修士学位論文

題目

潜在的意味解析法 LSA に基づくソフトウェアシステム分類法の提案

指導教官
井上 克郎 教授

報告者
川口 真司

平成 15 年 2 月 12 日

大阪大学 大学院基礎工学研究科
情報数理系専攻 ソフトウェア科学分野

平成 14 年度 修士学位論文

潜在的意味解析法 LSA に基づくソフトウェアシステム分類法の提案

川口 真司

内容梗概

近年, オープンソースソフトウェア開発が活発になるにつれて, SourceForge に代表されるソフトウェアリポジトリサービスが頻繁に利用されるようになってきている. ソフトウェアリポジトリサービスは, ソースコードやドキュメントなどのソフトウェア開発におけす様々な生成物の保管場所や, コミュニケーションの場となるメーリングリストなどのサービスを提供している. 現在のところ, 約 55,000 を越えるプロジェクトが存在する. 現在, それら大量のソフトウェアは現在手動で分類されている. 手動で分類を行うには分類対象のソフトウェア全てに対して深い知識が必要であり, 分類するソフトウェアの増加にしたがって困難なものとなってきている. 一般的に, これらのソフトウェアはあらかじめ定められた階層構造に基づいて分類される. このとき用いられる視点はソフトウェアがどのような機能 (ワードプロセサ, 表計算など) を提供するか, という点に重きがおかれる. しかし, ソフトウェアを分類する視点としては, この他にもどのようなライブラリを利用しているか, どのようなアーキテクチャを前提としているか, などが存在し, これらの視点もまた有用な視点である. このような多様な観点を全て満たす階層構造の作成には多様なライブラリ・アーキテクチャに対して深い知識が必要である. また, これらのライブラリは日々新しくなるため, 階層構造の維持には多大な労力が必要である.

本研究ではソースコードのみを入力とする分類手法の提案を行う. 特に, ソフトウェアが持つ非単一性に着目し, ソフトウェアを非排他的に分類する手法を提案する. 本手法を用いることにより, 分類対象となるソフトウェアに対して深い知識がなくとも, ライブラリやアーキテクチャに着目した分類を自動的に行うことが可能となる. 本手法は分類のキーとして, 変数名, 関数名などの識別子に着目する. これら識別子は, プログラムの動作の一断片を表している. 識別子をトークンとして抽出し, トークン間の類似度を潜在的意味解析手法 LSA を用いて算出する. そして類似したトークン同士をクラスタ分析によって抽出し, 一つのクラスタを得る. そうして得られたトークンのクラスタからソフトウェアのクラスタを算出する. また, 提案する手法を実現するシステムを構築を行い, 実際のソフトウェアの分類を行った. その結果, 提案手法により前提知識がなくともさまざまな観点からの分類が可能なことを確認した.

主な用語

自動分類 (Automatic Categorization)
ソフトウェアリポジトリ (Software Repository)
潜在的意味解析 (LSA)

