平成２８年度修士論文

相関ルールを利用した
ソースコードの識別子推薦手法

情報・通信工学専攻 コンピュータサイエンスコース

１５３１００３　阿部　真之

指導教員 寺田 実 准教授
指導教員 岩崎 英哉 教授

提出日 ２０１７年 1月３０日

概要

目的

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ことが多く, 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, ひとつのプロジェクト内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ことが多いという特徴に着目し, 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する. これにより, 利用者が既存プロジェクトに機能追加などを行う際の, コーディング時の識別子の名前付けを支援する.

方法

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, 相関ルールマイニングを利用して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, 推薦を改善するための推薦スコアをいくつか用意し, それぞれに対して評価を行った.

結論

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 58% 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評価実験により示した. また, 推薦候補の評価値としていくつかの指標を提案し, これらの推薦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. 一方で, 本手法による推薦は頻度順推薦よりも優れていることを実験により示した. 更に, 本手法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.

目次

第 1 章　序論 7
1.1 背景 7
1.2 着目点 7
1.3 目的 7
1.4 本論文の構成 9

第 2 章　関連研究 10
2.1 PR-Miner[2] 10
2.1.1 概要 10
2.1.2 本研究との比較 11
2.2 DynaMine[3] 11
2.2.1 概要 11
2.2.2 本研究との比較 11
2.3 鬼塚らの研究 [4] 12
2.3.1 概要 12
2.3.2 本研究との比較 12
2.4 山本らの研究 [5] 13
2.4.1 概要 13
2.4.2 本研究との比較 13
2.5 Eclipse Code Recommenders 14
2.5.1 概要 14
2.5.2 本研究との比較 15

第 3 章　相関ルールマイニング 16
3.1 相関ルールとは 16
3.1.1 トランザクションデータ 16
3.1.2 支持度 17
3.1.3 確信度 18
3.1.4 リフト値 19
3.2 アプリオリアルゴリズム 19
3.2.1 概要 19
3.2.2 頻出アイテム集合の抽出 19
3.2.3 相関ルールの抽出 20

第 4 章　提案手法 **21**
4.1 識別子同士の共起関係の解析 21
4.2 識別子の推薦 21
4.2.1 例 21
4.3 推薦精度の向上 23
4.3.1 型名による重み付け 23
4.3.2 重要語による重み付け 24
4.3.3 多対一のルールによる推薦 25

第 5 章　推薦システム **26**
5.1 概要 26
5.2 内部処理 27
5.2.1 ソースコードの解析 27
5.2.2 推薦元の取得 27
5.2.3 推薦候補の提示 27
5.3 設定画面 28
5.4 ステータスバーの表示 29

第 6 章　実装 **31**
6.1 開発環境 31
6.2 ソースコード解析 31
6.2.1 データベースへの登録 31
6.2.2 重要語の抽出 31
6.3 推薦候補の提示 32
6.3.1 編集中のカーソル位置取得 32
6.3.2 候補リスト 32
6.4 スレッド処理 33

第 7 章　評価実験 **34**
7.1 実験対象 34
7.2 実験方法 34
7.3 頻度順推薦 34
7.4 実験方法の例 34
7.5 平均逆順位 (MRR) 36

第 8 章　結果と考察 **37**
8.1 頻度順推薦との比較 37
8.2 提示候補は適切であったか 37
8.3 各評価指標の比較 38
8.4 合計確信度推薦と最大確信度推薦 39
8.5 多対一のルールによる推薦 39
8.6 重み係数による変化 40
8.7 重要語による重み付け 41
8.8 型名による重み付け 41

第 9 章　結論 **43**
9.1 まとめ 43
9.2 今後の課題 43
9.2.1 推薦指標の改善 43
9.2.2 リファクタリング支援への応用 43
9.2.3 文脈の考慮 44

謝辞 **45**

参考文献 **46**

付録 **47**

図目次

1.1 動作例 8

2.1 PR-Miner によるバグ検出の例 10
2.2 DynaMine のシステム設計 11
2.3 候補リストから挿入したい雛形を選択 12
2.4 雛形がコメント状態で挿入される 12
2.5 山本らの研究 13
2.6 標準的な補完機能 (アルファベット順) 14
2.7 Code Recommenders による補完 14

3.1 支持度 18
3.2 確信度 18

5.1 Eclipse 画面 26
5.2 推薦ビュー 27
5.3 設定画面 28
5.4 推薦中 29
5.5 メソッド名の表示 30

8.1 後半部分に含まれていた識別子の候補中の順位 37
8.2 各評価指標での推薦成功率 38
8.3 各評価指標での MRR 39
8.4 重み付けによる推薦成功率の変化 40
8.5 重み付けによる MRR の変化 40
8.6 推薦元と正解候補の型名の一致率 41

表目次

1.1 動作例 8

3.1 トランザクションデータの例 16
3.2 ソースコードにトランザクションデータを当てはめた例 17

4.1 コード 4.1 から抽出された識別子毎のルール 22
4.2 合計確信度推薦による評価 23
4.3 最大確信度推薦による評価 23
4.4 型名による重み付けを行ったルール 24

7.1 { `string`, `end`, `indexOf`, `sb` } から推薦された候補の例 35

8.1 頻度順推薦と提案手法の比較 37
8.2 推薦成功率と MRR 38

10.1 コード 10.1 合計確信度推薦 47
10.2 コード 10.1 最大確信度推薦 47
10.3 コード 10.3 合計確信度推薦 48
10.4 コード 10.3 最大確信度推薦 48
10.5 コード 10.5 合計確信度推薦 48
10.6 コード 10.5 最大確信度推薦 48
10.7 コード 10.7 合計確信度推薦 49
10.8 コード 10.7 最大確信度推薦 49
10.9 コード 10.9 合計確信度推薦 49
10.10 コード 10.9 最大確信度推薦 49

第1章　序論

1.1　背景

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ことが多い. 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.

しかし, 識別子の命名規則は, プログラミング言語や所属する開発組織, プロジェクトごとに存在し, 開発者はその中のみで使用される独自の命名規則に従う必要がある. また, 命名規則が常に文書化されているとは限らない. そのような場合は既にプロジェクト内に存在するソースコードを読んで, どのような命名規則が用いられているかを開発者自身で学習する必要がある.

1.2　着目点

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ことが多い. 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.

1.3　目的

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

```
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 1.1: 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.

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 1.1 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 1.1 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確信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, 確信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確信度を計算し, 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.

表 1.1: 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

1.4　本論文の構成

論文の構成を簡単に説明する．本章では, 序論として研究の背景と着目点, 目的について述べた. 第 2 章では, 本研究に関連する研究について述べる. 第 3 章では, 本研究の提案手法で利用する相関ルールマイニングについて述べる. 第 4 章では, 本研究の提案する推薦手法について述べる. 第 5 章では, 本研究の提案するシステムの概要について述べる. 第 6 章では, 本研究で提案するシステムの実装について述べる. 第 7 章では, 評価実験について述べる. 第 8 章では, 実験の結果と考察について述べる. 第 9 章では, 結論と今後の課題について述べる.

第2章　関連研究

2.1　PR-Miner[2]

2.1.1　概要

Li らは, 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バグを検出する手法を提案し, この手法を実装した PR-Miner を開発している.

```
linux-2.6.11/drivers/scsi/3w-9xxx.c:
1964 int __devinit twa_probe(struct pci_dev *pdev, ...)
1965 {
1966    struct Scsi_Host *host = NULL;
        ......
1985    host = scsi_host_alloc(...);
        ......
2036    scsi_add_host(host, &pdev->dev);
        ......
2069    scsi_scan_host(host);
        ......
2088 }
```

(a) Programming rule in twa_probe

