# 相関ルールを利用したソースコードの識別子推薦手法

阿部 真之[1,a)]　寺田 実[1,b)]

**概要**：大規模なプログラムの開発を行うプロジェクトでは, 既存のコードに手を加える形でコーディングを行う場面が多くなる. 更に, このようなプロジェクトでは複数人で開発を行う事が多く, コードの可読性やプログラミング効率を向上させるために, 変数名や関数名といった識別子には適切な名前付けを行うことが重要となる. 本研究では, 識別子名には互いに関連のある名前を持つものが存在し, それらは同じメソッド内で使われる事が多いという特徴に着目し, 機能追加や編集を行いたいプロジェクトのソースコード全体から識別子の共起関係を解析し, それを基に, 編集中のソースコードのメソッド内で識別子を自動的に推薦するシステムを作成した. また, 推薦に利用する評価指標をいくつか用意し, それぞれに対して評価を行った. 更に, 実験により, Java のプログラムに対して実際に識別子を約 56% の確率で適切に推薦できることを確認した。

**キーワード**：ソースコード推薦, 相関ルール, データマイニング

## 1. はじめに

大規模なプログラムの開発を行うプロジェクトでは, 既存のコードに手を加える形でコーディングを行う場面が多くなる. 更に, このようなプロジェクトでは複数人で開発を行う事が多い. ソフトウェアの保守や作成では, プログラムを理解するために, 識別子名からその関数や変数の役割を推測する. 例えば, getWidth という関数名は何らかの幅を取得する, sourceFile という変数名は送り元のファイル名を表している, というように識別子名からある程度何をしているのかを推測することが出来る. この時, 識別子の名前付けが正しい役割を表現していなければ, プログラムの理解により多くの時間がかかってしまう. コードの可読性やプログラミング効率を向上させるために, 変数名や関数名といった識別子には適切な名前付けを行うことが重要となる.

一方, 識別子の中には互いに関連のある名前を持つものが存在する. そして, これらの識別子は同じ関数内で使われる事が多い. 例えば, getWidth と getHeight や, sourceFile と destFile といった識別子名は一緒に使われることが多い. そのため, 関連のある識別子の内, 一方を入力すると, もう一方の識別子を自動的に推薦するようにすると, 開発者が識別子に対して適切な名前付けを行うための支援ができると考えられる. また, API の識別子名に対しても同様の共起関係が成り立つと考えられる. 開発者は多様な API から必要な機能を持つものを選択し, どのように使うかをドキュメントやソー

[1] 電気通信大学大学院
情報理工学研究科 情報・通信工学専攻
[a)] a1531003@edu.cc.uec.ac.jp
[b)] terada.minoru@uec.ac.jp

スコードから調査する必要がある．これらの手間を軽減するためのコード補完機能としての利用も期待できると考えられる．

そこで本研究では, 機能追加や編集を行いたいプロジェクトのソースコード全体から識別子の共起関係を解析し, それを基に, 編集中のソースコードのメソッド内で識別子を自動的に推薦するシステムを提案する．提案手法では, 既存のソースコードでは一定の命名規則に従った適切な識別子名が使われていることを前提とし, 既存の識別子の中から, これからメソッド内で使用する可能性のある識別子を候補として推薦する．提案手法を用いることにより, プロジェクト内独自の命名法によって命名された識別子も推薦が可能となると考えられる．これにより, 開発者が既存プロジェクトに機能追加, バグ修正, リファクタリングを行う際の識別子の名前付けを支援する．

本研究では, 相関ルールマイニング [1] を用いて識別子同士の一対一の共起関係を抽出し, コード編集中に識別子を推薦するシステムを作成し, 適切な候補を推薦できたか評価を行った．システムの動作例を以下に示す．図 1 のようなコードを記述している時に, for 文の 1 行目を記述し終えた所で

```
static String toHTML(String str) {
  StringBuffer buf = new StringBuffer();

  for(int i=0; i < str.length(); i++) {
    char ch;
    switch(ch=str.charAt(i)) {
    case '<': buf.append("&lt;"); break;
    case '>': buf.append("&gt;"); break;
    case '\n': buf.append("\\n"); break;
    case '\r': buf.append("\\r"); break;
    default:  buf.append(ch);
    }
  }

  return buf.toString();
}
```