目次

1 はじめに **5**

2 ソフトウェア分類 **7**

2.1 既存の関連研究 8

3 提案手法 **9**

3.1 概要 9

3.2 トークン抽出 9

3.3 共起行列の作成 11

3.4 孤立トークン, 普遍的トークンの削除 11

3.5 LSA 12

3.6 トークン間の類似度計測, クラスタ分析 12

3.7 ソフトウェアクラスタの作成 14

3.8 クラスタタイトルの作成 14

4 ソフトウェア分類システム **16**

4.1 ファイル形式 17

4.1.1 list 形式 17

4.1.2 mat 形式 17

4.1.3 cl 形式 18

4.2 トークン切り出しプログラム 19

4.3 共起行列作成プログラム 19

4.4 孤立トークン, 普遍トークンの削除を行うプログラム 19

4.5 LSA を行うプログラム 19

4.6 類似度計測, クラスタリングプログラム 20

4.7 ソフトウェアクラスタ生成プログラム 21

4.8 クラスタタイトル生成プログラム 21

4.9 クラスタファイル群整形プログラム 22

4.10 統合プログラム 22

5 実験 **24**

5.1 実験方法 24

5.2 結果 24

5.3 考察 30

6 まとめ **32**

謝辞 **33**

参考文献 **34**

付録 **36**

A Latent Semantic Analysis **37**

A.1 ベクトル空間モデル 37

A.2 tf, idf 38

A.3 特異値分解 38

1 はじめに

近年, オープンソースソフトウェアシステムの数は急速に増加しており, それに呼応するように, 多数のソフトウェアプロダクトを保持するサービスが提供されている. 例えばオープンソースソフトウェア開発向けに SourceForge[15] と呼ばれるサービスが存在する. その他, 同様のサービスとして SourceCast[14], OSDL(Open Source Development Lab.)[13] が存在する. また, 社内プロジェクトの集積場として Corporate Source [5, 6] が提案されている.

このようなサービスでは, ソースコードのストレージやコミュニケーションに必要なメーリングリストなどソフトウェア開発に必要とされる物的資源を提供することを主な目的としている. だが, 保持するソフトウェアプロダクトが増加するにつれて, 開発基盤としての価値だけでなく未知のソフトウェアプロダクトを検索, 発見する場としての価値も高まっている. 例えば SourceForge は 2003 年 2 月現在 55,000 以上ものプロジェクトを保持しており, 開発者として登録されているユーザ数は 55 万を越える.

一般的にこれらのサービスではソフトウェアがいわゆるディレクトリ構造で分類されている. ディレクトリ構造による分類では, サービス管理者がディレクトリ構造をあらかじめ定義しておき, そのディレクトリ構造に沿った形で各種ソフトウェアが分類される.

しかし, 実際にソフトウェアを分類する観点はさまざまな視点が考えられる. たとえば, 多くのソフトウェアリポジトリではソフトウェアが提供する機能 (ワードプロセサ, 表計算など) を用いて分類が行われているが, ソフトウェアが利用しているライブラリや, 前提となる動作環境などといった視点もまた分類基準として考えられる. このように様々な観点を同時に満たすディレクトリ構造の作成には多様なライブラリに対する深い知識が必要であり, 非常に困難である. また, 作成できたとしてもライブラリやアーキテクチャは次々と新しいものに置き換えられるため, そのつどディレクトリ構造を更新しなくてはならない. このため, 有用なディレクトリ構造を維持するためには多大な労力が必要である. 特にソフトウェアリポジトリでは, 多数のソフトウェアのコンパイル済みのバイナリファイルのみならず, ドキュメントやソースコードも公開されており, 単なる一般ユーザだけでなく, ソフトウェア作成者にとっても手本となるソフトウェアプロダクトを検索するなどの利用法が存在する. そのため, このような観点による分類も非常に有用である.

そこで, 本研究では, 自然言語で記述された文書を分類するための手法である LSA(Latent Semantic Alasysis)[9] を応用して, ソースコードから機械的にソフトウェアを分類する手法の提案を行う. 本手法は, 事前にカテゴリなどの前提知識を用いるとなく, ソースコードのみを用いて分類を行えるという特徴がある.

LSA では, まず分類対象である文書から単語を抽出し, 単語 - 文書行列を作成する. 本手法でも同様に分類対象のソフトウェアから識別子を抽出し, 識別子- ソフトウェア行列を作成

する.

　また, 提案手法を実現するシステムの実装を行った. そしてこのシステムを用いて分類を行い, その有用性を確かめた.