```
linux-2.6.11/drivers/ieee1394/sbp2.c:
688 struct scsi_id_instance_data *sbp2_alloc_device
        (struct unit_directory *ud)
689 {
        ......
692     struct scsi_id_instance_data *scsi_id = NULL;
        ......
745     scsi_host = scsi_host_alloc(...);
        ......
753     if (!scsi_add_host(scsi_host, &ud->device)) {
        ......
        // scsi_scan_host(scsi_host) is missing!
764 }
```

(b) Violation in sbp2_alloc_device

図 2.1: PR-Miner によるバグ検出の例 ([2] より引用)

図 2.1 は PR-Miner によって検出されたバグの例であり, この例では, `scsi_scan_host` という識別子名は本来 (a) のように `scsi_host_alloc` と `scsi_add_host` という識別子名と一緒に出現するはずだが, (b) では `scsi_host_alloc` と `scsi_add_host` が出現しているにも関わらず, `scsi_scan_host` が出現していない.

PR-Miner は暗黙のプログラミングルールを相関ルールマイニングによって見つける. C 言語のソースコードに対応し, 関数を 1 つのアイテム集合として, アイテム集合の部分集合がソースコー

ド全体にいくつ含まれているかを支持度で表す. FP-close アルゴリズムを用いてアイテム集合を効率的に発見し, アイテム集合から生成される相関ルールに対して, 支持度と確信度に閾値を設定してバグの検出を行う. 例えば, {a，b，d} の支持度が 100 で, {a，b} の支持度が 101 なら, {a，b} が出てきているにもかかわらず {d} が出てこないケースが 1 つだけあることが分かるため, これをバグとして検出する. 検出したバグは確信度の高い順に並べられ, 開発者に知らされる.

PR-Miner を Linux カーネルのソースコードに対して適用してバグを検出したところ, 60 個のバグのうち, 16 個が最新のバージョンで修正されていることが確認できた.

2.1.2　本研究との比較

既に存在する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るという点が共通しているが, 発見したルールをバグの検出に利用するところが本研究と異なっている.

2.2　DynaMine[3]

2.2.1　概要

Livshits らは, 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.

DynaMine は, ソフトウェアのエラーを検出する目的で改版履歴情報と動的解析を組み合わせた最初のツールである. DynaMine は Eclipse プラグインを通じて Eclipse のビュー画面で検出されたパターンを利用者に提示する.

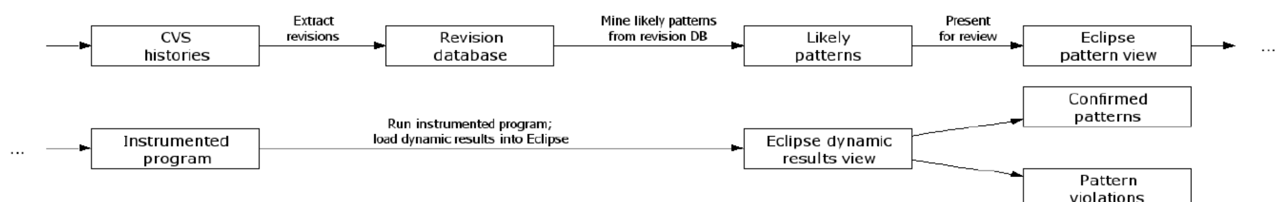


図 2.2: DynaMine のシステム設計. 上段が改版履歴の解析, 下段が動的解析 ([3] より引用)

DynaMine を利用して Eclipse と jEdit の改版履歴を解析した結果, それまで知られていなかった未知のルールが 56 個発見され, 合計で 263 個のバグを検出することが出来た.

2.2.2　本研究との比較

ソースコードの情報からプログラミングのルールを見つけるという点が共通しているが, Livshits らの研究では, ルールを発見するためにバージョン管理システムの改版履歴を利用している点が本研究と異なる. また, この研究でも 2.1 節の PR-Miner と同様に発見したルールをバグ検出に利用している.

2.3　鬼塚らの研究 [4]

2.3.1　概要

鬼塚らは, 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. この研究では, 過去の API 利用実績は, メソッド名とメソッド本体に強い関連があることに着目し, 既存のソースコードの識別子に対して相関ルールマイニングを適用し, 作成したメソッドのメソッド名からそのメソッドが内側でどのような API を用いるのかを推薦する.

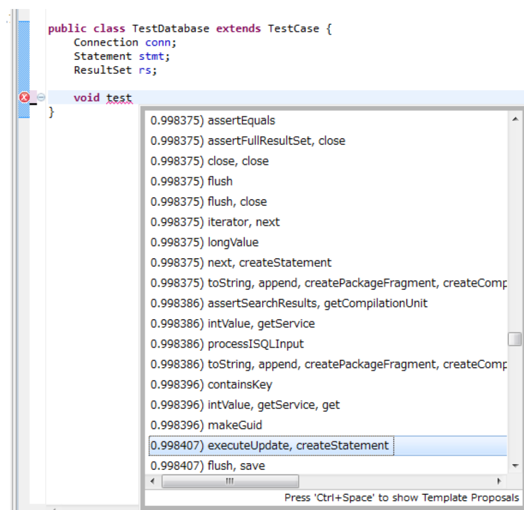


図 2.3: 候補リストから挿入したい雛形を選択 ([4] より引用)

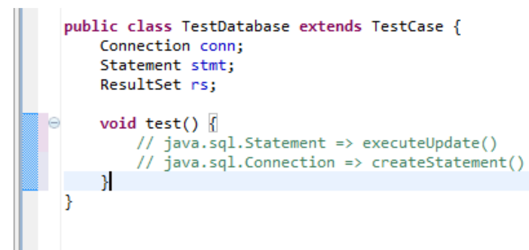


図 2.4: 雛形がコメント状態で挿入される ([4] より引用)

鬼塚らはこの手法を実装した Eclipse プラグインを開発している. このシステムの動作例を図 2.3, 図 2.4 に示す. 開発者が新規作成したいメソッド名を入力した後, Ctrl + Space キーを入力すると, 図 2.3 のようにツールが起動し, 雛形の候補リストが表示される. 候補リストから挿入したい候補を選択すると, 図 2.4 のようにコメント状態の雛形がメソッド本体に挿入される. 挿入されるコメントは, 「呼び出しメソッドが定義されているクラスの完全修飾クラス名 => 呼び出しメソッド」の形式で挿入される.

2.3.2　本研究との比較

推薦に相関ルールマイニングを用いることが共通しているが, 本研究では, 推薦にメソッド名を用いず, メソッドの内側に含まれる識別子同士から推薦を行う.

2.4　山本らの研究 [5]

2.4.1　概要

山本らは, 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.

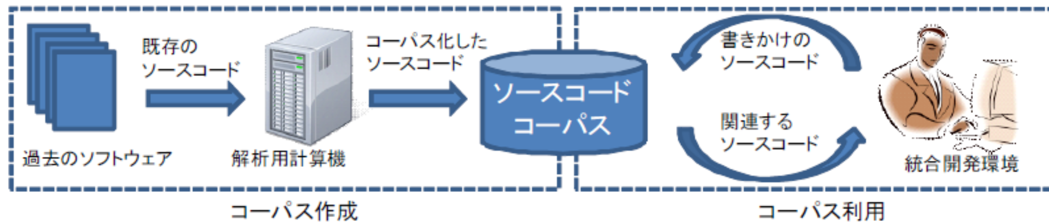


図 2.5: 山本らの研究 ([5] より引用)

この手法を以下のコード 2.1 を含むソフトウェアに適用した例を示す.