**図 1** 破線枠内をこれから記述するために, 下線部分の識別子を使って推薦を行う．

**表 1** 動作例: 候補の提示．上位候補ほど関連度が高い．

| 識別子名 | 型名 |
|---|---|
| append | StringBuffer |
| toString | String |
| charAt | char |
| substring | String |
| size | int |

本システムが動作すると, 識別子 $str, buf, i, length$ を入力として取得し, 表 1 のような候補が提示される．開発者は提示された候補を参考にしながら, 残りの部分を記述することができる．

本研究では更に, 相関ルールマイニングにより得られた関連度の数値に対して他の指標で重み付けすることによって, 上位に現れる推薦候補を増やすことを試みた．1 つ目の指標は型名である．同じ型名を持つ 2 つの識別子は, 共起している可能性が高いと考え, 推薦候補の持つ型名と, 候補の推薦に使用した入力元となる識別子 (以下, 推薦元と呼ぶ) の持つ型名が一致した場合に, 関連度に対して重み付けを行った．2 つ目の指標は重要語である．関数名の動詞部分や変数名の名詞部分を重要語として取り出し, この部分のみを用いて関連度を計算し, 共起していた場合に重み付けを行った．そして, これらの指標がそれぞれ有効であるかの評価を行った．更に, 識別子同士の共起関係を多対一に拡張した場合の評価を行った．

## 2. 相関ルール

### 2.1 相関ルールとは

相関ルール [1] は Agrawal らが提案したデータマイニングの手法である．

相関ルールとは, ある事象 A が発生したときに, 別の事象 B も発生するといった関連を示すものであり, A ⇒ B と表される．ここで, ルールの A の部分を条件部, B の部分を結論部と呼ぶ．

もともとはマーケットで販売される商品間の関連性を分析するために提案された手法であり, 大量のトランザクションを含むデータベースから意味のある関連性を抽出することが出来る．

相関ルールを抽出するための評価指標として, 支持度, 確信度, リフト値がある．支持度とは, ルー

ルそのものの出現率であり, 条件部と結論部を共に含むトランザクションが全体に占める割合である. 確信度とは, 条件部と結論部の関連の強さであり, 条件部が発生したときに結論部が起こる割合である. 例えば, 「おにぎりとお茶を同時に購入した人は全顧客のうち 10%だった」「おにぎりを購入した人のうち 70%は, お茶も一緒に購入した」というデータが得られた場合, 相関ルールは { おにぎり }⇒{ お茶 } と表せ, 支持度は 10%, 確信度は 70%となる.

アイテム集合を $X$,$Y$, 支持度 $sup$ とすると, 確信度 $conf$ は以下のように表される.

$$conf(X \Rightarrow Y) = \frac{sup(X \cup Y)}{sup(X)}$$

しかし, 確信度が高くても, 関連性が強いとは言えないパターンも存在する. 例えば, 上の例でほとんどの顧客がおにぎりを購入したかどうかにかかわらずお茶を購入していたとすると, おにぎりとお茶の関係に意味があるとは言えなくなる. そこで, リフト値と呼ばれる指標を用いて興味深いルールのみを選択する事が考えられた. リフト値 $lift$ は以下のように表される.

$$lift(X \Rightarrow Y) = \frac{conf(X \Rightarrow Y)}{sup(Y)}$$

リフト値が低い場合は, データベース内での $Y$ の出現率が高く, $X$ との関連があるとは言えないと考えられる.

### 2.2 アプリオリアルゴリズム

最少支持度 $minsup$ と最小確信度 $minconf$ を設定し, 支持度と確信度がそれ以上となる相関ルールを全て抽出することで, その中から興味深いルールを探すことが可能になる.