2　ソフトウェア分類

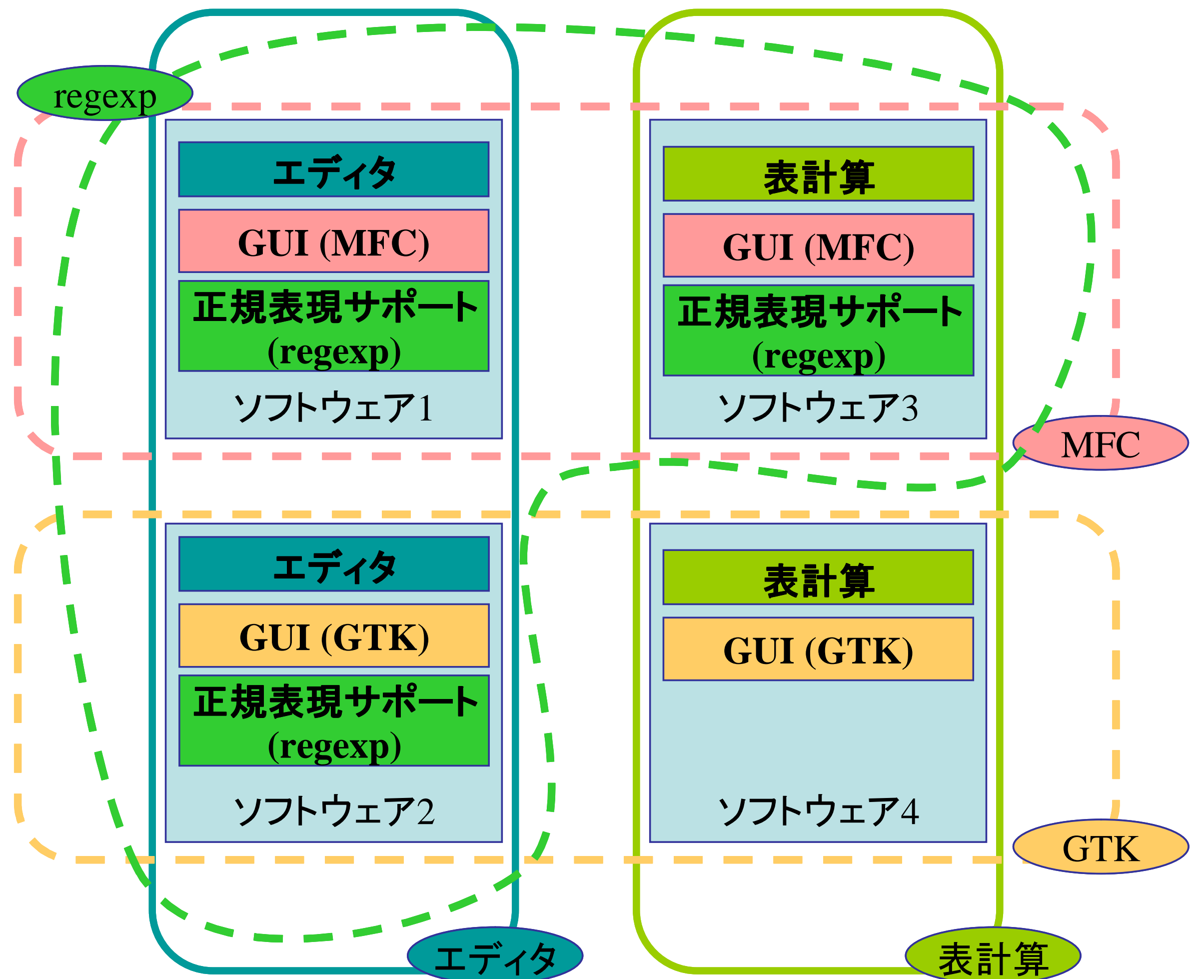

図 1: 非排他的ソフトウェア分類モデル

ソフトウェアは単一の部分のみで成りたっているわけではなく, 様々な側面を内包している. 例えば, Windows 上で動作するエディタを考える. このエディタにはエディタそのものとしての機能を実現する部分だけではなく, Windows での GUI フレームワークである MFC アプリケーションという一面も存在する. また, このエディタは正規表現による検索もサポートしているとすると, 正規表現を解釈し, マッチするテキストを検索する部分も持っていることになる.