```
1  /**
2   * find a file in a directory in case unsensitive way
3   * @param parentPath where we are
4   * @param soughtPathElement what is being sought
5   * @return the first file found or null if not found
6   */
7  private String findPathElementCaseUnsensitive(String parentPath,
8      String soughtPathElement) {
9      // we are already in the right path, so the second parameter
10     // is false
11     FTPFile[] theFiles = listFiles(parentPath, false);
12     if (theFiles == null) {
13         return null;
14     }
15     for (int icounter = 0; icounter < theFiles.length; icounter++) {
16         if (theFiles[icounter] != null &&
17             theFiles[icounter].getName().equalsIgnoreCase(soughtPathElement)) {
18             return theFiles[icounter].getName();
19         }
20     }
21     return null;
22 }
```

コード 2.1: FTP を介してファイル検索を行うソースコード ([5] より引用)

まず, このメソッドを含むソフトウェアのソースコードを解析し, コーパスを作成する. 次に, エディタにこのメソッドの冒頭 (FTPFile[] theFiles = listFiles(parentPath, false);) を入力する. ここでエディタ上からコンテンツアシスト機能を呼び出すことで, 補完候補を要求すると, コード 2.1 のメソッド本体の全体が補完候補として提示される.

2.4.2　本研究との比較

既存のソースコードをコーパス化し, 書きかけのソースコード片を入力として推薦を行う点が共通しているが, 山本らの研究では, 推薦されるコードがメソッド内の残りの部分の補完であり, そのまま使用できる形のソースコード片であるのに対し, 本研究で推薦される内容は, 識別子単体であり, 推薦に識別子同士の共起関係を用いる点が異なる.

2.5 Eclipse Code Recommenders

2.5.1 概要

Eclipse Code Recommenders[1]は, Eclipse プラグインの 1 つであり, Eclipse のコード補完機能にいくつかの補完エンジンを追加する. その 1 つにインテリジェントなコード補完 (Intelligent Code Completion) をする機能が備わっている.

一般的な IDE の標準的なコード補完機能では, クラス型の変数を入力後, ドット演算子を入力してフィールド変数やメソッドを呼び出そうとしたとき, 呼び出し可能な候補をアルファベット順に候補リストに表示する (図 2.6).

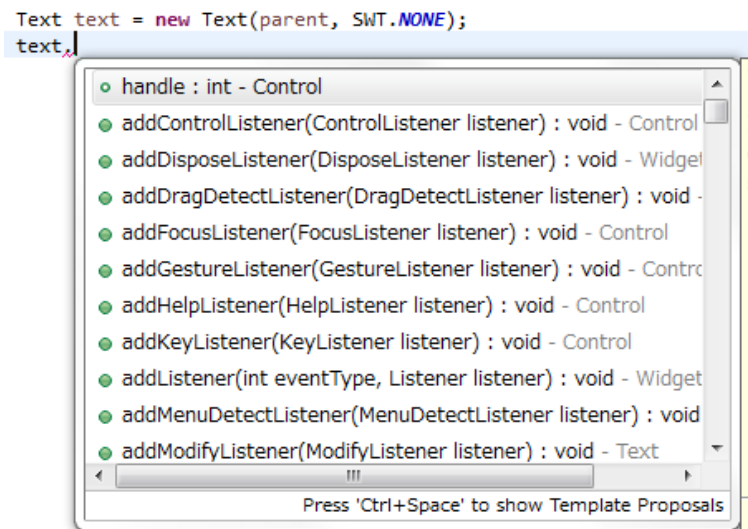


図 2.6: 標準的な補完機能 (アルファベット順)

Code Recommenders の機能を利用すると, 過去の統計情報を基に, 利用される可能性が高いメソッドを使用頻度順に並べて表示するようになる.

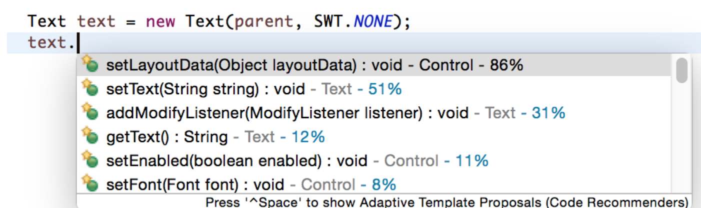


図 2.7: Code Recommenders による補完

図 2.7 は, Code Recommenders によって提示される候補リストの例である. Code Recommenders の補完機能を利用するためには, 補完対象ライブラリの API を使っている既存のソフトウェアの

[1]http://www.eclipse.org/recommenders/

ソースコードが必要であるが, 有名な API については標準でサポートしており, 開発者がソースコードを解析する必要はない.

2.5.2 本研究との比較

Code Recommenders は提示候補の順位付けに使用頻度を利用している点が本研究と異なる. 本研究では, 推薦した候補を相関ルールマイニングの確信度およびそれを応用した評価指標を用いて順位付けを行う.

第3章　相関ルールマイニング

3.1　相関ルールとは

相関ルールマイニング [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できる.

3.1.1　トランザクションデータ

相関ルールマイニングを利用している実例として, 小売業の POS システム (point of sales system) などが挙げられる. この場合, トランザクションデータにおいて, ID は顧客 1 人分を表し, アイテムは, その顧客が買った商品を表す. アイテムの順序は考慮しない. トランザクションデータは表 3.1 のようなデータ構造になっている.

表 3.1: トランザクションデータの例

ID(顧客)	アイテム (購入した商品)
1	{ おにぎり, パン, お茶 }
2	{ お茶, タバコ, 雑誌 }
3	{ パン, 菓子, おにぎり }
4	{ おにぎり, お茶 }
5	{ お茶, 菓子, おにぎり, タバコ }
6	{ 雑誌, おにぎり, お茶 }

このデータ構造をソースコードに対して適用しようと考えると, メソッドが 1 つのトランザクションデータであり, その内側に含まれる識別子がアイテムになるという対応関係が考えられる.

提案手法では, ソースコードコーパスに含まれるメソッドを表 3.2 のような形でデータベースに保持する. また, 保持する識別子はローカル変数, フィールド変数, メソッド呼び出しに使用される識別子とする. 識別子は, 識別子名の他に, 型名も持つ. ローカル変数やフィールド変数の場合は, その変数の型名, メソッド呼び出しの場合は, そのメソッドの返り値を型名とする. 同じ識別子名でも, 型名が違う場合は別の識別子として扱う.

表 3.2: ソースコードにトランザクションデータを当てはめた例

メソッド名	登場する識別子
drawBox	{ x, y, width, height, color }
drawCircle	{ x, y, r, color }
printString	{ str, color }
calcPosition	{ sin, cos, x, y, z, PI }
setSize	{ width, height }
rotation	{ r, sin, cos, PI, angle }

表 3.1 のトランザクションデータからは以下のようにさまざまな相関ルールを作ることができる. ただし以下のルールは有用であるとは限らない. 作成したルールが有用であるかはいくつかの評価指標を用いて判断する.

- { おにぎり }⇒{ お茶 }
  おにぎりを買った人がお茶も買う.
- { おにぎり, お茶 }⇒{ パン }
  おにぎりとお茶を買った人がパンも買う.
- { 菓子 }⇒{ パン, おにぎり }
  菓子を買った人がパンとおにぎりも買う.
- { おにぎり, お茶 }⇒{ 菓子, タバコ }
  おにぎりとお茶を買った人が菓子とタバコも買う.

有用な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. これらの指標の値が大きいほど, その相関ルールは有用なルールであるといえるため, これらの値に最小閾値を設定し, 有用でないルールを除外する.

また, 提案手法では結論部のアイテム数が複数になるルールは扱わない. これは, 推薦候補を表 1.1 のようにリスト形式で表示し, リストの 1 つの項目に 1 つの識別子を表示するためである.

3.1.2　支持度

支持度とは, アイテム集合に対して定義される値で, ルールを構成するアイテム集合の出現率であり, 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. 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」というデータが得られた場合, 支持度は 10% となる.

アイテム集合 X があるとすると, 支持度 sup は以下のように表される.

$$sup(X) = \frac{X\text{の全要素を含むトランザクションの数}}{\text{全トランザクション数}}$$

ルール $X \Rightarrow Y$ の支持度は X と Y の和集合をとり $sup(X \cup Y)$ として求める. このとき, X, Y を含むトランザクションの集合をそれぞれ TD_X, TD_Y とすると, $sup(X \cup Y)$ は TD_X と TD_Y の積集合として求められる (図 3.1).

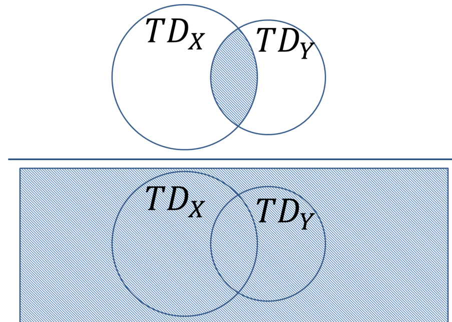


図 3.1: 支持度

支持度が高いほど, そのルールが多く出現していることを意味する.

3.1.3　確信度

確信度とは, 相関ルールに対して定義される値で, 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, 確信度は 70%となる.

アイテム集合 X, Y があるとすると, 確信度 $conf$ は, 支持度 sup を用いて以下のように表される.

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

図 3.1 と同様に確信度をベン図で表現すると図 3.2 のようになる.

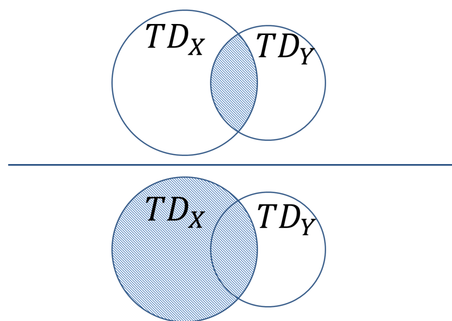


図 3.2: 確信度

3.1.4　リフト値

リフト値も相関ルールに対して定義される値である. 基本的に, 確信度が高いほど, X と Y の関連は強くなる. 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ことが考えられた. 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.

3.2　アプリオリアルゴリズム

3.2.1　概要

有用なルールを見つけるためには, 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必要がある.

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} ⇒ {d} や {b，c} ⇒ {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しようとするととても時間がかかる. 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.

本節ではそのアルゴリズムについて [1] に沿って説明する.

3.2.2　頻出アイテム集合の抽出

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.

3.2.3　相関ルールの抽出

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} ⇒ {c}，{a} ⇒ {b,c}，{a,c} ⇒ {b} . . . 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.

第4章　提案手法

本章では, 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こと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.

4.1　識別子同士の共起関係の解析

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. 他のオブジェクトのフィールド参照もフィールド変数として取得する. 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.

4.2　識別子の推薦

識別子の共起関係の推薦には 3 章で述べた相関ルールを基にした指標を用いる. 提案手法では, 相関ルールの抽出の際, 条件部と結論部のアイテム数が 1 つとなる集合のみに限定する. 条件部のアイテム数が複数となるルールについては 4.3.3 節で述べる.

推薦元となる識別子は, 編集中のメソッド内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である. これらの識別子を相関ルールの条件部としてもつルールをデータベースから検索し, 確信度の高い順に並べることで推薦候補とする. 以降の節で, 確信度を基にした推薦指標を提案する. それらの推薦指標を評価値と呼ぶ.

また, 異なる条件部から同じ結論部が推薦されることがある. このとき, 推薦候補の評価値の値として 2 通りの方法が考えられる. 1 つ目は確信度の合計値を評価値にする方法で, 2 つ目は確信度の最大値を評価値にする方法である. 以降, これらの計算方法をそれぞれ合計確信度推薦, 最大確信度推薦と呼ぶ. 2 つの計算方法のどちらの方が有用かについては 7 章以降で検証する.

次節でこの推薦の例を示す.

4.2.1　例

例えば, コード 4.1 のようなコードを記述しているとする. このソースコード片を入力とした識別子の推薦結果は次の手順で求める.