しかし, 集合内のアイテム数が増えるとルールの数が膨大になる. 例えば, アイテム集合 $\{a, b, c, d, e\}$ があるとする. このアイテム集合に含まれるルールの例を挙げると, $\{a, b\} \Rightarrow \{d\}$ や $\{b, c\} \Rightarrow \{a, d, e\}$ のようなルールも作ることができる. つまり, 要素の一部のみを使ったルールや, 条件部や結論部が複数のアイテムからなるルールも作成が可能ということである. アイテムの数を $n$ とするとルールの数は $\sum_{i=2}^{n} {}_n\mathrm{C}_i(2^i - 2)$ 通りになり, $n = 10$ の場合でも 57002 通りとなる. よって, データベース内のアイテムを使って組み合わせ可能な相関ルールを全て計算することは困難である. この課題を解決したのがアプリオリアルゴリズムである [1].

アイテム集合を $X$, $Y$ とすると, 支持度 $sup$ について次の式が成り立つ.

$$sup(X) \geq sup(X \cup Y)$$

よって, $minsup > sup(X)$ ならば, $minsup > sup(X \cup Y)$ となるため, $sup(X)$ の支持度が $minsup$ 未満であれば $sup(X \cup Y)$ の支持度は計算する必要がない.

アプリオリアルゴリズムでは, $sup(X) \geq minsup$ を満たすアイテム集合の全体の集合を頻出アイテム集合と呼び, 最初のステップで頻出アイテム集合 $L$ を求める. アイテム数 $k$ のアイテム集合で構成される頻出アイテム集合を $L_k$ とすると, $L$ を求めるアルゴリズムは以下のようになる.

(1) $k = 1$ とする.

(2) アイテム数 $k$ のアイテム集合を候補集合 $C_k$ とし, これらの集合の支持度を計算する. 支持度が $minsup$ 以上となる要素を $L_k$ の要素とする.

(3) $L_k$ の要素を組み合わせて新たに $C_{k+1}$ を作る. ここで, $C_{k+1}$ の要素となるアイテム集合は, 部分集合が $L_k$ に $k+1$ 個含まれているもののみである.

(4) $C_k$ が空なら $L = \bigcup_{i=1}^{k} L_i$ となり, そうでないなら $k = k + 1$ として (2) へ戻る.

次のステップでは, $L$ の要素であるアイテム集合から, 確信度が $minconf$ 以上となる相関ルールを全て抽出する. 例えば, $L$ の要素にアイテム集合 $\{a, b, c\}$ が含まれていたとすると, $\{a, b\} \Rightarrow \{c\}, \{a\} \Rightarrow \{b, c\}, \{a, c\} \Rightarrow \{b\} \ldots$ のようにアイテム集合内の要素でルールを作り, 各ルールの確信度が $minconf$ 以上であるかを調べる.

アプリオリアルゴリズムは, 頻出アイテム集合の部分集合は頻出アイテム集合であるという性質を利用して枝狩りを行うことで, 支持度を計算する必要のあるアイテム集合の数を大幅に削減している.

## 3. 提案手法

本節では, 編集中のソースコードのメソッド内で識別子を推薦する手法について述べる. 提案手法では, 識別子名には互いに関連のある名前を持つものが存在し, それらは同じメソッド内で使われる事が多いという特徴に着目する. これにより, 関連のある識別子の内, どちらか一方が入力されると, もう一方の識別子を推薦するようにすることで, プログラミングの際の識別子の名前付けを支援できると考えられる.

### 3.1 識別子同士の共起関係の解析

識別子の共起関係の解析には 2 節で述べた相関ルールを基にした指標を用いる. 共起関係を解析するためのソースコードコーパスは, 開発者が機能追加や編集を行いたいプロジェクトの全ソースコードとする. これらのソースコードの構文解析を行い, 1 つのメソッドの中に出現するフィールド変数, ローカル変数, メソッド呼び出しの識別子を取得し, これを 1 つのトランザクションデータとする. 1 つのメソッドの中に複数の同じ識別子が出現した場合は, 最初に出現した 1 つのみを登録する. 構文解析により得られたトランザクションデータはデータベースに保存し, 相関ルールマイニングに利用する.

