソースコードなどの開発情報の他，変更を加えた開発者名や日時，変更が加えられたファイル名，変更を加えた開発者の変更に関するメッセージ等のログも含まれている．開発履歴情報を利用すれば，ソースコードに誤った変更を加えてしまった場合などに過去の状態に復元することが可能となる．

バージョン管理システムを用いることで離れた場所にいる開発者間で開発情報の共有が容易になるという利点があるため，商用ソフトウェア開発やオープンソースソフトウェアの開発など，多数の開発者によってソフトウェア開発が行われる際に一般的に使用されている．なお，バージョン管理システムには，Git の他に，Subversion[18] や Concurrent Versions System （CVS）[19] などが存在する．

2.3.2 Kenja の詳細

Kenja は，Git で管理されているプロジェクトの開発履歴を入力とし，そのプロジェクトの開発過程において行われた *Extract Method* リファクタリングを検出する．

Kenja は，検出した各リファクタリングについて，以下の情報を出力する．

- *revision*

  メソッド抽出が行われたリビジョンを一意に特定する値

- *targetMethod*

  メソッド抽出の対象となったメソッド

- *extractedMethod*

  リファクタリングによって新たに作成されたメソッド

- *similarity*

  targetMethod から抽出されたコード片と *extractedMethod* の類似度

なお，類似度の計算には 2-shingles[20] を利用する．2-shingles とは，文書 s の先頭から単語を順に 2 個ずつ取り出し，取り出した単語の組を要素とする集合 $SH(s)$ のことである．2-shingles を用いた文書 s_1，s_2 間の類似度 $similarity(s_1, s_2)$ は以下の式で定義される．

$$similarity(s_1, s_2) = \frac{|SH(s_1) \cap SH(s_2)|}{|SH(s_1) \cup SH(s_2)|} \tag{1}$$

Kenja では，トークンを単語としてコード片の 2-shingles を取得し，類似度を計算している．

Kenja は，各リビジョン間における差分を分析することで，*Extract Method* リファクタリングの検出を行う．以下に，詳細な手順を示す．

手順 1： リビジョン間の変更において新規に作成されたメソッドを見つけ，*extractedMethod* の候補とする

手順 2： その変更において行の追加および削除が行われたメソッドを見つけ，*targetMethod* の候補とする

手順 3： *targetMethod* に追加されたコード片の中で *extractedMethod* が呼び出されているか調べる

手順 4： もし呼び出されていれば，式 1 を用いて，*targetMethod* から削除されたコード片と *extractedMethod* の類似度を計算する

これらの手順を変更のあったクラスごとに行い，最後に閾値以上の類似度を持つ候補をリファクタリングとして検出する．なお，藤原らの調査から，Kenja によるリファクタリング検出は，類似度を 0.3 としたときに最も精度が高くなることが明らかとなっている [16].

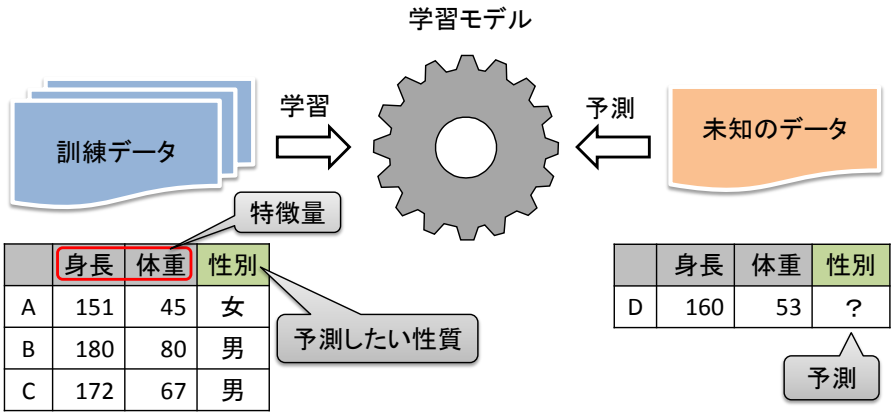


図 2: 機械学習の例

2.4 機械学習

機械学習とは，既存のデータを学習することにより，未知のデータの性質を予測・特定する技術のことである．機械学習では，学習用のデータのことを**訓練データ**，データの学習によって得られる，未知のデータの性質を予測するためのモデルを**学習モデル**と呼ぶ．各データは，そのデータの状態を表す数値や指標の集合と，予測したい性質から構成される．なお，機械学習では，データの状態を表すそれぞれの数値や指標のことを**特徴量**と呼ぶ．

機械学習の例を図 2 に示す．この例では，特徴量は身長および体重，予測したい性質は性別となる．また，A，B，C はすでに性別が判明しているデータであり，このデータを学習することで，性別が判明していない D の性別の予測を行う．機械学習を用いて D の性別を予測するまでの具体的な流れは以下のようになる．

1. まず，性別が判明している A，B，C のデータを学習し，学習モデルを作成する
2. 次に，性別が判明していない D のデータを構築した学習モデルに与える
3. 学習モデルは，学習したデータをもとに，D の性別を自動的に予測する

2.4.1 学習モデル構築アルゴリズム

機械学習は，障害要因の特定，患者の病状に基づく病名診断，市場動向の調査，音声認識や文字認識といったパターン認識などあらゆる場面で用いられており，それぞれの場面ごとにデータの特性や傾向が大きく異なってくる．そのため，それぞれの場面に応じて適切な学習モデルを構築できるよう，学習モデルを構築するための様々なアルゴリズムが提案されている．本節では，それらの中から決定木，ベイジアンネットワーク，ロジスティック回帰の 3 つのアルゴリズムを紹介する．

- 決定木 [21]

  決定木とは，特徴量の値に応じて分岐する木構造モデルを生成する学習アルゴリズムである．決定木上では，枝は特徴量の値に基づく条件分岐を表し，葉はそれらの分岐による結果（予測値）を表す．また，決定木を生成するための代表的なアルゴリズムには，C4.5[22] などがある．

- ベイジアンネットワーク [23]

  ベイジアンネットワークは，複数の変数間の確率的な因果関係を示すグラフィカルモデルを生成する学習アルゴリズムである．各特徴量を 1 つの変数と見なし，変数間の依存関係を有向リンクを用いたグラフ構造によって表す．ベイジアンネットワークでは，個々の変数間の関係は，条件付確率によって示される．

- ロジスティック回帰 [24]

  ロジスティック回帰とは，ベルヌーイ分布に従う変数の統計的回帰モデルを生成する学習アルゴリズムである．ロジスティック回帰では，以下の式に基づき，0〜1 の間の値を取るよう分類モデルを確率化する．

$$Logistic(t) = \frac{1}{1 + exp(-t)} \tag{2}$$

  この分類モデルを用いることで，入力変数の予測を行う．

2.4.2 変数選択

機械学習では，使用するすべての特徴量が予測に良い影響を与えるとは限らない．すなわち，予測精度を下げるノイズとなるものや，結果に何も影響を与えない不必要なものも存在する [25]．そのため，学習モデルを構築するにあたり，与えられた特徴量の中から，予測精度によい効果をもたらす特徴量のみを選択する必要がある．この操作を，**変数選択**と呼ぶ．変数選択には，Wrapper Subset Evaluation[26] や Correlation-based Feature Selection[27]，RelieF[28] など様々なアルゴリズムが存在する．Hall らは 6 種類の変数選択アルゴリズムについて，どのアルゴリズムが優れているか比較を行っている [25]．変数選択アルゴリズムの比較を行うために，Hall らはまずすべての変数選択アルゴリズムについてペアの組を作っている．そして，それらのペアについて同一のデータセットを用意し，それぞれの変数選択アルゴリズムをデータセットに適用する．これにより出された予測結果について，結果が良い方を Win，悪い方を Lose とし，各変数選択アルゴリズムの Win の総数と Lose の総数の差（Win の総数 − $Lose$ の総数）を求める．そして，この総数の差が高い値を示す変数選択アルゴリズムほど精度が良いと見なす．Hall らの調査の結果，良い結果を出す変数選択アルゴリズムは，特徴量の内容や使用する学習モデル構築アルゴリズムにより異なるということが明らかとなっている．例えば，学習モデル構築アルゴリズムとしてベイジアンネットワークを用いた場合は Wrapper Subset Evaluation を使用したときに予測精度が最も良くなり，決定木を用いた場合は，RelieF を使用した時に予測精度が最も良くなるということが示されている．また，前述した Win の

総数と Lose の総数の差でみると，Wrapper Subset Evaluation が最も高いことが判明している．この結果から，Hall らは，多くの場合において Wrapper Subset Evaluation は良い結果を出していると結論づけている．以下では，この Wrapper Subset Evaluation について説明を行う．

Wrapper Subset Evaluation とは，特徴量の部分集合に対して，実際に予測モデルの構築と評価を行うことによって変数選択を行うアルゴリズムである．この手法では，実際に予測モデルの構築を行うために，どの予測モデルを使用するかを変数選択の前に与える必要がある．Wrapper Subset Evaluation は，予測モデルの構築と評価を行うことによって，特徴量の部分集合を評価するため，他の変数選択アルゴリズムよりも良い結果を出力することが多いという長所を持つ一方で，計算コストが高く，他のアルゴリズムに比べて実行に時間がかかるという欠点もある．