```
1  public String encode(final String string) {
2      int end = string.indexOf(',');
3
4      if(-1 == end) {
5          return string;
6      }
7
8      final StringBuffer sb = new StringBuffer();
9
10     ...
11
12 }
```

コード 4.1: 途中まで記述されたコード (Apache Ant のソースコードに含まれるメソッドの一部)

1. このメソッド内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する. これにより, `string`, `end`, `indexOf`, `sb` を得る.

2. それぞれの識別子が条件部となる相関ルールをデータベースから取り出し, 確信度の高い順に並べる. 条件部となる集合はアイテム数 1 で, {`string`}, {`end`}, {`indexOf`}, {`sb`} がそれぞれ条件部となる. 取得したルールは表 4.1 のようになる.

表 4.1: コード 4.1 から抽出された識別子毎のルール (括弧内は型名)

条件部 (推薦元)	結論部 (推薦候補)	確信度
string (String)	i (int)	0.2632
	length (int)	0.2632
	substring (String)	0.2105
	indexOf (int)	0.1579
	⋮	
end (int)	start (int)	0.6154
	length (int)	0.5385
	substring (String)	0.5385
	charAt (char)	0.4615
	⋮	
indexOf (int)	substring (String)	0.5116
	length (int)	0.3101
	toString (String)	0.2558
	i (int)	0.2093
	⋮	
sb (StringBuffer)	toString (String)	0.9487
	append (StringBuffer)	0.9231
	length (int)	0.4103
	i (int)	0.3846
	⋮	

3. これらのルールの結論部が推薦候補となるが, {`indexOf`} ⇒ {`toString`} と {`sb`} ⇒ {`toString`} のように同じ識別子を推薦することがある. この時の評価値は, 複数のルールの確信度を合計

した値を評価値とする合計値確信度推薦または, 複数のルールの確信度のうち値の大きい方を評価値とする最大確信度推薦のどちらかの方法を用いて計算する. これらの方法で評価値の高い順にソートした結果を推薦候補として提示する. コード 4.1 を入力とした場合の推薦結果は表 4.2 または表 4.3 となる.

表 4.2: 合計確信度推薦による評価

推薦候補	推薦元	評価値
length (int)	string (String), end (int), indexOf (int), sb (StringBuffer)	1.5221
substring (String)	string (String), end (int), indexOf (int)	1.2606
toString (String)	indexOf (int), sb (StringBuffer)	1.2045
append (StringBuffer)	sb (StringBuffer)	0.9231
⋮		

表 4.3: 最大確信度推薦による評価

推薦候補	推薦元	評価値
toString (String)	sb (StringBuffer)	0.9487
append (StringBuffer)	sb(StringBuffer)	0.9231
start (int)	end (int)	0.6154
length (int)	end (int)	0.5385
⋮		

4.3　推薦精度の向上

提案手法では, 確信度に対して, 追加の指標による重み付けを行うことで, 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.

4.3.1　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, 確信度に対して重みづけを行う. 推薦元となる識別子 X, 推薦対象となる識別子 Y の型を X_{type}, Y_{type} とすると, 評価値 $score$ は重み係数 k_{type} と確信度 $conf$ を用いて次のように表される.

$$score = \begin{cases} (1 + k_{type})conf & (X_{type} = Y_{type}) \\ conf & (X_{type} \neq Y_{type}) \end{cases} \tag{4.1}$$

例えば, 表 4.1 の確信度に対して $k_{type} = 0.5$ で重み付けを行うと, 表 4.4 のようになる. 型名による重み付けではこれらの重み付き確信度から合計確信度推薦または最大確信度推薦を行い, 推薦候補を提示する.

表 4.4: 型名による重み付けを行ったルール ($k_{type} = 0.5$)

条件部 (推薦元)	結論部 (推薦候補)	評価値
string (String)	i (int)	0.2632
	length (int)	0.2632
	substring (String)	**0.3157**
	indexOf (int)	0.1579
	⋮	
end (int)	start (int)	**0.9231**
	length (int)	**0.8077**
	substring (String)	0.5385
	charAt (char)	0.4615
	⋮	
indexOf (int)	substring (String)	0.5116
	length (int)	**0.4651**
	toString (String)	0.2558
	i (int)	**0.3139**
	⋮	
sb (StringBuffer)	toString (String)	0.9487
	append (StringBuffer)	**1.3846**
	length (int)	0.4103
	i (int)	0.3846
	⋮	

4.3.2　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ことが多い. また, 同様に変数名は名詞で終わることが多い. 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, 重要語同士の確信度を計算した. この確信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.

$$score = conf + k_{keyword} \cdot conf_{keyword} \quad (4.2)$$

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確信度である.

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} の確信度に, 重要語のコーパスを用いて計算した {lock } ⇒ { unlock} の確信度に重み付けした値を足して求める.

重要語の抽出は, camelCase や snake_case で連結された識別子名を単語ごとに分解し, 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できれば, それを重要語とする.

4.3.3　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 {string, end, indexOf, sb} 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 {string, end, indexOf, sb} で構成された頻出アイテム集合 L を抽出する.
   ここで, L = {{string}, {end}, {indexOf}, {sb}, {string, sb}, {string, indexOf}} 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' = {{end}, {string, sb}, {string, indexOf}} 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.

第5章　推薦システム

5.1　概要

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.

推薦候補の表示は Eclipse のビュー部分で行う. Eclipse のコード編集画面は複数のビューとエディタで構成されており, エディタでコードの編集, ビューで情報の表示を行う (図 5.1).

本システムに対して, 開発者はなんらかの操作をする必要はない. 開発者は, 本システムのビューを Eclipse の画面上に表示した状態で, エディタ内のソースコードを編集する. すると, 本システムがソースコードの内容を自動的に取得して自動的に推薦候補を提示する.

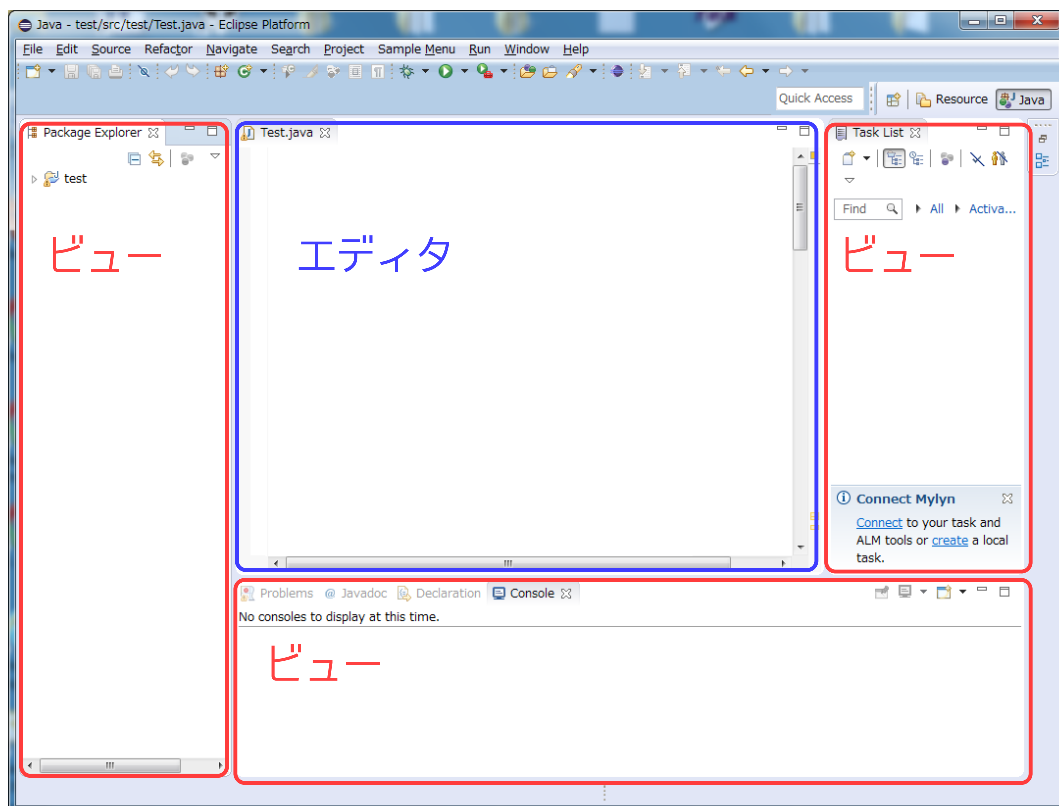


図 5.1: Eclipse 画面

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.

5.2　内部処理

本システムの内部処理の概要について述べる. 詳細は 6 章で扱う.

5.2.1　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.

5.2.2　推薦元の取得

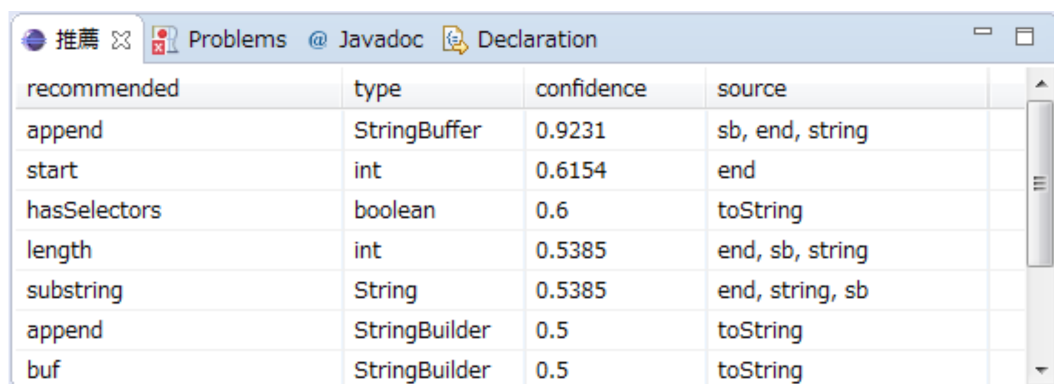
開発者が現在編集しているソースコードのカーソルの位置を取得し, カーソル位置がメソッドの内側であるかを調べる. メソッドの内側であった場合, そのメソッド内にあるフィールド変数, ローカル変数, メソッド呼び出しの識別子を取得する.

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確信度の評価を行い, 共起関係のある識別子を抽出する.

5.2.3　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, 推薦候補として確信度の高い順に提示する. 確信度として利用する指標には 4.3 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できる.