提案手法では, 関連度に対して, 追加の指標による重み付けを行う事で, 上位に現れる推薦候補が増えるかを検証した. また, 重み付け推薦では, 相関ルールは条件部, 結論部共にアイテム数 1 の一対一の対応関係のみのルールを用いる.

### 3.2 型名による重み付け

識別子が変数名の場合はその変数の型, メソッド名の場合は戻り値の型を識別子の持つ型名とし, 同じ型名を持つ識別子は, 互いに関連している可能性が高いと考えた.

そこで, 推薦候補の識別子が持つ型名と, 推薦元の識別子が持つ型名が一致した場合に, 関連度に対して重みづけを行う. 推薦対象となる識別子 $X, Y$ の型を $X_{type}, Y_{type}$ とすると, 評価値 $score$ は重み係数 $k_{type}$ を用いて次のように表される.

$$score = \begin{cases} (1 + k_{type})conf & (X_{type} = Y_{type}) \\ conf & (X_{type} \neq Y_{type}) \end{cases} \quad (1)$$

また, 事前調査として, Java で書かれたオープンソースプロジェクトである Apache Ant[*1]に対して実際に重み無しでの提案手法による推薦を行い, 推薦元と推薦候補の識別子の型名を比較した. その結果, 約 15% の推薦候補が推薦元と同じ型名を持っていることが分かった.

### 3.3 重要語による重み付け

一般的な命名規則では, メソッド名は動詞で始まる事が多い. また, 同様に変数名は名詞で終わる事が多い. そこで, メソッド名の動詞部分, 変数名の名詞部分をそれぞれ重要語として取り出し, 重要語同士の関連度を計算した. この関連度を用いて重みづけを行う. 評価値は次のように表される.

$$score = conf + k_{keyword} \cdot conf_{keyword} \quad (2)$$

ここで, $k_{keyword}$ は重み係数であり, $conf_{keyword}$ は推薦対象となる識別子に含まれる重要語同士の関連度である.

重要語のアイテム集合は識別子のアイテム集合とは別の集合として扱う. 例えば 2 つの識別子 lockXxx, unlockYyy があるとき, これらの重要語はそれぞれ lock, unlock である. この識別子間の評価値は, 識別子のコーパスを用いて計算した {lockXxx } ⇒ { unlockYyy} の関連度に, 重要語のコーパスを用いて計算した {lock } ⇒ { unlock} の関連度に重み付けした値を足して求める.

重要語の抽出は, 次のような手順で行う. まず, camelCase や snake_case で連結された識別子名を単語ごとに分解し, OpenNLP[*2]を利用して品詞解析を行う. 品詞解析の結果から, メソッド名の動詞部分や変数名の名詞部分を取り出すことが出来れば, それを重要語とする. 重要語が見つからなければ, メソッド名は先頭の単語, 変数名は末尾の単語に対して WordNet[*3]を利用して動詞, 名詞に相当

[*1] http://ant.apache.org/
[*2] https://opennlp.apache.org/
[*3] https://wordnet.princeton.edu/

する単語であるかを調べ, 該当する単語が存在すればそれを重要語とする.

### 3.4 多対一のルールによる推薦

これまでの指標では, 一対一の対応関係のルールのみを扱っていた. そこで, 重み付けを行わずに, 多対一の対応関係も含めたルールを扱った場合についても評価を行う.

例えば, 編集中のソースコードで使用している識別子が $\{a, b, c, d\}$ の 4 つだった場合は, 以下の手順によって推薦を行う.