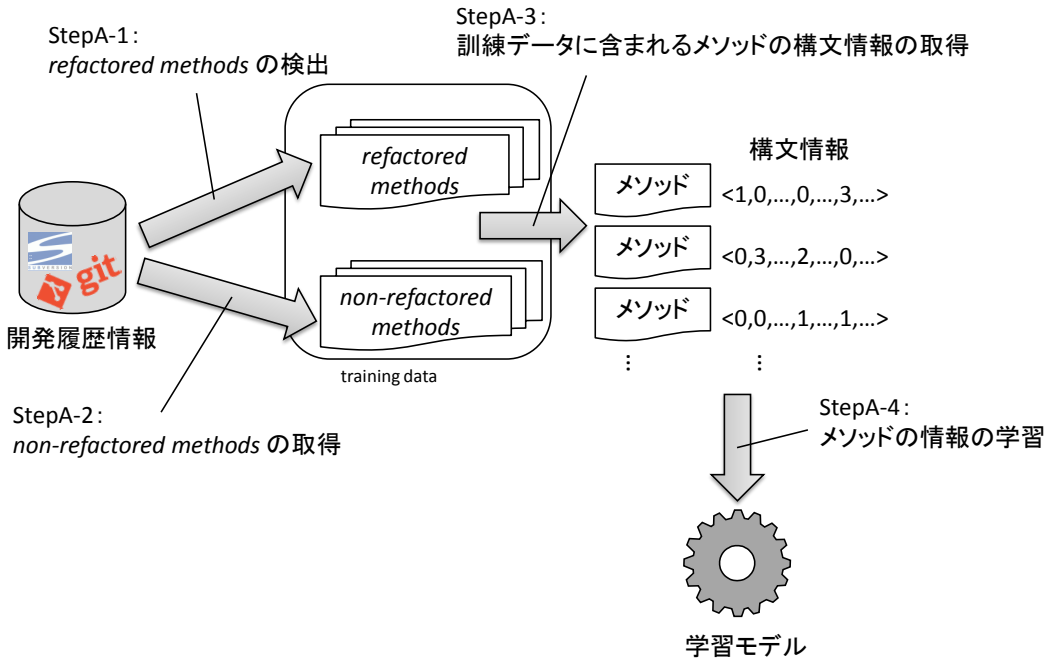


図 3: 学習モデルの構築（StepA）

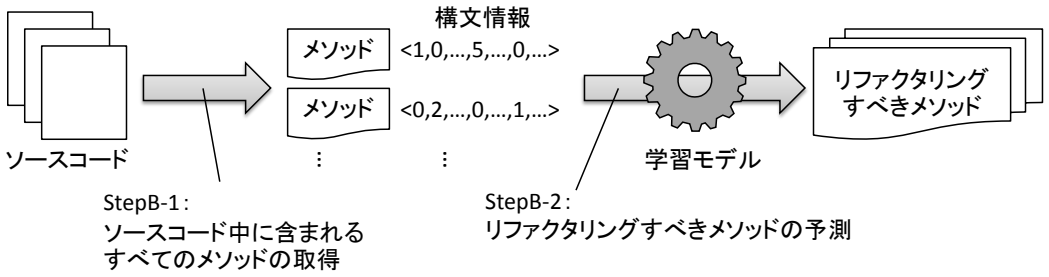


図 4: リファクタリング箇所の予測（StepB）

3 提案手法

3.1 提案手法の概要

本研究では，機械学習を利用し，*Extract Method* リファクタリングを行うべきメソッドを推薦する手法を提案する．本研究における提案手法の入力はソフトウェアの開発履歴情報とソースコードであり，与えられたソースコードの中からリファクタリングすべきメソッドを推薦する．提案手法の流れを図 3 および図 4 に示す．

提案手法は，StepA と StepB の 2 つのステップから構成される．StepA では，入力として与えられたソフトウェアの開発履歴情報からメソッドの情報を学習し，学習モデルを構築する．StepB で

は，StepA で構築した学習モデルを用いて，入力として与えられたソースコード中からリファクタリングすべき箇所を予測する．以下では，それぞれのステップについて説明する．

まず，StepA について概要の説明を行う．StepA はさらに以下の 4 つの段階に分かれている．

StepA-1: まず，ソフトウェアの開発履歴情報を解析し，開発の中で *Extract Method* リファクタリングが行われたメソッドを取得する．本論文中では，ここで取得したメソッドを *refactored methods* と呼ぶ．なお，本研究では，Kenja[16] を用いて *Extract Method* リファクタリングが行われたメソッドを取得する．

StepA-2: 次に，同じソフトウェアの開発履歴情報から，*Extract Method* リファクタリングが行われなかったメソッドを取得する．本論文中では，ここで取得したメソッドを *non-refactored methods* と呼ぶ．本研究では，StepA-1 と StepA-2 の操作によって取得したメソッドを訓練データとして，学習モデルを構築する．

StepA-3: 訓練データの取得が完了したら，その後，訓練データに含まれるそれぞれのメソッドについて，そのメソッドの構文情報を調べる．本研究では，まず訓練データに含まれるそれぞれのメソッドについて，そのメソッドを構成するプログラム要素を取得する．そして，各プログラム要素について，そのプログラム要素がメソッド中で何回出現したかをカウントする．これにより，メソッドの中にどのようなプログラム要素が何回現れたかを表すベクトルを取得することができる．本研究では，このベクトルをメソッドの構文情報として用いる．

StepA-4: 訓練データとして取り出した各メソッドについて，その構文情報と，*Extract Method* リファクタリングが行われたかどうかという情報を合わせて学習し，学習モデルを構築する．この学習モデルに，メソッドの構文情報を与えることで，そのメソッドに対して *Extract Method* リファクタリングを行うべきかどうかを判定できるようになる．

StepA では，これらの手順により，学習モデルを構築する．

次に，StepB について概要の説明を行う．StepB は以下の 2 つの段階に分かれている．

STEPB-1: ユーザによって入力として与えられたソースコードについて，その中に含まれるすべてのメソッドを取り出す．そして，それぞれのメソッドに対して，その構文情報を取得する．

STEPB-2: StepA で構築した学習モデルに対して，StepB-1 で取得したメソッドの構文情報を与えることで，リファクタリングすべきメソッドを特定する．

以上が，本研究における提案手法の概要である．以下では，StepA について，それぞれの段階の詳細を説明する．

3.2 StepA-1 : *refactored methods* の検出

本研究では，Kenja を用いて *Extract Method* リファクタリングが行われたメソッドを取得する．ただし，Kenja によって *Extract Method* リファクタリングとして検出されたものの中には，実際に

は *Extract Method* リファクタリングではないものや，あまり有用ではないものが含まれる．例えば，メソッド中のあるコード片が削除され，偶然同じ箇所に別のメソッドを呼び出す処理が追加された場合，実際には *Extract Method* リファクタリングではないにも関わらず，*Extract Method* リファクタリングとして検出されてしまう．そこで，このような誤検出の事例を除外するために，検出結果のフィルタリングが必要となる．本研究では，*Extract Method* リファクタリングが行われたメソッドから削除されたソースコードとそれにより新たに生成されたメソッドの類似度が，0.3 以上のもののみを取得するようにフィルタリングを行っている．この値は，前述の通り，藤原らの調査 [16] により，Kenja の検出結果のフィルタリングに最も適していることが確認されている値である．

3.3 StepA-2：*non-refactored methods* の取得

このステップでは，*Extract Method* リファクタリングが行われていないメソッド（*non-refactored methods*）を取得する．このステップの流れは以下の通りである．

1. まず StepA-1 で取得したメソッド群の中からランダムで 1 つメソッドを選択する
2. 1. で選択したメソッドが含まれるファイル f と，そのメソッドに対して *Extract Method* リファクタリングが行われたリビジョン r を調べる
3. リビジョン r において，ファイル f に含まれるメソッドのうち，次のリビジョンとの間に *Extract Method* リファクタリングが行われなかったものをランダムで 1 つ取得し，候補とする
4. 3. で取得した候補に対して，開発過程において一度も *Extract Method* リファクタリングが行われていないことを確認する．なお，もしもいずれかのリビジョンにおいてそのメソッドに対して *Extract Method* リファクタリングが行われていた場合は，そのメソッドを *non-refactored method* とは見なさない．

この操作を，学習モデルの構築に必要な数の *non-refcactored methods* が集まるまで，繰り返し行う．提案手法では，学習モデルの構築に用いる *non-refactored methods* の数は，ユーザが任意に決定することが可能である．また，この方法でメソッドの収集を行った場合，*non-refactored methods* として，同じメソッドが複数回選択されることがある．しかし，訓練データの中に同じデータが複数存在すると，そのデータによる影響が大きくなってしまい，予測精度の低下を招く恐れがある．したがって，提案手法では，こういった重複が起こらないように，取得したメソッドと同じものを既に取得している場合は，新たに取得したメソッドを訓練データに加えない．

なお，本研究では，*non-refactored methods* とは，リファクタリングが行われなかったメソッドのことではなく，開発者によってリファクタリングが必要ないと判断されたメソッドのことを指す．この理由は，リファクタリングが行われなかったメソッドは，本来はリファクタリングすべきであったにも関わらず，開発者による見落としや時間的制約などによりリファクタリングが行われなかった可能性があるためである．そのようなメソッドが訓練データに含まれる場合，予測精度が低下す