このように, ソフトウェアは複数の面を持っており そのような存在を単一の観点から排他的に分類しても, 必ずしも有効な分類結果とは言えない. 様々な面のうち, 最も特徴的な, し

かし閲覧者にとって重要とは限らないどれか一つの側面のみにかたよった分類になってしまわざるを得ない. したがって, 例えば「Windows 上で動作するエディタ」は 図 1 にあるように, 「MFC アプリケーション」, 「エディタ」という二つのカテゴリに属するものとして分類するように, 非排他的に分類するほうがより現実に沿った分類であると考えられる.

2.1 既存の関連研究

これまでに, ソフトウェアの分類ではなく, ある単一のソフトウェアを何らかの機能的単位へ分割する研究が主に行われている. このような研究として, LSA を用いた手法 [11] のほかに, Self-Organizing-Map を利用する手法 [3], ファイル構成やファイル名に基づく分類 [1], コールグラフなどプログラムの構造を使って分割を行う手法 [10, 4, 12] が提案されている. これらの研究では, 各ソフトウェアの関数やファイルがそれぞれ単一の機能を提供していることを前提とした分類が行われている. しかし, ソフトウェアは多くの機能や側面を持つため, これらの分類方法をそのまま適用してもソフトウェアのある側面しかとらえることができない.

ソフトウェアシステム間の関連を対象にした研究として, 山本らの SMMT[17] が挙げられる. SMMT は二つのソフトウェアシステムの類似度を計測する. SMMT ではコードクローン検出技法 [8] に基づいてソースコード行単位での類似度を測定する. 山本らは SMMT を用いて 4.4BSDLite から派生したソフトウェアである FreeBSD, NetBSD, OpenBSD の各バージョン間の類似度を計測し, これらのソフトウェアを適切に分類できることを示している.

しかし, SMMT での類似度の定義は二つのソフトウェア全ての行のうち, コードクローンを含む行, および全く同一の行が存在する割合としている. そのため, 同じソースから派生したようなソフトウェア間の類似度は適正な値が出力されるが, 全く出自が異なるソフトウェア間では類似度の値が極端に低くなる.

3　提案手法

われわれの提案するソフトウェア分類手法ではソフトウェアが持つそれぞれの側面を抽出するために, プログラムに含まれる変数名や, 関数名などの識別子に着目する. これら識別子は, それらが使われている文脈での動作を表しているものと考える. 例えば window という識別子は何らかのウィンドウを表しており, この識別子が使われているプログラム文の近傍ではなんらかの GUI に関する動作が行われている可能性が高い.

このように識別子はプログラムの機能の一部を表していると考えられる. そこで, 多量の識別子の中から関連の高い識別子群を抜きだせば, それらが一つのカテゴリを表していることが予想される.

関連の高い識別子群を特定するために, 本手法では自然言語の分野で利用されている潜在的意味解析手法 Latent Semantic Analysis を応用する. LSA は純粋に統計学的手法であるにも関わらず, 直接関連した単語がなくとも, 類似した単語を抽出することを可能にする強力な手法である. LSA についての詳細は付録 A を参照のこと.

3.1　概要

本手法の概要を図 2 に示す.

本手法の入力, および出力は以下のとおりである.

入力:　分類対象のソフトウェア $s_1, s_2, \ldots, s_m$ のソースコード

出力:　ソフトウェアのクラスタ集合 $Cs = \{cs_i \mid cs_i \neq S\}$, および各 cs_i に対応するタイトル文字列 $Title(cs_i)$

本手法では, まず各ソフトウェアからトークンの抽出を行い, 共起行列を作成する. (共起行列については付録 A.1 参照) 次に, 分析に不要なトークンを取りのぞき, LSA を適用する. その結果を用いて識別子間の類似度を計測し, その類似度に基づいてクラスタ分析を行う. その結果得られたクラスター一つ一つをカテゴリとして, ソフトウェアの分類を行う.

以下, それぞれのステップについて, その詳細を述べる.

3.2　トークン抽出

トークンとは, 分類対象の文書に含まれる単語のことである. 通常の LSA においては, 文書に含まれる単語を全て利用するが, LSA をプログラムに適用するにあたり, 単語として何を使うか, いくつか選択肢が存在する.