推薦候補は図 5.2 のように Eclipse のビューで表示される. 表示内容の更新は 5.2.2 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.



recommended	type	confidence	source
append	StringBuffer	0.9231	sb, end, string
start	int	0.6154	end
hasSelectors	boolean	0.6	toString
length	int	0.5385	end, sb, string
substring	String	0.5385	end, string, sb
append	StringBuilder	0.5	toString
buf	StringBuilder	0.5	toString

図 5.2: 推薦ビュー

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ことにより, 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.

5.3 設定画面

各指標の閾値の設定はプラグインの設定画面で行う. 設定画面は Eclipse のメニューバーから各プラグインの設定画面を開くことで表示される. 設定画面は図 5.3 のようになっており, 閾値の設定の他にも, さまざまなデータの設定を行うことができる.

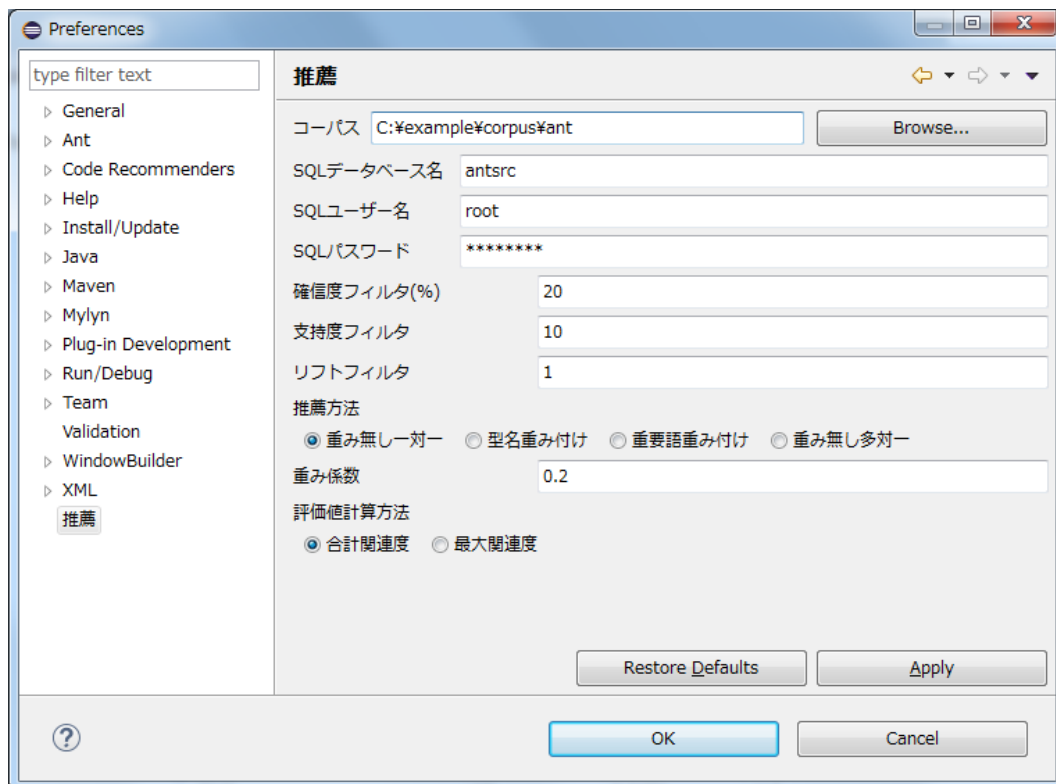


図 5.3: 設定画面

各項目では以下のような設定を行うことができる.

コーパス

データベースに登録するソースコードのあるディレクトリを設定する. 設定を変更した後, OK または適用ボタンを押すと, ディレクトリ内のソースコードを新たなコーパスとして下の項目で設定した SQL データベースを更新する.

SQL データベース名

コーパスとして使用する SQL データベースの名前を設定する.

SQL ユーザー名

SQL を利用するユーザーの名前を設定する.

SQL パスワード

SQL にアクセスするためのパスワードを入力する.

確信度フィルタ

推薦候補として提示する識別子の確信度の下限値. パーセントで入力する. この数値未満になった候補は推薦ビューに表示されず, これによりあまり関連していないルールを除外することができる.

支持度フィルタ

推薦候補として提示する識別子の支持度の下限値. ここでの支持度は, 識別子がコーパス内に登場した回数でフィルタリングするため, 整数値を入力する. この数値未満になった候補は推薦ビューに表示されず, これによりほとんど登場しないルールを除外することができる.

リフトフィルタ

推薦候補として提示する識別子のリスト値の下限値. この数値未満になった候補は推薦ビューに表示されず, これにより頻繁に登場するありふれたルールを除外することができる.

推薦方法

4 章で示した推薦方法を切り替える. 切り替える推薦方法の種類は, 一対一のルールによる推薦, 型名による重み付け推薦, 重要語による重み付け推薦, 多対一のルールによる推薦の 4 種類である.

重み係数

上の推薦方法の項目での, 型名重み付けと重要語重み付けの推薦で利用する重み係数の設定を行う.

評価値計算方法

4.1 節で述べた合計確信度推薦と最大確信度推薦を切り替える.

これらの設定は適用または OK ボタンを押すと即座にビュー画面に反映される.

5.4　ステータスバーの表示

推薦ビューにフォーカスを合わせると, ステータスバーに現在の情報が表示される. 推薦処理中は図 5.4 のように表示される. 推薦が終わる前に入力候補が切り替わっても新たな推薦は行わない. 推薦処理は別スレッドで行っているため, 推薦処理中でも利用者は他の IDE の機能を利用することができる.

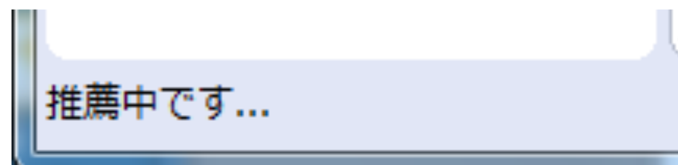


図 5.4: 推薦中

推薦処理が終わり, ビューに推薦候補が提示されると, 図 5.5 のように推薦を行ったメソッドの名前が表示される.

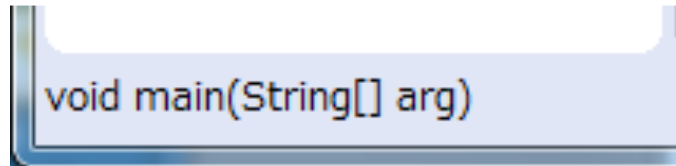


図 5.5: メソッド名の表示

第6章　実装

6.1　開発環境

Java を用いて Eclipse 上で開発した. また, ソースコードの解析結果を保管するデータベースとして MySQL を使用した.

6.2　ソースコード解析

ソースコードコーパスとして指定された既存のプロジェクトのソースコードを解析し, データベースへ登録する. データベースへ登録された情報から相関ルールマイニングを行い, 有用なルールを発見する.

ソースコードの解析は, コーパスのディレクトリを設定した最初の 1 度だけ行う.

6.2.1　データベースへの登録

ソースコードコーパスは設定画面によって設定したディレクトリから読み込まれる. 読み込むファイルは, 拡張子が.java となっているファイルである. サブディレクトリが存在する場合はその中も全て読み込む.

トランザクションデータへの変換は以下の手順で行う.

1. 読み込まれたソースコードを Eclipse JDT の ASTParser によって抽象構文木へ変換する. 抽象構文木は, Java の各要素を ASTNode というクラスのツリーとして保持している. この構文木は Visitor パターンを用いることで辿ることができる.

2. Visitor によって構文木を辿り, メソッド定義のノードを訪れた時に, そのメソッドの名前を取得し, クラス定義またはフィールド変数宣言のノードを訪れた時に, 取得したメソッド名を破棄する. これによって, 現時点のノードが含まれるメソッドの名前を保持する.

3. Visitor が識別子のノードを訪れると, その識別子の名前と型名を取得し, データベースへ登録する. このとき, その識別子の重要語も抽出し, 識別子名, 型名と共に登録する.

これによってソースコードに含まれるメソッドを全てトランザクションデータに変換する.

6.2.2　重要語の抽出

重要語の抽出は, 次のような手順で行う.

1. camelCase や snake_case で連結された識別子名を単語ごとに分解し, OpenNLP[1]を利用して品詞解析を行う.
2. 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できれば, それを重要語とする.
3. 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[2]を利用して動詞, 名詞に相当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.

6.3　推薦候補の提示

推薦候補は Eclipse のビュー画面で表示される.

6.3.1　編集中のカーソル位置取得

カーソルの位置が変化したとき, 推薦を行うかの判定を行う. 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.

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.

カーソル位置を含むメソッドの取得にも ASTParser を用いる. 取得は以下の手順で行う.

1. 編集中エディタ内のコードを取得し, ASTParser を使って抽象構文木へ変換する.
2. 構文木を辿り, ノードがメソッド定義だったときはそのメソッド名とエディタ内での位置を取得する. このとき, メソッド内に含まれる識別子も取得し, リスト化しておく. この情報は推薦処理時に使用する.
3. エディタ内でのメソッド位置を取得したら, 実際のカーソル位置と比較し, カーソル位置を含むメソッドを調べる.

ここで得たメソッド名とその内側の識別子名リストを用いて推薦を行う.

6.3.2　候補リスト

編集中のメソッド取得時に同時に取得した識別子名リストが, これまで保持していた識別子名リストと一致しなかった場合, 新たに推薦処理を行う. 推薦は識別子名リストに含まれる識別子を入力として, データベースからルールを検索する.

推薦は 4 章で述べた手法を用いて行う. このとき, 設定画面にて設定した評価指標のフィルタ値により, 閾値未満の推薦候補を除外する.

得られた推薦候補は評価値の高い順にソートされてビュー画面に表示される.

[1] https://opennlp.apache.org/
[2] https://wordnet.princeton.edu/

6.4　スレッド処理

推薦処理は別スレッドを生成して行う. そのため, コード編集中にこのプラグインが原因で Eclipse が固まって操作が一時的にできなくなるようなことは無い.

第7章　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, 提案手法の確信度に基づく評価指標が有効なものであったかの評価を行った.

7.1　実験対象

実験対象のプロジェクトとして, Apache Ant[1]を利用した. バージョンは 1.9.7 で, 総ファイル数は 873, 総行数は 223,141 行である. 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 57,533 個をデータベースに格納してコーパスとした.

7.2　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. 例えば, あるメソッドに識別子が 8 種類存在するなら, そのうちのメソッド後半に登場する 4 つを削除する. 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.

実験の評価尺度については, 推薦成功率と MRR を用いる. 推薦成功率とは, 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する. MRR については 7.5 節で述べる.

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, 実験内では確信度, 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.

7.3　頻度順推薦

比較対象として, コーパスに登場する頻度順に並んだ識別子を推薦候補として提示する頻度順推薦に対しても実験を行う. 頻度順推薦では, コーパス内の識別子に対して, 同名の識別子がそれぞれ何回登場するかをカウントし, 登場回数の多い上位 10 個の識別子を推薦候補とする.

7.4　実験方法の例

実験の例として以下のコード 7.1 に対して評価実験の手順を適用した場合を示す.

[1] http://ant.apache.org/