る恐れがある．一方，*refactored methods* と同じリビジョンおよびファイルに含まれるメソッドは，そうでないメソッドと比較すると，開発者によってリファクタリングをすべきかどうか検討されている可能性が高い．そのため，開発者が意図的に *Extract Method* リファクタリングを行わなかったメソッドのみを取得するために，提案手法では，同じリビジョンおよびファイルのみを対象として *non-refactored methods* を取得している．

non-refactored methods を取得するためのアルゴリズムを，Algorithm1 に示す．Algorithm1 の入力変数は表 2，出力変数は表 3 の通りである．また，Algorithm1 中の各関数の説明を表 4 に示す．

表 2: Algorithm1 の入力変数

変数名	説明
refactoredList	*refactored methods* のリスト（STEP1 で検出されたメソッド群）
num	訓練データとする *non-refactored methods* の数

表 3: Algorithm1 の出力変数

変数名	説明
nonRefactoredList	*non-refactored methods* のリスト

表 4: Algorithm1 の関数

変数名	説明
getSize	引数として与えられたリストに含まれる要素の数を返す
getRandomElement	引数として与えられたリストの中からランダムで 1 つの要素を返す
getRevision	引数として与えられた *refactored method* に対してリファクタリングが行われたリビジョンの番号を返す
getFileName	引数として与えられたメソッドを含むファイルの名前を返す
getRandomMethod	第二引数として与えられたリビジョンにおいて，第一引数として与えられたファイル中に含まれるメソッドを 1 つランダムで返す
isInList	第一引数として与えられたメソッドのリストが，第二引数として与えられたメソッドを含む場合は *true*，それ以外は $false$ を返す．具体的には，第二引数として与えられたメソッドのシグネチャとファイル名の療法が一致するメソッドが，第一引数として与えられたメソッドのリスト中に存在した場合に，*true* を返し，それ以外は $false$ を返す．
add	第二引数として与えられたメソッドを，第一引数として与えられたリストに追加する

Algorithm 1 Detecting *non-refactored methods*

Input: $refactoredList$, num

Output: $nonRefactoredList$

```
1: nonRefactoredList ← ∅
2: while num ≠ getSize(nonRefactoredList) do
3:    method ← getRandomElement(refacteredList)
4:    revision ← getRevision(method)
5:    file ← getFileName(method)
6:    candidate ← getRandomMethod(revision, file)
7:    if !isInList(nonRefactoredList, canadidate) then
8:       if !isInList(refactoredList, candidate) then
9:          add(nonRefactoredList, candidate)
10:      end if
11:   end if
12: end while
13: return nonRefactoredList
```

3.4 StepA-3：訓練データに含まれるメソッドの構文情報の取得

訓練データの取得が完了したら，その後，訓練データに含まれるそれぞれのメソッドについて，構文情報を調べる．本研究では，メソッドの構文情報をベクトルとして表す（これ以降，本論文ではメソッドの構文情報を表すベクトルのことを**状態ベクトル**と呼ぶ）．状態ベクトルでは，それぞれのプログラム要素の出現回数が 1 つの要素となる．なお，プログラム要素とは，例えば，文（*if* 文, *switch* 文など），識別子，数字，記号（“ . ”，“ == ”など）を指す．

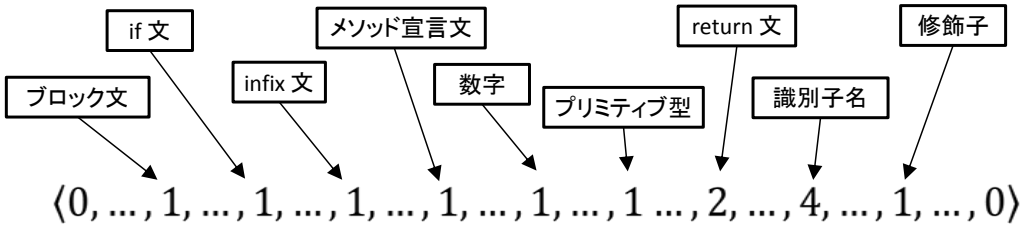
本研究では，Murakami らの既存研究と同様の方法を用いて，状態ベクトルを取得する [29]．具体的には，JDT[30] によって生成される AST（Abstract Syntax Tree）を元に，構文情報を取得する．提案手法では，この構文木における各ノードの種類を 1 つのプログラム要素として扱う．なお，JDT では 84 種類のノードが定義されているため，本研究で用いる状態ベクトルは 84 次元となる．

状態ベクトルの例を図 5 に示す．図 5(a) がメソッドのソースコードであり，図 5(b) がそのメソッドの状態ベクトルである．このメソッドの中には，84 のプログラム要素のうち，9 つが出現している．したがって，このメソッドの状態ベクトルでは，出現した 9 つの要素の値はメソッド中におけるその要素の出現回数となり，それ以外の 75 の要素の値は 0 となる．例えば，*return* 文はメソッド中で二回出現しているため，状態ベクトルにおいて *return* 文を指す要素の値は 2 となる．一方，*while* 文はメソッド中に記述されていないため，状態ベクトルにおいて *while* 文を指す要素の値は 0 となる．

2.2 節で紹介した後藤らの手法では，メソッドのサイズ，複雑度，凝集度など，メソッドの抽象的な情報に基づいてリファクタリング箇所の予測を行っている [9]．これらのメトリクスは，まったく

```
1 public int hoge(){
2  if(n == 0)
3   return a;
4  return b;
5 }
```

(a) ソースコード



(b) 状態ベクトル

図 5: 状態ベクトルの例

異なる構造のプログラムであっても，同一の値を示すことがある．このような場合，まったく異なる構造のプログラムがメトリクスの上では類似したプログラムであると識別されてしまうといったことが起こり得る．しかし，リファクタリングはプログラムの構造を変更するものであるため，異なるプログラム同士を類似したものとして扱ってしまうと，メソッドの特徴を適切に捉えることができない可能性がある．以上の理由から，本研究では，プログラムを具体的な構造ごとに識別するため，プログラムの構造を詳細に表現することができる構文情報に着目した．

さらに StepA-3 では，状態ベクトルを取得した後，取得した状態ベクトルを用いて，それぞれの *non-refactored method* と *refactored method* の類似度を計算する．そして，いずれかの *refactored method* と類似している *non-refactored method* が存在した場合，その *non-refactored method* を訓練データから除外する．この操作を行う理由は，もしも *refactored method* と類似した *non-refactored method* が訓練データ中に存在した場合，予測精度が低下する恐れがあるためである．なお，類似度の計算には，コサイン類似度 [31] を用いる．コサイン類似度とは 2 つのベクトル間の類似度のことである．コサイン類似度は 0〜1 の値を取り，値が 1 に近いほど，2 つのベクトルは類似しているということになる．なお，2 つのベクトル間のコサイン類似度の計算式は，以下の式で表わされる．

$$cos(\vec{p}, \vec{q}) = \frac{\vec{p} \cdot \vec{q}}{|\vec{p}||\vec{q}|} \tag{3}$$

各 *non-refactored method* について，そのメソッドとのコサイン類似度が 0.95 以上の *refactored method*

が存在した場合に，そのメソッドと類似した *refactored method* が訓練データに含まれるとし，そのメソッドを *non-refactored method* から除外する．なお，*non-refactored method* を除外すると，訓練データに含まれる *non-refactored method* の数が減少してしまう．そのため，本研究では，再度 StepA-2 の操作を行い，減少した数だけ *non-refactored method* を取得する．

3.5 StepA-4：メソッドの情報の学習

この時点で，訓練データに含まれるずべてのメソッドについて構文情報の取得が完了している．次に，提案手法では，訓練データに含まれるそれぞれのメソッドについて，その構文情報と，*Extract Method* リファクタリングが行われたかどうかという情報を機械学習によって学習し，学習モデルを構築する．この学習モデルに，メソッドの構文情報を与えることで，そのメソッドに対して *Extract Method* リファクタリングを行うべきかどうか予測することが可能になる．なお，本研究では，学習モデルの構築にあたり，データマイニングツール Weka[32] をデフォルト設定のまま用いている．

4 評価実験

4.1 実験対象

本研究では，提案手法の評価を行うために，提案手法をツールとして実装した．そして，そのツールを用いて，5 つのオープンソースプロジェクトを対象として実験を行った．なお，本研究で実装したツールは，Java 言語で開発されたプロジェクトのみを解析対象としているため，実験対象プロジェクトはいずれも Java 言語で開発されているものを選択した．そして，それぞれのプロジェクトについて，3 節で説明した手順により，*refactored methods* と *non-refactored methods* を同じ数だけ取得し，それらをデータセットとして実験を行った．ただし，*refactored methods* については，後藤らの手法で用いられたメソッドと同一のものを用いている．対象としたプロジェクトを表 5 に示す．なお，表中の *refactored* は *refactored methods* のことを示し，*non-refactored* は *non-refactored methods* のことを示している．