- 識別子

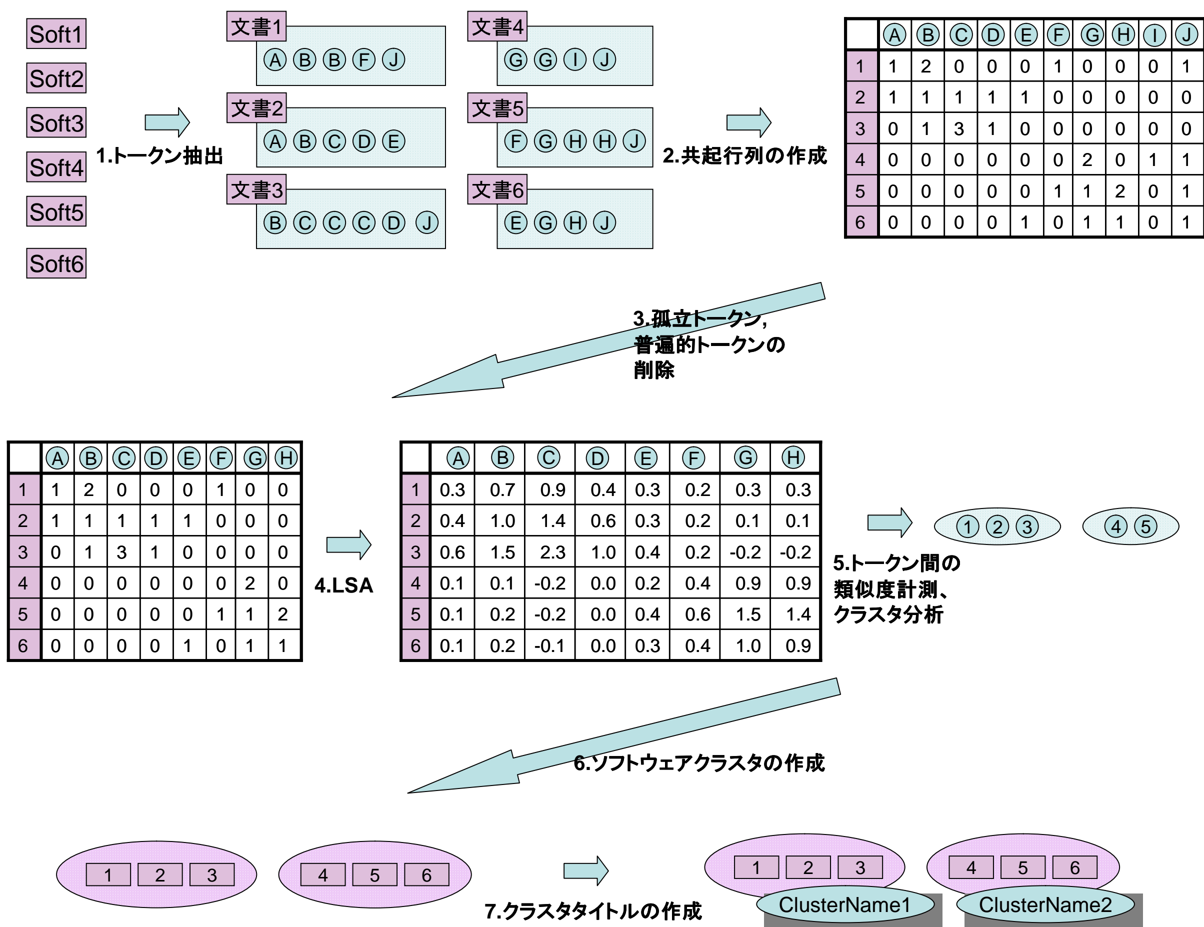

図 2: ソフトウェア分類手法

  - 関数名, もしくはメソッド名
  - 変数名
  - 型名, 定数名などその他の識別子

- 予約語や演算子
- コメントに含まれる単語

これらの中からどの要素をトークンとして利用するかは分析に重大な影響を与えるため慎重に決定しなければならない. 一番単純な方法は識別子, 予約語や演算子, コメントを全てトークンとする方法である. この方法はソースコードを字句解析した結果得られたトークンを全て使うというものである. 利点としては, 実装が容易である点が挙げられる. しかし, 言語によって利用するものが決まっている予約語や演算子は, それによって実装言語以外のソフトウェアの特徴を表すものとは言いがたい. また, 様々なソフトウェアに均等に入ってい