```
1  public String encode(final String string) {
2      int end = string.indexOf(',');
3
4      if (-1 == end) {
5          return string;
6      }
7
8      final StringBuffer sb = new StringBuffer();
9
10     int start = 0;
11
12     while (-1 != end) {
13         sb.append(string.substring(start, end));
14         sb.append("\\,");
15         start = end + 1;
16         end = string.indexOf(',', start);
17     }
18
19     sb.append(string.substring(start));
20
21     return sb.toString();
22 }
```

コード 7.1: Apache Ant のソースコードに含まれるメソッド

1. このメソッドに含まれるローカル変数, フィールド変数, メソッド呼び出しの識別子を取り出すと, { `string`, `end`, `indexOf`, `sb`, `start`, `append`, `substring`, `toString` }の 8 個の識別子が得られる.
2. この識別子集合から, 後半 5 割部分を削除し, 前半部分の集合 { `string`, `end`, `indexOf`, `sb` } を取り出す.
3. 前半部分の集合を入力として提案手法による推薦を行う.
4. 推薦候補として提示された識別子のうち, 上位 10 個を取得する. その結果, 表 7.1 のような候補が提示されていたとする.

表 7.1: { `string`, `end`, `indexOf`, `sb` } から推薦された候補の例

推薦候補	型名
toString	String
append	StringBuffer
start	int
length	int
substring	String
charAt	char
i	int
c	char
len	int
LINE_SEP	String

5. 得られた推薦候補と削除した後半部分 { `start`, `append`, `substring`, `toString` } を比較する. この場合, 推薦候補の中に, 後半部分の識別子が含まれているので, それらの順位を記録し, こ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. 識別子の比較は, 型名まで一致して同一の識別子であるとする. メソッドに対する推薦の成功の判定は, 後半部分の識別子が 1 つでも推薦候補に含まれていれば, 成功とする.

この手順を, コーパスに含まれる識別子数 3 以上の全てのメソッドに対して行う.

7.5　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[2])[6] 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.

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

例えば, 5 個のメソッドに対して候補を提示し, それぞれのメソッドで正解候補が現れた順位が 1 位, 3 位, 8 位, 2 位, 5 位だったとすると, MRR は $(\frac{1}{1}+\frac{1}{3}+\frac{1}{8}+\frac{1}{2}+\frac{1}{5})/5 \simeq 0.431$ のように計算する.

[2]Mean Reciprocal Rank

第8章　結果と考察

8.1　頻度順推薦との比較

頻度順推薦と, 合計確信度推薦を用いた一対一の重み無し推薦の評価実験の結果は表 8.1 のようになった.

表 8.1: 頻度順推薦と提案手法の比較

	頻度順	一対一
推薦成功率	0.389	0.581
MRR	0.165	0.378

表 8.1 から, 提案手法による推薦は頻度順推薦による推薦よりも推薦成功率と MRR の面で優れているということが確認できる.

8.2　提示候補は適切であったか

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 8.1 のようになった.

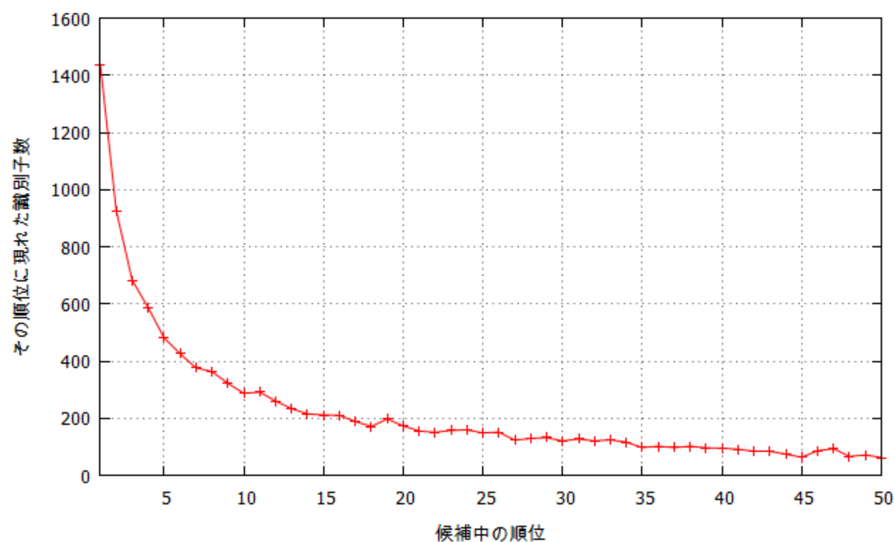


図 8.1: 後半部分に含まれていた識別子の候補中の順位

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.

8.3　各評価指標の比較

各評価指標での推薦成功率と MRR を表 8.2 に示す. 型名と重要語は, 式 (4.1)(4.2) の重み係数 $k = 0.5$ の場合の数値である.

表 8.2: 推薦成功率と MRR

	推薦方法	一対一	多対一	型名	重要語
推薦成功率	合計確信度推薦	0.581	0.569	0.579	0.531
	最大確信度推薦	0.563	0.554	0.561	0.501
MRR	合計確信度推薦	0.378	0.371	0.369	0.343
	最大確信度推薦	0.356	0.354	0.341	0.324

表 8.2 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確信度による評価との比較では, 推薦成功率と MRR の増加は見られなかった.

また, 推薦成功率と MRR について, 合計確信度推薦と最大確信度推薦での比較を図 8.2 と 図 8.3 に示す.

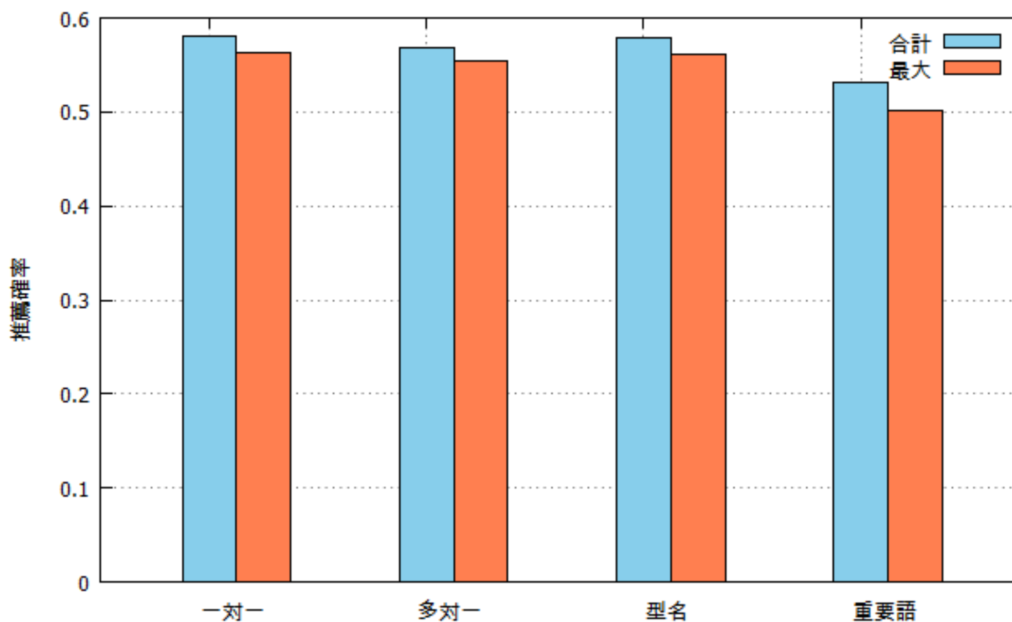


図 8.2: 各評価指標での推薦成功率

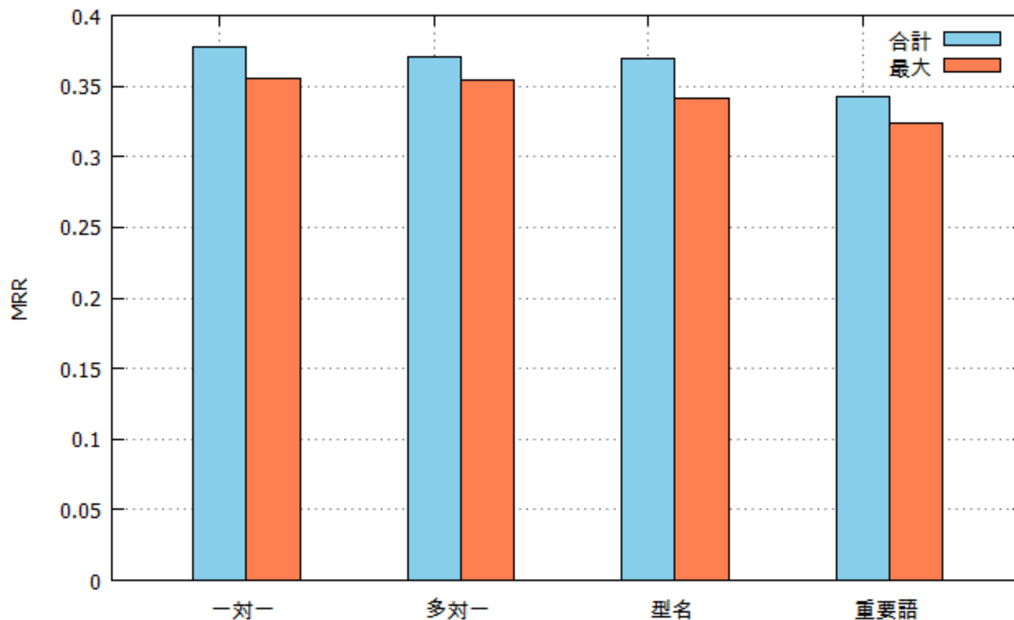


図 8.3: 各評価指標での MRR

8.4　合計確信度推薦と最大確信度推薦

図 8.2 と図 8.3 の結果から, 合計確信度推薦と最大確信度推薦を比較すると, どの評価指標を用いた場合においても合計確信度推薦の方が, 推薦成功率, MRR ともに高くなっていることが分かる.

しかし, 筆者がこの推薦システムを試用して 2 つの推薦を比較したところ, 合計確信度推薦の推薦候補には `toString` や `i` といった頻繁に出てくるありふれた識別子が比較的多く出現するように感じた. そのため, これを検証するために, 2 つの推薦によって提示された候補全体のリフト値の平均を求めてみると, 合計確信度推薦の場合ではリフト値の平均が 74.8, 最大確信度推薦の場合ではリフト値の平均が 91.2 となった. リフト値は, 値が低いほど提示された推薦候補の出現率が高いことを意味している. そのため, 合計確信度推薦で提示される候補は, 最大確信度推薦で提示される候補よりもありふれた名前の識別子であることが多いことが分かる. ありふれた名前の識別子は, 出現しやすいものであるため, それらを推薦することは理にかなっているとも考えられるが, ありふれた名前の識別子は開発者にとって既知である可能性も高くなる. よって, 合計確信度推薦は推薦成功率と MRR においては最大確信度推薦よりも良い結果が出たが, 一概に最大確信度推薦よりも優れているとは言えないと考えられる.