4.2 実験結果

4.2.1 提案手法によって構築した学習モデルの精度

本実験では，構築した学習モデルの評価手法として交差検証法を用いた．交差検証法では，まずデータセットを N 個のブロックに分割する．なお，データセットの分割は，各ブロックのデータ数が同程度になるように行う．交差検証法では，分割したブロックのうち $N-1$ 個のブロックを訓練データとし，残りの 1 個のブロックをテストデータとして評価を行う．具体的には，まず訓練データを用いて学習モデルを構築する．その後構築した学習モデルを用いてテストデータの予測を行い，精度を計測する．この操作を，各ブロックをテストデータとした場合についてそれぞれ行い，それらの精度の平均を取る．なお，本研究では $N = 10$ として実験を行う．実験では，この交差検証法を 100 回繰り返したときの精度の平均を，最終的な学習モデルの精度とする．

また，本実験では評価尺度として *Precision* と *Recall* の 2 つの指標を用いた．これらの説明を表

表 5: 実験対象プロジェクト

Project	対象リビジョン数	総行数		データセットの数	
		開始	最新	*refactored*	*non-refactored*
Ant	12,783	0	260,624	766	766
ArgoUML	17,748	0	370,161	740	740
jEdit	5,787	2,851	181,453	502	502
jFreeChart	916	270,121	327,865	90	90
Mylyn	8,414	28,158	166,149	490	490

6 にまとめる．また，本実験では，学習モデル構築アルゴリズムとして決定木，ベイジアンネットワーク（BayesNet），ロジスティック回帰（Logistic）の 3 つのアルゴリズムを用い，それぞれについて交差検証法を用いて評価を行った．

評価実験の結果を，表 7 にまとめる．表より，*Precision* が 86〜93 %，*Recall* が 79〜97 %と高い精度でリファクタリング箇所を特定できている．また，いずれの学習モデル構築アルゴリズムについても，多くの項目が 90 %を超えている．*Precision* については，90 %を超える項目は 15 項目中 14 項目であり，*Recall* については，15 項目中 6 項目が 90 %を超えている．このことから，特に *Precision* が高い数値を示しているということが分かる．これは，提案手法によってリファクタリングすべきと予測されたメソッドのうち，多くが実際にリファクタリングすべきであったということを表している．以上の結果より，提案手法は，非常に高い精度で *Extract Method* リファクタリングを行うべきメソッドを予測できていると言える．また，特に *Precision* が高いことから，リファクタリングすべきメソッドのみを的確に推薦することができているということが分かる．

4.2.2 後藤らの手法との精度比較

表 8 は，後藤らの提案手法の精度を表している．なお，表中の下線部は，本研究における提案手法の精度（表 7）よりも高かった項目である．表より，ほとんどの項目において，本研究における提案手法の方が後藤らの提案手法よりも高い精度で予測に成功していることが分かる．また，後藤らの

表 6: 評価尺度

評価尺度	説明
Precision	構築した学習モデルによって *Extract Method* リファクタリングを行うべきであると予測されたメソッドのうち，実際に *Extract Method* リファクタリングが行われたものの割合
Recall	*Extract Method* リファクタリングが行われたメソッドのうち，実際に構築した学習モデルによって *Extract Method* リファクタリングを行うべきであると予測されたクローンセットの割合

表 7: 提案手法によって構築した学習モデルの精度

	Ant		AlgoUML		jEdit		jFreeChart		Mylyn	
	P	*R*	*P*	*R*	*P*	*R*	*P*	*R*	*P*	*R*
決定木	0.90	0.91	0.91	0.92	0.90	0.87	0.93	0.97	0.91	0.89
BayesNet	0.90	0.79	0.90	0.82	0.91	0.85	0.86	0.97	0.93	0.86
Logistic	0.91	0.86	0.92	0.88	0.91	0.89	0.91	0.95	0.93	0.90

提案手法において *Precision*，*Recall* ともに 6 割程度の予測精度であった jEdit に関しては，本研究における提案手法では，他のプロジェクトとほとんど変わらない高い精度でリファクタリング箇所を予測できている．

さらに，後藤らの研究では，テストデータ中に含まれる *refactored methods* の割合を変化させて実験を行っている．これは，提案手法を実際のソースコードへ適用することが可能かどうか調査するためである．後藤らの実験で扱っているデータセットは，同数の *refactored methods* と *non-refactored methods* から構成される．しかし，実際のソースコード中に含まれる *refactored methods* の数と *non-refactored methods* の数は必ずしも一致せず，多くの場合は *refactored methods* に比べて多数の *non-refactored methods* が存在する．そこで，後藤らはテストデータ中に含まれる *refactored methods* の割合を 40 %，30 %，20 %，10 %と減少させていき，それぞれの場合について精度を調査している．なお，*refactored methods* の割合を変えるのはテストデータのみであり，学習モデルの構築に用いる訓練データにおける *refactored methods* の割合は変化させない．本研究でも，これと同様の実験を行った．以下では，その結果を述べる．

図 6 は，本研究における提案手法の *Precision* の変化を表しており，図 7 は，後藤らの提案手法の *Precision* の変化を表している．図より，いずれの手法についても，データセット中に含まれる *refactored methods* の割合が少なくなるほど，*Precision* が低下していることが分かる．低下の度合いに着目すると，本研究における提案手法では，*refactored methods* の割合が 50 %の時点と 10 %の時点とでは，*Precision* に 30 %程度の低下が見られる．一方で，後藤らの提案手法では，*refactored methods* の割合が 50 %の場合と 10 %の場合で，*Precision* に 40〜50 %ほどの低下が見られる．このことから，本研究における提案手法の方が，後藤らの提案手法と比較して，*refactored-methods* の割合が減少した場合における精度の低下の度合いが少ないと言える．また，*Precision* の値そのものについても，ほとんどの地点において，本研究における提案手法の方が高くなっている．

図 8 は，本研究における提案手法の *Recall* の変化を表しており，図 9 は，後藤らの提案手法の *Recall* の変化を表している．図より，いずれの手法についても，データセット中に含まれる *refactored methods* の割合に依存せず，*Recall* の値がほぼ一定であることが分かる．また，*Recall* の値そのものに着目すると，ほとんどの地点において，本研究における提案手法の方が後藤らの提案手法よりも高くなっている．

表 8: 後藤らの手法によって構築した学習モデルの精度

	Ant		AlgoUML		jEdit		jFreeChart		Mylyn	
	P	*R*	*P*	*R*	*P*	*R*	*P*	*R*	*P*	*R*
決定木	0.78	0.84	0.78	0.80	0.61	0.65	0.85	0.92	0.77	0.76
BayesNet	0.79	0.79	0.76	0.76	0.67	0.57	0.87	0.94	0.78	0.82
Logistic	0.83	0.77	0.79	0.69	0.62	0.65	0.93	0.95	0.80	0.73

以上の点から，多くの場合において，*refactored methods* の割合によらず，本研究における提案手法の方が後藤らの提案手法よりも優れていると言える．

4.2.3 あるプロジェクトから構築した学習モデルを他のプロジェクトへ適用した場合の精度

次に，あるプロジェクトから構築した学習モデルを，他のプロジェクトに適用する実験を行った．本研究は，プロジェクトや開発者の特色に柔軟に対応したリファクタリング箇所の推薦を行うことを目的としているため，この実験は本来の目的とは矛盾している．しかし，同一のプロジェクト間だけでなく，異なるプロジェクト間でも学習モデルを適用可能であることが証明されれば，より多くの場面で提案手法を活用することができるようになる．

例えば，開発が始まって間もないプロジェクトなどは，リファクタリング作業があまり行われていないため，過去の開発履歴情報から十分なリファクタリング情報を得ることができない．そのため，開発期間が短いプロジェクトに対しては，提案手法の適用が難しいという問題点がある．しかし，もし異なるプロジェクト間での学習モデルの応用が可能であれば，開発期間が短いプロジェクトに対しても，既に十分にリファクタリング作業が行われているプロジェクトから構築した学習モデルを応用することで，リファクタリングの推薦ができるようになる．

表 9: 異なるプロジェクト間において学習モデルを適用した場合の精度

		Ant		ArgoUML		jEdit		jFreeChart		Mylyn	
		P	R	P	R	P	R	P	R	P	R
Ant	決定木	–	–	0.85	0.66	0.87	0.83	0.60	0.48	0.84	0.82
	BayesNet	–	–	0.91	0.75	0.89	0.83	0.74	0.34	0.91	0.80
	Logistic	–	–	0.90	0.80	0.87	0.78	0.75	0.60	0.85	0.90
ArgoUML	決定木	0.84	0.83	–	–	0.92	0.70	0.78	0.40	0.84	0.89
	BayesNet	0.85	0.81	–	–	0.92	0.75	0.73	0.59	0.88	0.88
	Logistic	0.85	0.82	–	–	0.90	0.72	0.84	0.89	0.86	0.89
jEdit	決定木	0.89	0.79	0.92	0.79	–	–	0.74	0.50	0.89	0.85
	BayesNet	0.89	0.77	0.92	0.75	–	–	0.78	0.42	0.91	0.81
	Logistic	0.90	0.80	0.90	0.72	–	–	0.66	0.34	0.89	0.86
jFreeChart	決定木	0.84	0.77	0.87	0.76	0.84	0.70	–	–	0.87	0.80
	BayesNet	0.86	0.75	0.92	0.66	0.89	0.78	–	–	0.91	0.78
	Logistic	0.67	0.07	0.44	0.34	0.38	0.03	–	–	0.52	0.04
Mylyn	決定木	0.87	0.76	0.90	0.77	0.87	0.78	0.81	0.43	–	–
	BayesNet	0.89	0.78	0.92	0.77	0.89	0.84	0.75	0.42	–	–
	Logistic	0.90	0.72	0.92	0.75	0.88	0.75	0.82	0.20	–	–