ることが予想される単語は, 分析結果に悪影響を及ぼしこそすれ, 良い影響を与えるとは考えにくく, 計算量の観点から言ってもこのような無駄な単語を用いることは好ましくない. よって, 予約語や演算子は入力トークンから除外する.

ソースコードに含まれるコメント中の単語もトークンの有力な候補の一つである. 一般にコメントはソースコードの動作を比較的高い抽象度で記述していることが多い. しかし, ソフトウェアによってコメントが存在する度合やその品質にはかなりのばらつきがあり, 一律に利用することが難しい. また, オープンソースプログラムなど, ライセンス条項などがコメントとして付随しているソフトウェアもあるため必ずしもコメントがプログラムの動作を表しているとは限らない. そのため, コメントも入力トークンからは除外することとする.

予約語や演算子, コメントに対して, 識別子はソースファイル中に豊富に含まれており LSA の入力として十分な量が得られることが期待できる. また, ソフトウェアの内容をよく表していることが期待できる. したがって, 本手法では識別子をトークンとして用いる.

以下, 各 s_i について, それぞれのソフトウェアに含まれるトークンの集合を $T(s_i)$ とする. また, トークン $t_j \subset T(s_i)$ が出現した回数を $\mathrm{count}(s_i, t_j)$ とする.

3.3 共起行列の作成

ソフトウェア × トークンの共起行列 A を作成する. (共起行列については付録 A 参照) まず, 各ソフトウェアに含まれるトークン全体の集合

$$T = \bigcup_{i=1}^{m} T(s_i)$$

を求める. そして次式によって定義される $m \times n$ 行列 A を作成する. (ただし $m = |S|, n = |T|$)

$$A = [a_{ij}]$$

$$\left(a_{ij} = \begin{cases} \mathrm{count}(s_i, t_j) & \text{if } \ t_j \subset T(s_i) \\ 0 & \text{others} \end{cases} \right)$$

3.4 孤立トークン, 普遍的トークンの削除

トークンのうち, ある単一のソフトウェアにのみ存在するトークン, および $|S|/2$ 個以上のソフトウェアに存在するトークンを取りのぞく. LSA においては単一のソフトウェアにしか含まれないトークンは全く無意味であり, 逆に多数のソフトウェアに含まれるトークンは分析結果に影響がないと考えられるからである.

3.5 LSA

共起行列 A に対して LSA を適用する. まず, LSA を行う前段階として単語頻度と文書頻度に基づいて変換を行う. 詳細は付録 A.2 を参照のこと.

次に共起行列 A に対して特異値分解を行い, 行列 U, S, V を得る. そして各行列の次元を r に落とした後に再び掛け合わせて $\hat{A}$ を得る.

3.6 トークン間の類似度計測, クラスタ分析

LSA の結果得られた行列 $\hat{A}$ から単語ベクトルを抽出し, それらの類似度からクラスタ分析を行う. 単語ベクトルの定義, および類似度の算出方法については, 付録 A.1 参照.

クラスタ分析とは, 複数個の個体を, それら個体間の類似度に基づいていくつかのクラスタに分類する分析手法である. クラスタ分析にはいくつかの分析方法があるが, ここでは全てが独立したクラスタという状態から始めて, もっとも類似度の高いクラスタから順次結合していく方式を採用する. この手法は以下のアルゴリムで表される.

1. 初期状態 $C = \{c_j \mid 1 \leq j \leq |T|\}, c_j = \{t_j\}$
2. $\forall i \forall j$ similarity$(c_p, c_q) \geq$ similarity(c_i, c_j) なる p, q を探索
3. $c_p = c_p \cup c_q$, C から c_p を取りのぞく
4. これを $|C| = 1$ となる, もしくは終了条件 $\forall i \forall j\ s \geq$ similarity(c_i, c_j), (s は与えられた閾値) を満たすまで繰りかえす.