(1) データベースに登録されたソースコードコーパスから, アプリオリアルゴリズムを用いて $\{a, b, c, d\}$ で構成された頻出アイテム集合 $L$ を抽出する. ここで, $L = \{\{a\}, \{b\}, \{c\}, \{d\}, \{a, c\}, \{b, c\}\}$ が得られたとする.

(2) $L$ から他要素の部分集合となる要素を取り除き, $L'$ とする. $L' = \{\{d\}, \{a, c\}, \{b, c\}\}$ となる.

(3) $L'$ の各要素を条件部とする相関ルールをデータベースから抽出する.

## 4. 推薦システム

提案手法を実装した推薦システムを Eclipse のプラグインとして作成した. 推薦対象とするプログラミング言語は Java である.

本システムは, ソースコードを解析する事前処理と, 編集内容に基づいた推薦を行う本処理からなる.

### 4.1 ソースコードの解析

編集対象となるプロジェクトのソースコード全体を構文解析し, 各メソッド内で使用されているフィールド変数, ローカル変数, メソッド呼び出しを抽出し, それらの識別子名, 型名, 重要語をデータベースへ格納する. ソースコードの構文解析には, Eclipse JDT の ASTParser を使用した.

この処理は事前処理として 1 回のみ実行する.

### 4.2 推薦元の取得

現在開発者が編集しているソースコードに対して構文解析を行い, 更にエディタ内のカーソルの位置を取得する. カーソルがメソッドの内側に存在した場合, そのメソッド内にあるフィールド変数, ローカル変数, メソッド呼び出しの識別子を取得する. 取得した識別子を相関ルールマイニングの入力データとして与え, データベースを検索して提案手法の評価値による関連度の評価を行い, 共起関係のある識別子を抽出する.

編集中の識別子を取得する処理は, 開発者がソースコード内でセミコロンを入力し, 構文上正しいコードになっていた場合と, カーソルが別のメソッド内に移動した場合に自動的に実行される. 編集中のソースコードの構文解析に失敗した場合は, 失敗した 1 文に含まれる識別子は取得せず, 構文解析に成功した部分に含まれる識別子のみを入力として推薦を行う.

### 4.3 推薦候補の提示

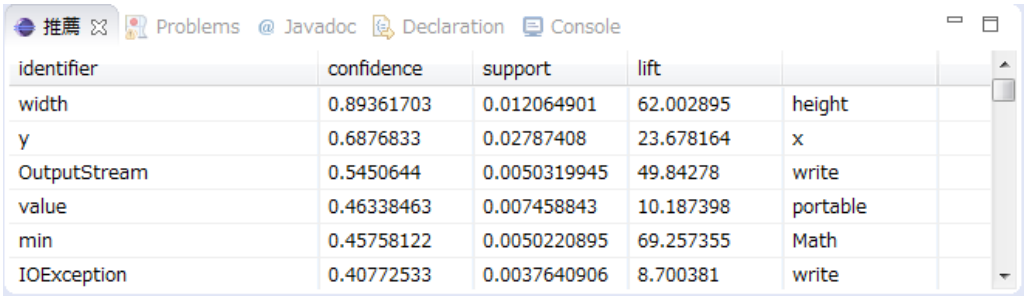
相関ルールマイニングによって, 取得した識別子と共起関係のある識別子をデータベースから抽出し, 推薦候補として関連度の高い順に提示する. 関連度として利用する指標には 3 節で示したように複数存在するが, これらは設定画面により設定することでどの指標による評価を行うかを選択することが出来る.

推薦候補は図 2 のように Eclipse のビューで表示される. 表示内容の更新は 4.2 節の処理により新たな推薦元が取得される度に実行される.

このとき, 支持度やリフト値に閾値を設定し, 設定以下の数値となった識別子を候補から取り除くことができる. これらの閾値を設定する事により, ソースコードコーパス中でほとんど登場しない識別子や, 頻繁に登場するありふれたルールを推薦候補から除外して提示することができる.