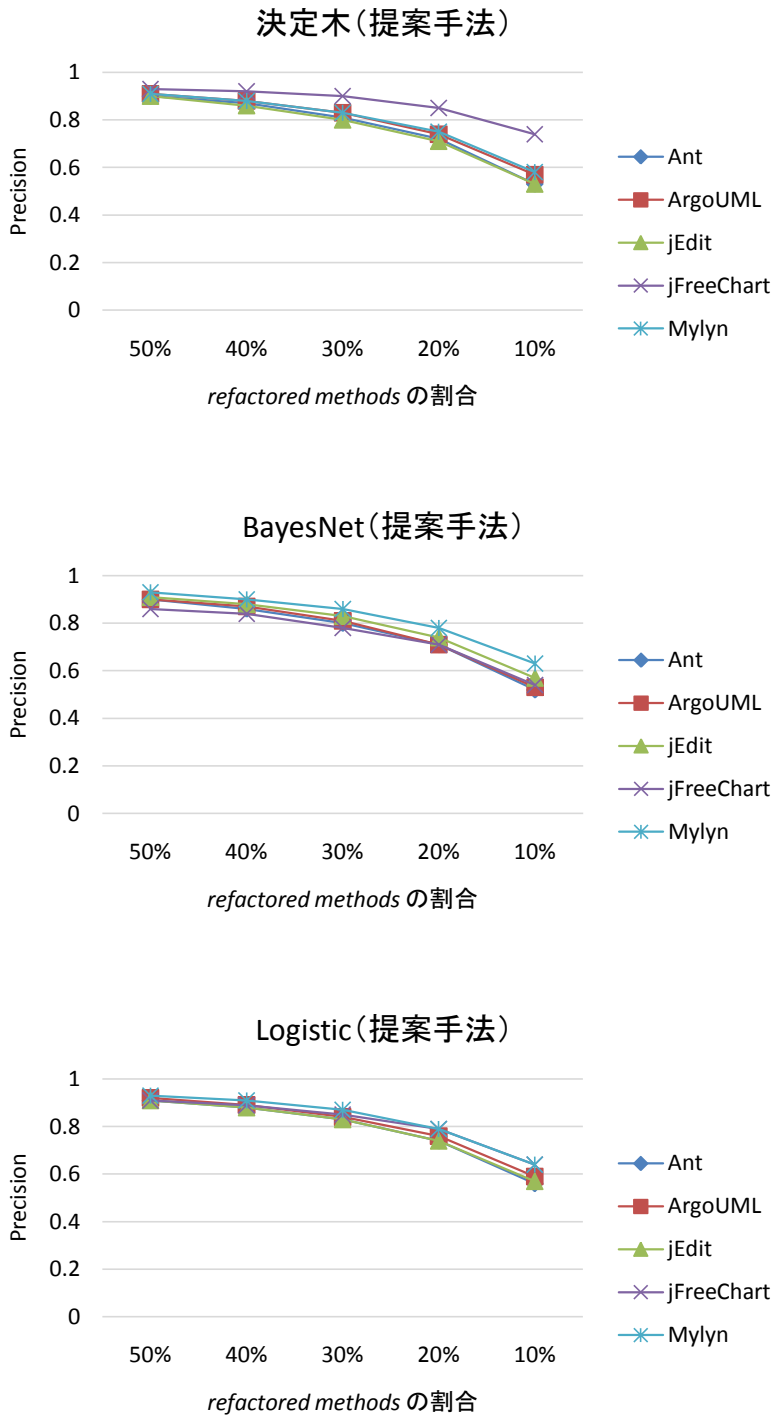


図 6: *refactored methods* の割合を変化させた場合の *Precision*（本研究における提案手法）

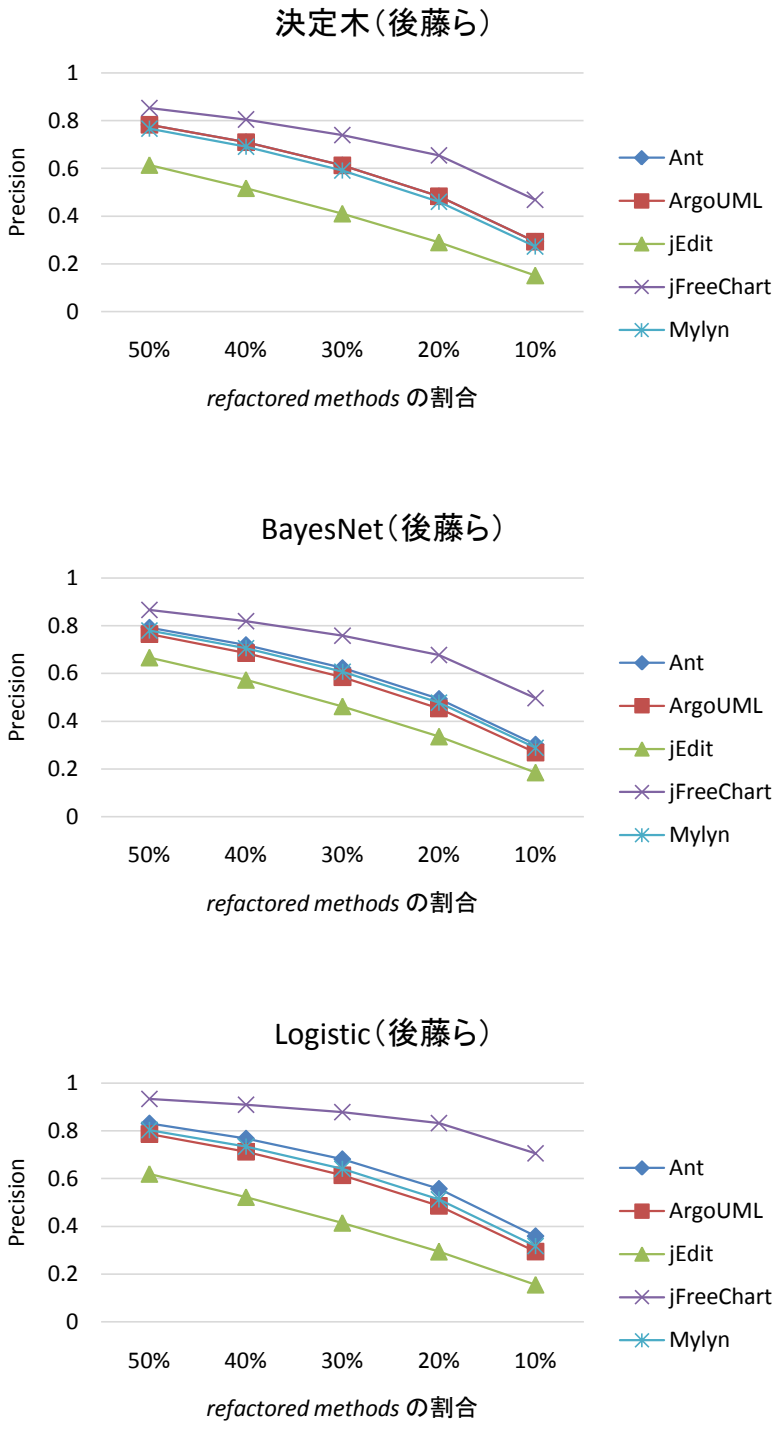


図 7: *refactored methods* の割合を変化させた場合の *Precision*（後藤らの提案手法）

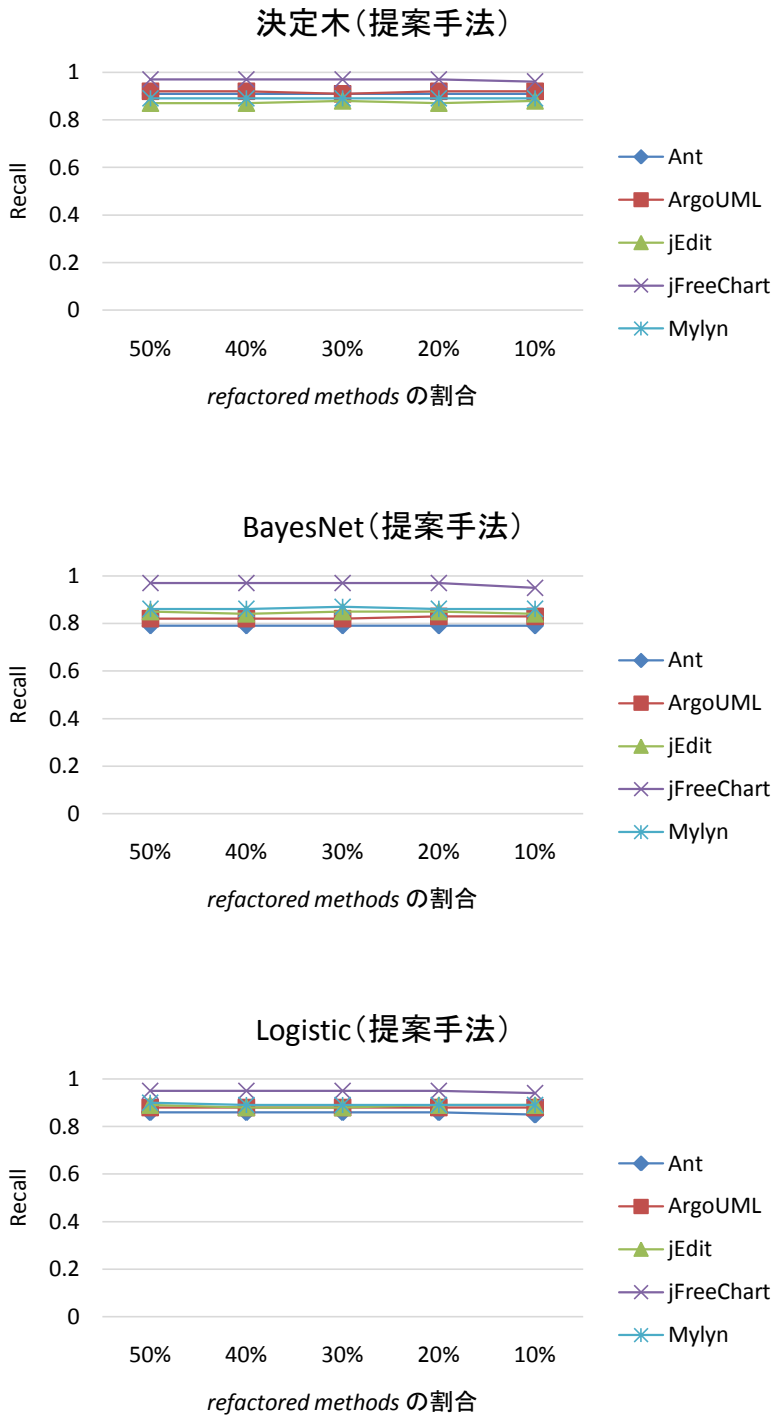


図 8: *refactored methods* の割合を変化させた場合の *Recall*（本研究における提案手法）

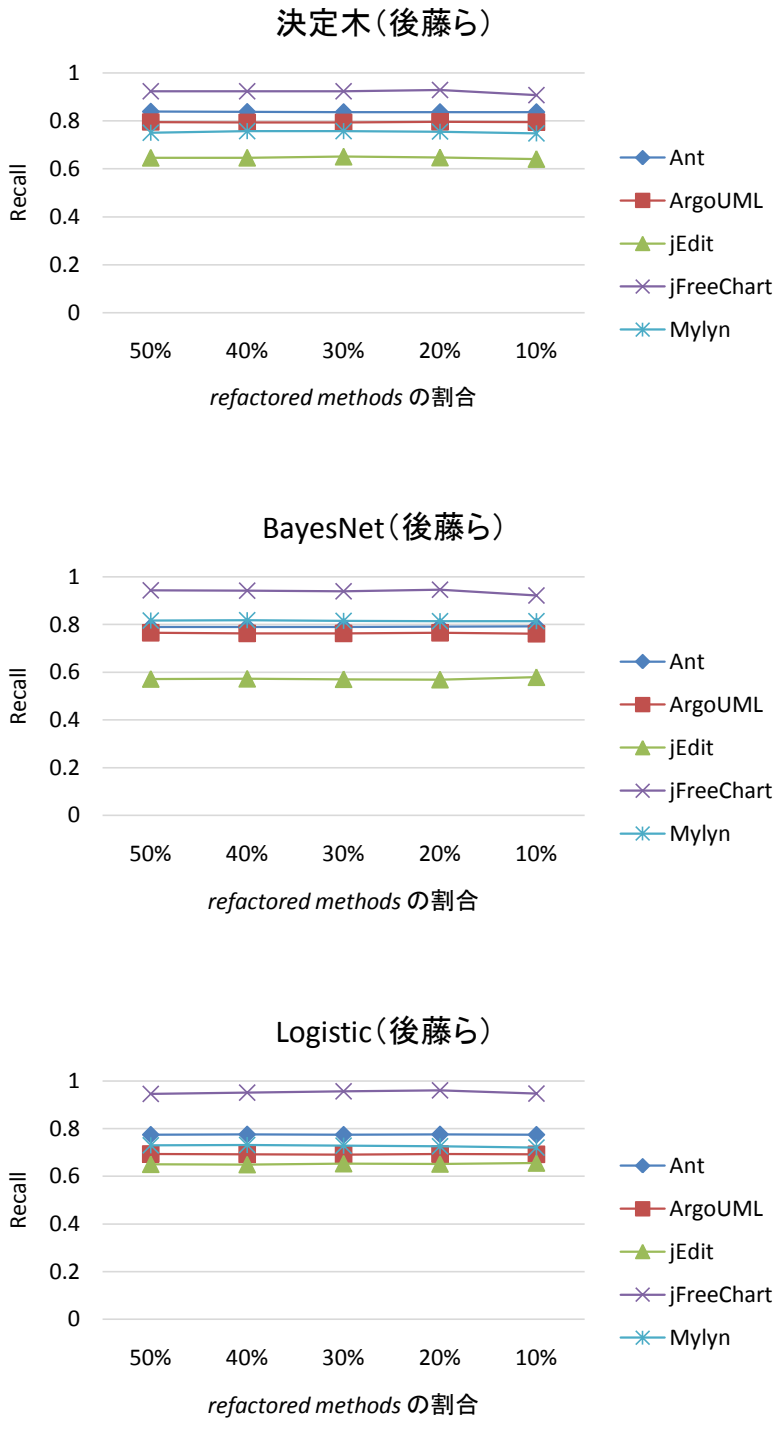


図 9: *refactored methods* の割合を変化させた場合の *Recall*（後藤らの提案手法）

そこで，それぞれの対象プロジェクトについて，相互的に 1 つのプロジェクトから構築した学習モデルを別のプロジェクトに適応し，精度を計測した．この実験結果を表 9 に示す．表の列は学習モデルを構築するために用いたプロジェクトを表し，行はその学習モデルを適用したプロジェクトを表す．なお，表中の下線部は，同一のプロジェクト間における予測結果（表 7）よりも高かった項目である．表より，ほとんどの項目について，同一のプロジェクト間における予測精度よりも異なるプロジェクト間における予測精度の方が低いことがわかる．また，*Precision* は 38〜92 %，*Recall* は 3〜89 %であり，同一のプロジェクト間における予測結果（表 7 果）ほど高くはない．プロジェクト別に見ると，jFreeChart を用いて構築した学習モデルを他のプロジェクトに適用した場合に，同一のプロジェクト間において学習モデルを適用するよりも精度が大きく低下していることが分かる．特に *Recall* については，最大で精度が 86 %低下している．さらに，他のプロジェクトから構築した学習モデルを jFreeChart に適用した場合も，*Precision* については 10〜30 %，*Recall* については 5〜75 %ほど精度が低下している．このように，異なるプロジェクト間において学習モデルを適用した場合に精度が大きく低下してしまうプロジェクトが存在することから，あるプロジェクトから構築した学習モデルを他のプロジェクトへ適用することは必ずしも有効であるとは言えない．