なお, similarity(c_i, c_j) の定義方法も複数の方法が存在する. そのうちいくつかの方法は類似度の計算方法がユークリッド距離でなくてはならない. 本手法では類似度の計算方法は角度に拠っているためこれらの手法を利用することはできない. そのような仮定を置かない similarity(c_i, c_j) の定義法としてよく知られているものとしていくつかの方法があるが,

最短距離法 二つのクラスタ間のうち, 最も距離の短い (類似度の高い) 要素間の距離をクラスタの距離とする.

$$\text{similarity}(c_i, c_j) = \cos(t_p, t_q)$$

ただし,

$$\forall i \forall j\ \ \cos(t_p, t_q) \geq \cos(t_i, t_j), t_p \in c_i, t_i \in c_i, t_q \in c_j, t_j \in c_j$$

最長距離法 二つのクラスタ間のうち, 最も距離の長い (類似度の低い) 要素間の距離をクラスタの距離とする.

$$\text{similarity}(c_i, c_j) = \cos(t_p, t_q)$$

ただし,

$$\forall i \forall j \ \cos(t_p, t_q) \leq \cos(t_i, t_j), t_p \in c_i, t_i \in c_i, t_q \in c_j, t_j \in c_j$$

群平均法 二つのクラスタに属する要素間の距離の平均を, そのクラスタの距離とする.

$$\text{similarity}(c_i, c_j) = \frac{\displaystyle\sum_{t_i \in c_i} \sum_{t_j \in c_j} \cos(t_i, t_j)}{|c_i| + |c_j|}$$

最短距離法は最も単純な方法であり, その実現も容易である. しかし, 結果として出力されるクラスタに鎖が出現する可能性が高いことが知られている. 鎖とは単一のクラスタが肥大している状態を指す. 最短距離法では, クラスタ数が多くなればなるほど他クラスタとの類似度が高くなっていく. こうして有意なクラスタ群ではなく, 巨大単一クラスタと矮小ないくつかのクラスタ, というように分割されてしまう.

群平均法は最短距離法のように鎖ができるという問題点もなく, 均等なクラスタができることが期待できる. しかし, 距離を計算するために平均を逐一計算しなおさなければならないため莫大な計算量が必要である. また, 全てのクラタ間の類似度行列を保存しておかなければならないため, クラスタの数が多くなると巨大な行列を保持しなければならない.

最長距離法は, 最短距離法と比較すると計算量が増えるが, 類似度行列を保持する必要はないため, 群平均法よりは算出にかかるコストが小さい. また, 出力されるクラスタも群平均法と遜色ない性質のクラスタが得られる. そこでをそれらクラスタの距離とする.

なお, この段階ではトークン数が多すぎて一度にクラスタ分析することが困難である. また, トークンの中には, いくつかのトークンと高い類似度を持って明確なクラスタを構成する分類に有用なトークンもあれば, 逆にどのトークンともそれほど高い類似度を持たず, 明確な分類の妨げとなるトークンも数多く存在する. このようなトークンを含んだまま分類を押し進めても分析結果に悪影響を及ぼす. そこで, 二段階に分けてクラスタ分析を行うことにする. 一度目のクラスタ分析において, 明確なクラスタを形成するトークンを抽出し, 二度目のクラスタ分析で, トークンのクラスタを算出する.

トークン集合 T に対して終了条件の閾値を p_{s1} として一度目のクラスタ分析した結果得られるクラスタ集合 C は次の式で表される.

$$C = \{c_1, \ldots, c_{p_{t1}} \mid \forall i \forall j c_i \cap c_j = \emptyset, c_i \subset T\}$$

このクラスタ分析で, 一定数 p_t 以上のトークンを持つクラスタに含まれるトークンのみをこの先の分析に利用する. これを有意トークン集合 T' と呼ぶ. T' は以下の式で表される.

$$\begin{aligned} T' &= \bigcup c_i \\ C &= \{c_i \mid |c_i| > p_t\} \end{aligned}$$

次に有意トークン集合 T' のみを対象として再びクラスタ分析を行う. このように, 不要なトークンを排除してからクラスタ分析を行うことによって, より明示的なクラスタを得ることができる.