## 5. 評価実験

提案手法がメソッド内に登場する識別子名を適切に推薦できているかを評価するために, 実験を行った. また, 提案手法の関連度に基づく評価指標




| identifier | confidence | support | lift | |
|---|---|---|---|---|
| width | 0.89361703 | 0.012064901 | 62.002895 | height |
| y | 0.6876833 | 0.02787408 | 23.678164 | x |
| OutputStream | 0.5450644 | 0.0050319945 | 49.84278 | write |
| value | 0.46338463 | 0.007458843 | 10.187398 | portable |
| min | 0.45758122 | 0.0050220895 | 69.257355 | Math |
| IOException | 0.40772533 | 0.0037640906 | 8.700381 | write |

**図 2** 推薦ビュー

が有効なものであったかの評価を行った.

### 5.1 実験対象

実験対象のプロジェクトとして, Apache Ant を利用した. 構文解析により 10,710 個のメソッドを抽出し, その中に含まれる識別子をデータベースに格納してコーパスとした.

### 5.2 実験方法

実験対象のソースコードコーパス内に含まれる全てのメソッドに対して, コーパスに登録された後半 5 割の部分に初登場する識別子を削除する. その後, 残った前半 5 割の部分のみから推薦を行い, 推薦候補の上位 10 個までを取得し, 推薦された候補が削除した後半部分に含まれているかを調べる. 候補が含まれていた場合, その候補の順位を記録する.

例として, あるメソッドに 4 つの識別子 $\{a, b, c, d\}$ がこの順番に登場していたとする. この場合, 後半 5 割部分に相当する識別子 $\{c, d\}$ を削除し, 残りの識別子 $\{a, b\}$ のみを使用して提案手法による推薦を行う. この推薦により提示された候補の上位 10 個の中に, 削除した $\{c, d\}$ のうち, どちらか 1 つでも含まれていれば, その識別子が推薦されていた順位を記録し, そのメソッドに対しての推薦に関しては成功したものとする.

識別子の登場順に前半と後半に分けたのは, 開発者がソースコードを書くときに, 先頭から順番に記述していくと想定したためである.

また, 実験内では関連度, 支持度, リフト値の閾値による推薦候補の除外は行わない.

### 5.3 平均逆順位 (MRR)

実験の評価尺度の 1 つとして, 平均逆順位 (MRR) を用いる. これは, 順位付けされた推薦候補の評価に用いられるもので, 候補の中で最初に正解が現れた順位の逆数の平均によって求められる. 実験対象となるメソッドの数を $N$, i 番目のメソッドでの推薦候補のなかで, 最初に正解が現れた順位を $r_i$ とすると, MRR は次のように表される.

$$\mathrm{MRR} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{r_i}$$

### 5.4 結果

各評価指標での推薦確率と MRR を表 2 に示す. 型名と重要語は, 式 (1)(2) の重み係数 $k = 0.5$ の場合の数値である. 推薦確率とは, 実験対象となったメソッドに対して, 推薦が成功した確率である. 提示された推薦候補が削除した後半部分に 1 つでも含まれていれば成功したとしている.

更に, 型名と重要語では重み係数 $k$ を 0 から 1 までの間で 0.1 ずつ変化させて実験を行った. 重み係数による推薦確率と MRR の変化をグラフでプロットした結果, 図 3, 図 4 のようになった.

**表 2** 推薦確率と MRR

| | 一対一 | 多対一 | 型名 | 重要語 |
|---|---|---|---|---|
| 推薦確率 | 0.563 | 0.554 | 0.561 | 0.501 |
| MRR | 0.356 | 0.354 | 0.341 | 0.324 |