一方で，この結果は別の新たな知見を生む．異なるプロジェクト間における予測精度が同一のプロジェクト間における予測精度よりも低いことは，プロジェクトごとにリファクタリングを行う箇所の傾向が異なるということと同等である．すなわち，リファクタリング箇所 はプロジェクトに依存している可能性が高い．したがって，あらかじめ設けた基準や条件に合致したもののみをリファクタリング箇所として推薦する手法では，こういったプロジェクトに依存したリファクタリング箇所を特定することができない恐れがある．しかし，提案手法では，実際にそのプロジェクトに対して行われたリファクタリングの事例をもとにリファクタリングすべき箇所の特定を行うため，プロジェクトに依存したリファクタリング箇所を見落とす危険性は低い．以上より，プロジェクトごとにリファクタリングを行う箇所の傾向が異なる可能性があることから，提案手法によりそういったプロジェクトに依存したリファクタリング箇所を特定することが必要であろう．

5 考察

5.1 リファクタリング箇所の特徴

4.2.3 節の結果より，プロジェクトごとにリファクタリング箇所の傾向が異なるであろうことが分かった．そこで，実際にプロジェクトごとにリファクタリング箇所にどのような特色があるのか調査を行った．その結果，2 つの知見が得られた．

まず 1 つ目に，いずれのプロジェクトについても，長いメソッドほどリファクタリングされやすいことが分かった．これは，長いメソッドを *Extract Method* リファクタリングによって短く簡約化することで，可読性を向上させ，開発者によるコードを理解を高めるためであると考えられる．

2 つ目に，リファクタリング箇所の特定のために重視されるプログラム要素はプログラムごとに必ずしも一致しないということが分かった．実験対象としたすべてののプロジェクトについて，構築した学習モデルによるリファクタリング箇所の特定においてメソッド呼び出し文の出現回数が重視されていた．具体的には，メソッド呼び出し文が多いものはリファクタリング候補となりやすかった．一方で，あるプロジェクトではリファクタリング箇所の特定のために重視されるプログラム要素が，別のプロジェクトでは重視されないというケースも存在した．例えば，jEdit では *while* 文の出現回数がリファクタリング箇所の特定に大きな影響を与えていたが，Mylyn ではそれはまったく考慮されていなかった．