8.5　多対一のルールによる推薦

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.

8.6　重み係数による変化

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成功率の変化をグラフでプロットした結果, 図 8.4 のようになった.

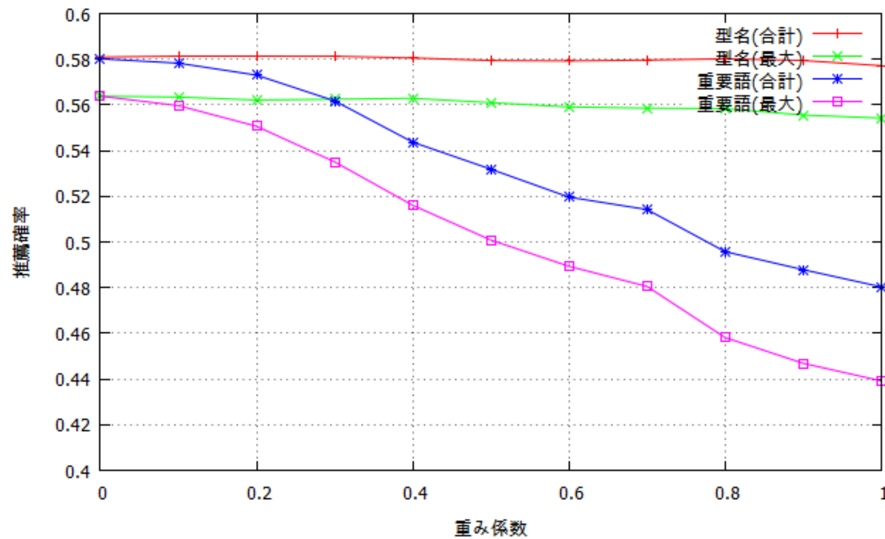


図 8.4: 重み付けによる推薦成功率の変化

また, 重み係数による MRR の変化をグラフでプロットした結果, 図 8.5 のようになった.

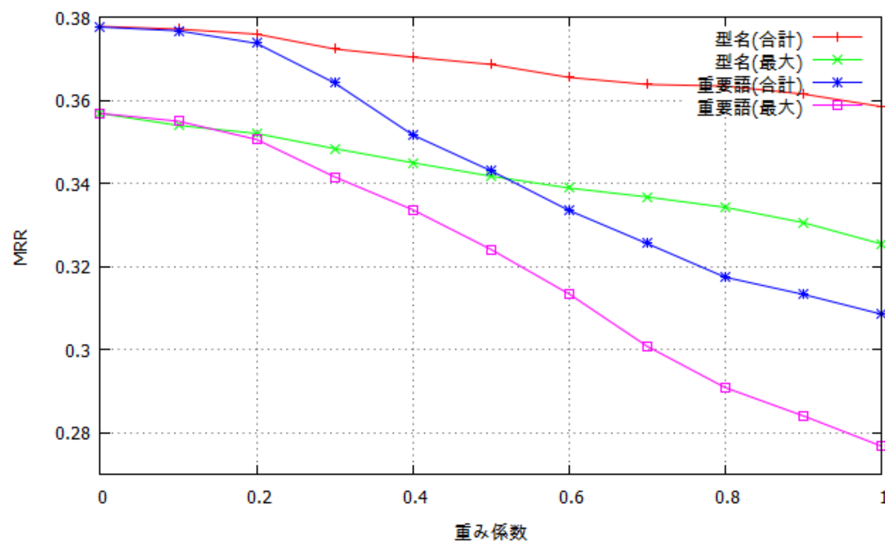


図 8.5: 重み付けによる MRR の変化

図 8.4, 図 8.5 の結果から, 重み付けによる評価では推薦成功率や MRR が下がってしまうということが確認できる.

8.7　重要語による重み付け

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.

8.8　型名による重み付け

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. しかし, 型名による重み付け推薦では, 評価実験において推薦候補として提示された識別子のうち, 後半 5 割部分でも使用されていた候補 (以降, 正解候補と呼ぶ) は, その候補の推薦元となった識別子と同じ型を持つものが増加していた.

例えば, 表 4.3 の推薦候補 `start` (int 型) の場合, 推薦元は int 型の end である. このように, 正解候補と推薦元が同じ型をもっている確率は, 型名重み付け推薦の重み係数に比例して増加していることが確認できた. 重みの変化による推薦元と正解候補の型の一致率を測定したグラフを図 8.6 に示す. 合計確信度推薦の場合, 表 4.2 のように推薦元が複数になることがあるが, この場合はその推薦元の中で最も確信度の大きい推薦元と正解候補を比較した.

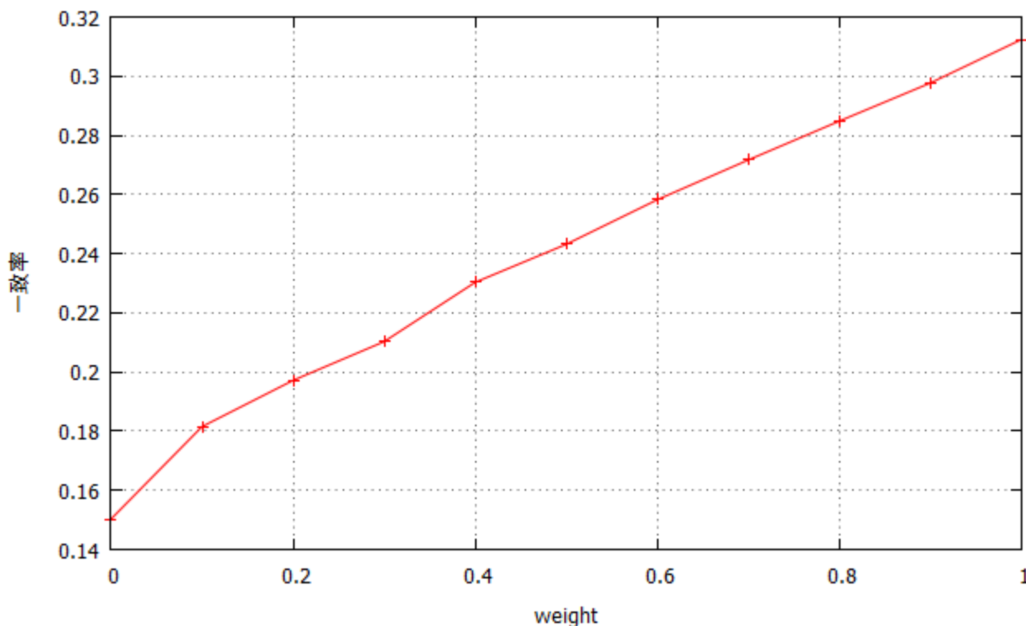


図 8.6: 推薦元と正解候補の型名の一致率

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.

第9章　結論

9.1　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こと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, 相関ルールマイニングを利用して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.

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 58% 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評価実験により示した. また, 推薦候補の評価値としていくつかの指標を提案し, これらの推薦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. 一方で, 本手法による推薦は頻度順推薦よりも優れていることを実験により示した. 更に, 本手法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.

9.2　今後の課題

9.2.1　推薦指標の改善

提案手法では, 型名による重み付け推薦, 重要語による重み付け推薦, 多対一のルールによる推薦といった推薦指標を提案し, これらの有効性を検証したが, 一対一のルールによる推薦と比較して, 推薦成功率, MRR のどちらも向上することが無かった. 型名による重み付け推薦は, 8.8 節で述べたように特定の状況下で有効な場合があると考えられるが, その状況判断は利用者に委ねられており, 使い勝手がいいとは言い難い.

そのため, 状況によらず一対一のルールによる推薦より優れた推薦候補を提示できる推薦指標を考案し, より良い推薦システムへ改善することが今後の課題として考えられる.

9.2.2　リファクタリング支援への応用

提案手法を応用し, 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指摘することで, リファクタリング支援ができると考えている.

例えば {`srcFile`} ⇒ {`dstFile`} という有用なルールがあり, 現在のメソッド内に `srcFile` と `destFile` という名前の識別子があったとき, `dstFile` を推薦することでより良い命名を促すことができると考えられる.

9.2.3　文脈の考慮

提案手法では, 開発者がメソッド内でこれから何を記述しようとしているのかを考慮せずに推薦を行っている. 記述中のソースコードの状態と, エディタ内でのカーソルの位置から, 開発者が変数の宣言, 関数の呼び出しなど, これから何を記述しようとしているのかが分かれば, 変数の宣言ならローカル変数に用いる識別子, 関数の呼び出しなら関数呼び出しに用いる識別子のみをそれぞれ推薦するなど, より良い推薦候補が提示できると考えられる.

このように, 変数名とメソッド名では, 候補として出てきてほしいタイミングが異なるので, ソースコードの文脈を考慮した推薦を行い, 文脈に応じた識別子名を提示するようにすることで, 推薦システムとしての利便性が向上すると考えられる.

また, メソッド呼び出しは, 呼び出しの順番に規則性があることが多いので, 順序を考慮した相関ルールを利用することでメソッド名の推薦結果を改善することができる可能性がある.

謝辞

本研究は, 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 コンピュータサイエンスコース 寺田研究室において, 寺田 実准教授のご指導のもと行われました.

寺田 実准教授には, 研究の方針の検討, アイデアの提供など, 数多くの助言を賜りました. 心より御礼申し上げます.

また, 岩崎 英哉教授には, 研究の助言, アイデアの提供など, 数多くの助言を賜りました. 心より御礼申し上げます.

寺田研究室の修士課程 2 年の鈴木 佑樹 さん, 平田 吉久さん, 本田 裕人さん, 修士課程 1 年の安部 文紀さん, 山本 愛美さん, 渡邊 裕貴さん, 学部 4 年の岡川 翔子さん, 佐々木 透さん, 下澤 一輝さん, 肥後 亮佑さん, 藤本 明優さん, 村松 啓寛さんには多くの研究についてのアイデアやアドバイスを頂き, また研究室での生活など数多くの面でお世話になりました. 心からの感謝を申し上げます.

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).

[6] E. Voorhees. “The TREC-8 Question Answering Track Report” Proceedings of the 8th Text Retrieval Conference, pp. 77-82 (1999).

付録

本システムが Apache Ant に含まれるメソッドの後半部分を削除したコード片から推薦した実際の候補を掲載する. 推薦結果は一対一の重み無し推薦によって提示されたものの上位 5 つである. 提示した識別子が本来の後半部分にも含まれていることが確認できる. 更に, 合計確信度推薦と最大確信度推薦で候補が変化するメソッドがあることも確認できる.