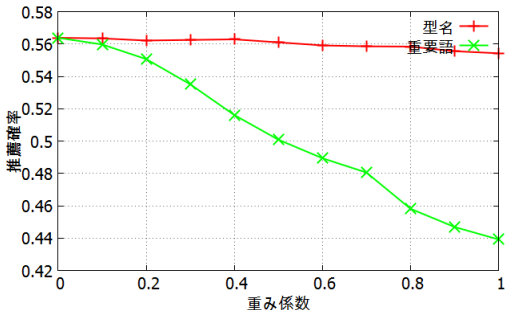



図 3　重み付けによる推薦確率の変化

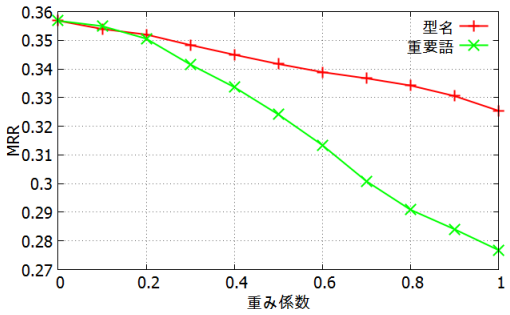



図 4　重み付けによる MRR の変化

### 5.5　考察

表 2 の結果を見ると, 提案手法で述べた 3 つの評価指標は一対一の重み無し関連度による評価との比較では, 推薦確率と MRR の増加は見られなかった. また, 図 3, 図 4 の結果から, 重み付けによる評価ではこれらの評価値が下がってしまうということが確認できる.

重要語による重み付け推薦では, 他の指標と比べても大きく評価値が減少してしまった. この原因として, 1 単語のみで構成された識別子が多く存在していたことが考えられる. 1 単語のみで構成された識別子では, 識別子名と重要語が同じ単語になってしまう. 今回の実験で使用したソースコードコーパスの中で調査した結果, 全体の約 42% が 1 単語のみからなる識別子であった. これにより, 重要語という指標がうまく機能していなかったと考えられる. また, 1 単語で構成された識別子の約 87% が変数名であった. 重要語を評価指標として扱う場合は, メソッド名の重要語のみに限定した方が効果があると考えられる.

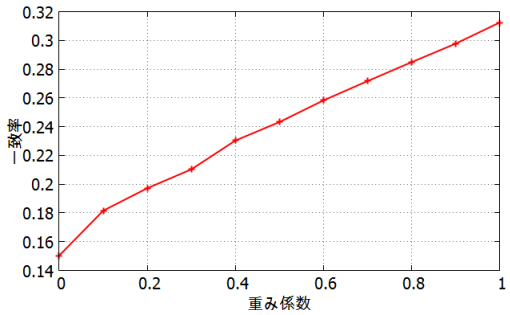



図 5　推薦元と正解候補の型名の一致率

一方, 型名による重み付け推薦では, 評価値の増加は無かったものの, 重さ係数の増加による評価値の減少も少なかった. また, 型名による重み付け推薦では, 正解の推薦候補として提示された識別子は, 推薦元の識別子と同じ型を持つものが増加していた. 重みの変化による推薦元と正解推薦候補の型の一致率を測定したグラフを図 5 に示す. この結果から, 型名による重み付け推薦では, 評価値をほとんど減少させることなく推薦候補の識別子を一部変更することができると考えられる. これにより, 開発者が, 提示して欲しい識別子の型がメソッド内に記述した識別子と同じ型を持っていることを予測できた場合に, この推薦を利用することで目的の識別子を見つけることができる可能性があると考えられる.

多対一のルールによる推薦は, 一対一のルールによる推薦の結果とほとんど変わらなかった. 多対一のルールを見つけ出す処理は, 一対一のルールのみを扱うよりも多くの処理時間を必要とするため, 推薦結果がほとんど変化しないのであれば一対一のルールによる推薦を利用することで, 計算コストを削減するべきである.

また, 正解の推薦候補が提示された順位を調べ, その順位に提示された識別子数をグラフにすると図 6 のようになった. この結果を見ると, 上位に提示された推薦候補ほどよく使われていることが分かる. これにより, 提案手法による識別子の提示では推薦候補を適切に上位に並べられていることが分かる.