以上より，プロジェクトごとに実際にリファクタリング箇所の傾向が異なることがあるため，機械学習によってプロジェクトの特色に対応したリファクタリング箇所を特定する必要がある．

5.2 機械学習によるリファクタリング箇所の推薦の有用性

機械学習を用いたリファクタリング箇所の推薦について，著者らは 2 つのメリットを挙げる．1 つ目は，プロジェクトや開発者の特色に柔軟に対応した推薦を行うことができる点である．リファクタリング箇所は，プロジェクトによって異なる可能性があるため，リファクタリング箇所を特定するための明確な基準を設けることはできない．しかし，機械学習により実際に行われたリファクタリングのパターンを学習することで，事前に基準を設定せずともそのプロジェクトにとって有用なリファク

表 10: 学習モデルの構築において選択された特徴量のランキング

Project	Top-1	Top-2	Top-3
Ant	メソッド呼び出し文	変数宣言文	*catch* 節
ArgoUML	メソッド呼び出し文	*return* 文	数字
jEdit	変数宣言文	*while* 文	メソッド呼び出し文
jFreeChart	メソッド呼び出し文	*this* 節	型名
Mylyn	メソッド呼び出し文	*switch* 文	*synchronized* 文

タリング箇所を特定することができる．

2 つ目は，リファクタリング箇所の見落としを防止する点である．ソースコードの量はしばしば膨大であるため，リファクタリング箇所を手動で特定する場合，開発者はリファクタリング箇所を見落とす可能性がある．例えば，開発過程では，異なるメソッド中に存在する互いに類似したコード片（重複コード）を抽出し，1 つのメソッドとして集約することがしばしば行われる．しかし，開発者の見落としによって，集約すべきであったコード片の一部を見落とす可能性がある．実際に，Higo らの調査では，*Extract Method* リファクタリングによって重複コードを集約する際に，それらの一部を見落とすことがあるということを確認している [33]．しかし，機械学習を用いて過去に実際に行われた事例を学習することで，こういったリファクタリングの見落としを防ぐことができる．このように，過去にリファクタリングが行われた箇所と類似した箇所を検出することができるため，機械学習によるリファクタリング箇所の推薦は有用である．

5.3 推薦の対象とする単位

提案手法では，メソッド単位でのリファクタリング箇所の推薦を行っており，具体的にメソッド中のどの箇所をリファクタリングすべきかまでは特定することができない．この問題については，メソッド中におけるリファクタリング箇所を特定することができる外部ツール（JDeodorant[34] など）を用いることによって解決することができる．すなわち，提案手法によってリファクタリングすべきメソッドを特定した後，外部ツールを用いてメソッド中におけるリファクタリング箇所を特定すればよい．

6　結果の妥当性

計測対象

本実験で対象としたソフトウェアは 5 つであり，Java で開発されたオープンソースソフトウェアに限定して調査を行った．このため，より多くのソフトウェアを対象として実験を行った場合や，異なる言語で記述されたソフトウェアや商用ソフトウェアを対象とした場合は，本研究で得られた結果と異なる結果が得られる可能性がある．

データマイニングツールの設定

本研究では，Weka のライブラリを用いて学習モデルを構築している．Weka では，学習モデルの構築にあたり，決定木の枝刈りなどといったオプションを選択することができる．しかし，本研究では，追加のオプションを設定せず，Weka をデフォルト設定のまま用いている．したがって，学習モデルを構築する際にオプションを選択した場合や，Weka 以外のデータマイニングツールを使用した場合は，本研究で得られた結果と異なる結果が得られる可能性がある．

Kenja の検出結果

本研究では，*refactored methods* を検出するために，*Extract Method* リファクタリングの検出ツールである Kenja を用いている．しかし，Kenja によって検出されたメソッドは，そのすべてが正しいとは限らない．すなわち，Kenja の検出結果には，*Extract Method* リファクタリングが行われたメソッド以外のメソッドも含まれている可能性がある．さらに，Kenja は，開発過程において *Extract Method* リファクタリングが行われたメソッドをすべて検出できているわけではなく，検出漏れも存在する．したがって，実際に *Extract Method* リファクタリングが行われたメソッドをすべて正確に取得し，それらを用いて実験を行った場合は，本研究で得られた結果と異なる結果が得られる可能性がある．

データセット

本研究では，同数の *refactored methods* と *non-refactored methods* から構成されるデータセットを用いて実験を行った．しかし，実際のソースコード中に含まれる *refactored methods* と *non-refactored methods* の数は必ずしも一致するとは限らず，多くの場合，*non-refactored methods* の方が多く含まれる．したがって，実際のソースコード中に含まれるすべてのメソッドを用いて実験を行った場合は，本研究で得られた結果と異なる結果が得られる可能性がある．

状態ベクトル

本研究では，メソッドの状態ベクトルを用いることにより，プロジェクの特色に柔軟に対応してリファクタリング箇所の推薦を行っている．状態ベクトルを用いた理由には，状態ベクトルはメソッドの詳細な構文情報を保持しているため，プロジェクトの特色を反映しているのではないかといった推定が存在する．しかし，実際には状態ベクトルがプロジェクトの特色を反映しているとは限らないため，プロジェクトの特色に柔軟に対応してリファクタリング箇所を推薦できていない可能性がある．したがって，状態ベクトルが実際にプロジェクトの特色を反映しているか検証することが必要である．

7　関連研究

リファクタリングは，ソフトウェアの保守性を向上させるための重要な技術であり，ソフトウェア開発において多くの場面で用いられている．Mens らは，サーベイ論文の中でリファクタリングに関する既存研究についてまとめている [1]．サーベイ論文の中で Mens らは，リファクタリング作業はいくつかの段階に分割することができると主張している．Mens らは，以下のようにファクタリングの段階を定義している．

1. リファクタリングの対象とする箇所を特定
2. 1. で特定したリファクタリング箇所に対して適用するリファクタリングを決定 [35]
3. リファクタリングを適用によりソフトウェアの振る舞いが変化しないかどうかを調査 [36], [37]
4. リファクタリングを適用
5. リファクタリングがソフトウェアの品質におよぼす影響を評価 [38], [39]
6. リファクタリングを適用した箇所との整合性を保たせるため，他の箇所を調整 [40], [41]

本節では，1. のリファクタリングの対象とする箇所を特定する手法をいくつか紹介する．

Emden らは，ソースコード中の *Code Smells* を検出し，可視化する手法を提案している [5]．*Code Smells* とは，ソースコードに深刻な問題が存在することを示す何らかの兆候のことを指す [2]．Emden らの実装したツールは，まず，長すぎるメソッド，重複コード，機能の多すぎるクラスといった *Code Smells* を検出する．ツールでは，あらかじめそれぞれの *Code Smells* について条件を設定しており，その条件に合致した場合に，該当するコード片を *Code Smells* として検出する．そして，*Code Smells* の検出後，それらの情報を *Code Smells* の専用ブラウザにグラフとして映し出す．出力されるグラフでは，パッケージやクラス，メソッドといったプログラムの要素がノードで表され，継承やメソッド呼び出しといった要素間の関係が辺で表わされる．また，グラフは，使用者の好みに合わせてノードの色や大きさを変更することが可能である．著者らは，提案手法による *Code Smells* の可視化は，システムに影響をおよぼしている箇所や，問題になりそうなコード片を多く含んでいる箇所の特定に役立つと主張している．

Hotta らは，リファクタリングパターンの 1 つである *Form Template Method* リファクタリングの適用が可能な箇所を特定し，開発者に提示する手法を提案している [6]．*Form Template Method* リファクタリングとは，親クラスに処理の概要を記述し，詳細な実装は子クラスごとに行うというものである．このリファクタリングにより，*CodeSmells* の 1 つである重複コードを集約することが可能である．Hotta らは，入力として与えられたソースコードから，*Form Template Method* リファクタリングの適用が可能なメソッドのペアを自動的に特定し，提示するツールを実装している．そのツールは，*Form Template Method* リファクタリングの適用が可能であると思われる候補をす

べて提示するため，使用者は *Form Template Method* リファクタリングを適用したい候補のみを任意に選択することが可能である．また，選択した候補について，どのようにソースコードを変更すればよいかという情報を GUI で提示する．

Ratzinger らは，機械学習を用いてリファクタリング箇所を特定する手法を提案している [42]．Retzinger らの提案手法では，ソースファイルの特徴とそのソースファイルがリファクタリングされたことがあるかどうかという情報を合わせて学習し，学習モデルを構築する．そして，その学習モデルにソースファイルの情報を与えることで，そのソースファイルがリファクタリングされやすいかどうかを予測する．また，評価実験の結果，彼らの提案手法により，*Precision* が 49〜69 %，*Recall* が 29〜64 %の精度で，リファクタリングされやすいソースファイルの予測が可能であることを明らかにした．