```
1  public String encode(final String string) {
2      int end = string.indexOf(',');
3
4      if (-1 == end) {
5          return string;
6      }
7
8      final StringBuffer sb = new StringBuffer
         ();
9
10 }
```

コード 10.1: 推薦対象

```
1      int start = 0;
2
3      while (-1 != end) {
4          sb.append(string.substring(start,
             end));
5          sb.append("\\,");
6          start = end + 1;
7          end = string.indexOf(',', start);
8      }
9
10     sb.append(string.substring(start));
```

コード 10.2: 本来の後半部分

表 10.1: コード 10.1 合計確信度推薦

推薦候補	型名	評価値
length	int	1.5221
substring	String	1.2606
toString	String	1.2045
append	StringBuffer	0.9231
i	int	0.8571

表 10.2: コード 10.1 最大確信度推薦

推薦候補	型名	評価値
toString	String	0.9487
append	StringBuffer	0.9231
start	int	0.6154
length	int	0.5385
substring	String	0.5385

```
1  public static Object invoke(Object obj,
     String methodName, Class<?> argType,
     Object arg) {
2
3  }
```

コード 10.3: 推薦対象

```
1      try {
2          Method method;
3          method = obj.getClass().getMethod(
             methodName, new Class[] { argType
             });
4          return method.invoke(obj, new Object
             [] { arg });
5      } catch (Exception t) {
6          throwBuildException(t);
7          return null; // NotReached
8      }
9      return null;
```

コード 10.4: 本来の後半部分

表 10.3: コード 10.3 合計確信度推薦

推薦候補	型名	評価値
getClass	Class	1.0145
invoke	Object	0.9273
getMethod	Method	0.6545
t	Exception	0.6545
method	Method	0.5636

表 10.4: コード 10.3 最大確信度推薦

推薦候補	型名	評価値
invoke	Object	0.7273
getClass	Class	0.56
getMethod	Method	0.4545
t	Exception	0.4545
method	Method	0.3636

```
1  private Hashtable<String, Object>
     getAllSystemProperties() {
2      Hashtable<String, Object> ret = new
         Hashtable<String, Object>();
3      Enumeration<?> e = System.getProperties
         ().propertyNames();
4
5  }
```

コード 10.5: 推薦対象

```
1      while(e.hasMoreElements()) {
2          String name = (String) e.nextElement
             ();
3          ret.put(name, System.getProperties
             ().getProperty(name));
4      }
5      return ret;
```

コード 10.6: 本来の後半部分

表 10.5: コード 10.5 合計確信度推薦

推薦候補	型名	評価値
getProperty	String	1.0
hasMoreElements	boolean	1.0
nextElement	Object	1.0
propertyNames	Enumration	0.5455
value	String	0.5455

表 10.6: コード 10.5 最大確信度推薦

推薦候補	型名	評価値
getProperty	String	1.0
hasMoreElements	boolean	1.0
nextElement	Object	1.0
propertyNames	Enumration	0.5455
value	String	0.5455

```
1 protected void writeDOMTree(Document doc,
    File file) throws IOException {
2     OutputStream os = new FileOutputStream(
        file);
3     try {
4         PrintWriter wri = new PrintWriter(
            new OutputStreamWriter(new
            BufferedOutputStream(os), "UTF8"
            ));
5
6
7     } finally {
8     }
9 }
```

コード 10.7: 推薦対象

```
1     try {
2         ...
3
4         wri.write("<?xml version=\"1.0\"
            encoding=\"UTF-8\" ?>\n");
5         (new DOMElementWriter()).write(doc.
            getDocumentElement(), wri, 0, "  "
            );
6         wri.flush();
7         // writers do not throw exceptions,
            so check for them.
8         if (wri.checkError()) {
9             throw new IOException("Error
                while writing DOM content
                ");
10        }
11    } finally {
12        os.close();
13    }
```

コード 10.8: 本来の後半部分

表 10.7: コード 10.7 合計確信度推薦

推薦候補	型名	評価値
close	void	0.9197
write	void	0.6699
createElement	Element	0.6154
log	void	0.5694
appendChild	Node	0.4615

表 10.8: コード 10.7 最大確信度推薦

推薦候補	型名	評価値
createElement	Element	0.6154
appendChild	Node	0.4615
close	void	0.4063
write	void	0.3438
flush	void	0.3077

```
1 public AntHandler onStartChild(String uri,
    String name, String qname,
2             Attributes attrs, AntXMLContext
                context) throws
                SAXParseException {
3
4 }
```

コード 10.9: 推薦対象

```
1     if (name.equals("project") && (uri.
        equals("") || uri.equals(ANT_CORE_URI
        ))) {
2         return ProjectHelper2.projectHandler
            ;
3     }
4     if (name.equals(qname)) {
5         throw new SAXParseException("
            Unexpected element \"{" + uri + "}
            "
6             + name + "\" {" + ANT_CORE_URI +
                "}" + name,
7             context.getLocator());
8     }
9     throw new SAXParseException("Unexpected
        element \"" + qname + "\" "
10        + name, context.getLocator());*/
```

コード 10.10: 本来の後半部分

表 10.9: コード 10.9 合計確信度推薦

推薦候補	型名	評価値
tag	String	2.0778
equals	boolean	1.7461
getLocator	Locator	1.316
getProject	Project	0.9287
i	int	0.9162

表 10.10: コード 10.9 最大確信度推薦

推薦候補	型名	評価値
tag	String	0.7
equals	boolean	0.5
getLocator	Locator	0.5
attrUri	String	0.3
elementHandler	AntHandler	0.3