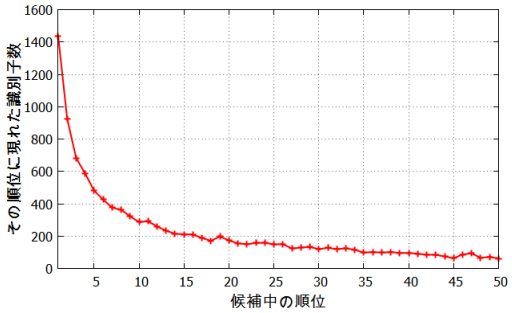



図 6　後半部分に含まれていた識別子の候補中の順位

## 6. 関連研究

Li ら [2] は, ソフトウェアのソースコードから暗黙のプログラミングルールを見つけ, そのルールを用いて違反を検出する手法を提案している. この手法を実装した PR-Miner は, Linux カーネルのソースコードから 16 個の違反を検出した. また, Livshits ら [3] は, バージョン管理システムの改版履歴を解析し, 同時に変更されたメソッド呼び出しの組み合わせを抽出することでルールを見つけ, バグ検出に応用するツール DynaMine を開発している. これらの研究では, 既存のソースコード集合から共起関係のある識別子のルールを見つけているが, それをバグ検出に利用している点で本研究と異なる.

鬼塚ら [4] は, 開発者が新規作成したいメソッド名を記述したときに, そのメソッドで使用される API を推薦することで, API の選択を支援する手法を提案している. 本研究では, 推薦にメソッド名を利用せず, メソッドの内側に含まれる識別子同士で推薦を行っているが, この研究では, 作成するメソッド名に対して相関ルールを適用し, メソッド名からそのメソッドが内側でどのような識別子名を用いるのかを推薦する.

山本ら [5] は, 既存の大規模なソースコード集合から, 再利用可能なソースコードの候補を推薦する手法を提案している. この研究では, 既存のソフトウェアのソースコードからソースコードコーパスを作成し, 書きかけのソースコード片を入力として, そのソースコード片に対応した残りのソースコード片を頻度の多い順に推薦する.

## 7. まとめ

本研究ではソースコードから識別子間の共起関係を解析し, 編集中のメソッド内に存在する識別子と関連のある識別子を推薦する手法を提案した. 本手法を用いることにより, 途中まで記述されたメソッドから約 56% の確率でそれ以降メソッド内で使用する識別子名を推薦することができることを示した. また, 本手法による推薦によって上位に提示された識別子ほどよく使われていることを示し, 適切な候補を上位に提示できていることを確認した. 一方で, 推薦候補の評価値としていくつかの指標を提案し, これらの評価指標が有効であるかの検証を行ったが, 評価値の向上は見られなかった.

今後の課題としては, 評価値の向上する指標の考案が挙げられる. また, 完成したメソッドに対して相関ルールの適用を行い, 現在の識別子よりも良い名前の識別子があればそれを推薦する, というようなリファクタリング支援への応用ができるかを考えたい.

### 参考文献

[1] Rakesh Agrawal, Ramakrishnan Srikant. “Fast Algorithms for Mining Association Rules” Proceedings of the 20th VLDB Conference (1994).

[2] Zhenmin Li, Yuanyuan Zhou. “PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code” ESEC-FSE (2005).

[3] Benjamin Livshits, Thomas Zimmermann. “DynaMine: Finding Common Error Patterns by Mining Software Revision Histories” ESEC-FSE (2005).

[4] 鬼塚 勇弥, 早瀬 康裕, 山本 哲男, 石尾 隆, 井上 克郎. “メソッド周辺の識別子名とメソッド本体の API 利用実績に基づいた API 集合推薦手法の提案と評価” 情報処理学会研究報告, 2014-SE-183(16) (2014).

[5] 山本 哲男, 吉田 則裕, 肥後 芳樹. “ソースコードコーパスを利用したシームレスなソースコード再利用手法” 情報処理学会論文誌, 53(2), 644-652 (2012).