8　まとめと今後の課題

リファクタリングは，ソフトウェア開発においてしばしば用いられる重要な技術である．リファクタリングによるソースコードの整理は，ソフトウェアの保守性を向上させることに役立つ．しかし，ソースコードは膨大であるため，開発者がリファクタリング箇所を手動で特定することは，しばしばコストのかかる作業となる．さらに，開発者が本来リファクタリングすべきであった箇所を見落とす可能性もあり，効率的ではない．したがって，リファクタリング箇所の特定に要するコストの削減や，リファクタリング箇所の見落としの防止のために，リファクタリング箇所を自動で特定し，効率的なリファクタリング作業を促す必要がある．一方で，リファクタリングを行う箇所はプロジェクトごとに異なる可能性があるため，リファクタリング箇所を特定するための一般的および厳密な基準というものは存在しないといった問題がある．そのため，もしある基準に基づいてリファクタリング箇所の特定を行った場合，ある開発者にとってはリファクタリングすべきと特定された箇所が有用であっても，他のある開発者にとってはあまり有用でない可能性がある．

この問題を解決すべく，後藤らは機械学習を用いて過去に実際に行われたリファクタリングの情報を学習し，学習した情報をもとにリファクタリング箇所を特定する手法を提案している [9]．しかし，後藤らの提案手法では，一部のプロジェクトに対しては高い精度での予測に成功しているが，すべてのプロジェクトに対して高精度で予測が可能というわけではない．

そこで本研究では，後藤らの提案手法よりもさらに高い精度でリファクタリング箇所を予測することができる学習モデルの構築を目的とし，後藤らの手法の改良を行った．後藤らの提案手法では，メソッドのサイズや複雑度，凝集度など 21 種類の特徴量を用いて，*Extract Method* リファクタリングを行うべきメソッドの推薦を行っている．しかし，後藤らが用いている特徴量の多くはメソッドの抽象的な情報であり，具体的にメソッドがどのような構文から成り立っているかまでは考慮していない．そこで本研究では，後藤らよりもメソッドの構造を詳細に表すことのできる構文情報に着目し，それらを特徴量として利用した．さらに，訓練データの取得方法についても，開発者によってリファクタリングすべきかどうか検討されている可能性の高いメソッドから選択するなどの改良を加えた．

また，提案手法の評価を行うため，5 つのオープンソースプロジェクトを用いて実験を行った．実験の結果，提案手法により，*Precision* が 86〜93 %，*Recall* が 79〜97 %と高い精度でリファクタリング箇所の特定に成功した．また，多くの場合において，後藤らよりも高い精度でリファクタリング箇所を予測できていることを確認した．

今後の課題は以下の通りである．

- より多くのソフトウェアに対して実験を行う
- データマイニングツール Weka の設定を変更する（オプションの追加など）
- Weka 以外のデータマイニングツールを使う

- Kenja によって検出することのできなかった *Extract Method* リファクタリングを含めた，過去に行われたすべての *Extract Method* リファクタリングを用いる
- データセットに含まれる *refactored methods* と *non-refactored methods* の割合を変えてモデルの構築および評価を行う
- 本研究において特徴量として用いた状態ベクトルが，実際にプロジェクトの特色を反映しているかどうか調査を行う
- *Extract Method* リファクタリング以外のリファクタリングについても，リファクタリング箇所を特定できるように手法を拡張する

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました 楠本真二教授に心から感謝申し上げます.

本研究に関して，有益かつ的確なご助言を頂きました岡野浩三准教授に深く感謝申し上げます.

本研究に多大なるご助言およびご指導を頂きました井垣宏特任准教授に深く感謝致します.

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました肥後芳樹助教に深く感謝申し上げます.

本研究を行うにあたり，多大なるご助言，ご助力を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士後期課程 3 年の堀田圭佑氏に深く感謝申し上げます.

本研究に関して，多大なるご助言をいただきました，奈良先端科学技術大学院情報科学研究科ソフトウェア設計学講座の藤原賢二氏に深く感謝申し上げます.

本論文の作成にあたり，多くの場面でお力添え頂きました，新日鉄住金ソリューションズの後藤祥氏に深く感謝申し上げます.

その他，楠本研究室の皆様のご助言，ご協力に心より感謝致します.

最後に，本研究に至るまでに, 講義などでお世話になりましたコンピュータサイエンス専攻の諸先生方に，この場を借りて心から御礼申し上げます.

参考文献

[1] Tom Mens and Tom. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, Vol. 30, No. 2, pp. 126–139, Feb 2004.

[2] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Jun 1999.

[3] E. Murphy-Hill, C. Parnin, and A.P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 5–18, Jan 2012.

[4] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, Vol. 27, No. 1, pp. 1–12, Jan 2001.

[5] E. Van Emden and L. Moonen. Java Quality Assurance by Detecting Code Smells. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pp. 97–106, May 2002.

[6] Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Identifying, Tailoring, and Suggesting Form Template Method Refactoring Opportunities with Program Dependence Graph. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, pp. 53–62, Mar 2012.

[7] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the Relation of Refactorings and Software Defect Prediction. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pp. 35–38, May 2008.

[8] M.J. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *11th IEEE International Symposium Software Metrics*, p. 15, Sep 2005.

[9] 後藤祥, 吉田則裕, 藤原賢二, 崔恩瀞, 井上克郎. 機械学習を用いたメソッド抽出リファクタリングの推薦手法. 情報処理学会論文誌, 第 56 巻, 2014. （採録決定）.

[10] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, pp. 308–320, Dec 1976.

[11] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pp. 439–449, 1981.

[12] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476–493, Jun 1994.

[13] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of Extract Method Refactoring Opportunities. In *European Conference on Software Maintenance and Reengineering*, pp. 119–128, Mar 2009.

[14] Katsuhisa Maruyama. Automated Method-extraction Refactoring by Using Block-based Slicing. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, pp. 31–40, May 2001.

[15] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. Recommending Automated Extract Method Refactorings. In *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 146–156, Jun 2014.

[16] 藤原賢二, 吉田則裕, 飯田元. 構文情報を付加したリポジトリによるメソッド抽出リファクタリングの検出. 電子情報通信学会技術報告, 第 113 巻, pp. 19–24. 電子情報通信学会, 5 2013.

[17] Git. `http://git-scm.com`.

[18] Apache subversion. `http://subversion.apache.org/`.

[19] CVS - Open Source Version Control. `http://www.nongnu.org/cvs/`.

[20] A.Z. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences*, pp. 21–29, Jun 1997.

[21] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[22] Quinlan J. Ross. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., Jan 1993.

[23] Finn V Jensen. *An introduction to Bayesian networks*, Vol. 210. UCL press London, 1996.

[24] David W Hosmer Jr and Stanley Lemeshow. *Applied logistic regression*. John Wiley & Sons, 2004.

[25] M.A. Hall and G. Holmes. Benchmarking attribute selection techniques for discrete class data mining. *Knowledge and Data Engineering, IEEE Transactions on*, Vol. 15, No. 6, pp. 1437–1447, Nov 2003.

[26] Ron Kohavi and George H. John. Wrappers for feature subset selection. *Artificial Intelligence*, Vol. 97, No. 1–2, pp. 273–324, 1997.

[27] Mark A Hall. *Correlation-based feature selection for machine learning.* PhD thesis, The University of Waikato, 1999.

[28] Kenji Kira and Larry A Rendell. A practical approach to feature selection. In *Proceedings of the ninth international workshop on Machine learning*, pp. 249–256, 1992.

[29] Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Predicting next changes at the fine-grained level. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference*, pp. 126–133, 12 2014.

[30] Eclipse Java development tools (JDT). `http://www.eclipse.org/jdt/`.

[31] Pang-Ning Tan, Michael Steinbach, Vipin Kumar, et al. *Introduction to data mining*, Vol. 1. Pearson Addison Wesley Boston, 2006.

[32] Weka 3: Data Mining Software in Java. `http://www.cs.waikato.ac.nz/ml/weka/`.

[33] Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Enhancement of crd-based clone tracking. In *Proceedings of the 13th International Workshop on Principles of Software Evolution*, pp. 28–37, Oct 2013.

[34] JDeodorant. `http://www.jdeodorant.com/`.

[35] Deepak Alurm, D Malks, and J Crupi. Core j2ee patterns. *Sun Microsys. terns Inc*, 2001.

[36] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising Behaviour Preserving Program Transformations. In *Graph Transformation*, Vol. 2505 of *Lecture Notes in Computer Science*, pp. 286–301. Springer Berlin Heidelberg, Oct 2002.

[37] Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for Generalization Using Type Constraints. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, pp. 13–26, Oct 2003.

[38] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings International Conference On Software Maintenance*, pp. 576–585, Oct 2002.

[39] Ladan Tahvildari and Kostas Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *Proceedings 7th European Conference on Software Maintenance and Reengineering*, pp. 183–192, Mar 2003.

[40] Paolo Bottoni, Francesco Parisi-Presicce, and Gabriele Taentzer. Coordinated distributed diagram transformation for software evolution. *Electronic Notes in Theoretical Computer Science*, Vol. 72, No. 4, pp. 59–70, Feb 2003.

[41] Vaclav Rajlich. A model for change propagation based on graph rewriting. In *Proceedings International Conference on Software Maintenance*, pp. 84–91, Oct 1997.

[42] Jacek Ratzinger, Thomas Sigmund, Peter Vorburger, and Harald Gall. Mining Software Evolution to Predict Refactoring. In *International Symposium on Empirical Software Engineering and Measurement*, pp. 354–363, Sep 2007.