有意トークン集合 T' に対して終了条件の閾値を p_{s2} としてクラスタ分析した結果のクラスタ集合を C' とすると

$$C' = \{c'_1, \ldots, c'_{p_{t2}} \mid \forall i \forall j c'_i \cap c'_j = \emptyset, c'_i \subset T'\}$$

このようにして得られたクラスタ c'_i は, それぞれソフトウェアに含まれるある種の側面を表しているものといえる.

3.7 ソフトウェアクラスタの作成

クラスタを構成するトークンから, そのトークンを含むソフトウェアの和集合を一つのクラスタとする. 各 c'_i に対応するソフトウェアクラスタ cs_i は以下の式によって定義される.

$$cs_i = \{s_{cs_1}, \ldots, s_{cs_n} \mid \forall j T(s_{cs_j}) \cap c'_i \neq \emptyset\}$$

3.8 クラスタタイトルの作成

以上の手法により, クラスタリングされたソフトウェアの集合 $cs_1, \ldots, cs_n$ が得られる. しかし, このソフトウェア集合そのものだけではこのクラスタにどのようなソフトウェアが集まっているのかを理解することは極めて困難である. そこで, このクラスタに入っているソフトウェア群を代表する文字列, すなわちクラスタタイトル $Title(s_i)$ の抽出を行う.

クラスタ内ソフトウェアベクトル全体を表す $\vec{cs}$ を以下の式のように定義する.

$$\vec{cs} = \sum_{i=1}^{|cs|} \vec{s_{cs_i}}$$

上述のとおり, ソフトウェアベクトル $\vec{s} = (w_1, \ldots, w_n)$ は, 語彙集合 W 内の単語の頻度を並べたものである. そこで, $\vec{cs}$ から次のような $m_1, \ldots, m_N$ を求める.

$$\forall i \forall j \; m_i \neq j \rightarrow \vec{cs}(m_i) \geq \vec{cs}(j)$$

$m_1, \ldots, m_N$ は直感的には $\vec{s}$ から値の大きい要素の上位 N 個を表わしている. これらのトークンは, ソフトウェアクラスタ内のソフトウェアに強い影響を及ぼしているトークン, すなわちこれらソフトウェアクラスタを特徴づけるトークンである. したがって, $w_{m_1}, \ldots, w_{m_N}$ をこのクラスタを表すクラスタタイトルとする.

以上の手順でソフトウェアのクラスタ集合 Cs, および各 cs に対応するタイトル文字列 $Title(cs)$ を得る.

4　ソフトウェア分類システム

我々は, 前節で提案した手法に基づくソフトウェア自動分類システムの試作を行った. 本システムは以下のプログラム群で構成される.

- トークン切り出しを行うプログラム
- 共起行列作成プログラム
- 孤立トークン, 普遍トークンの削除を行うプログラム
- LSA を行うプログラム
- 類似度計測, クラスタリングプログラム
- トークンクラスタからソフトウェアクラスタを求めるプログラム
- ソフトウェアクラスタからタイトルクラスタを求めるプログラム
- クラスタファイル群整形プログラム
- 上記プログラムを呼びだし, 分類を行うプログラム

トークン切りだしプログラムは現在のところ C 言語にのみ対応している. しかし, 手法そのものにはプログラム言語とは独立に定められているため, トークン切りだし部さえ作成すれば, Java や C++ 等といった他の言語に対応することが可能である.

LSA に必要な特異値分解を行う部分には SVDPACKC[2] を利用した. SVDPACKC は疎行列を高速かつ効率的に特異値分解するライブラリである. 提案手法で作成したプログラムとトークンの行列の場合, その中身はほとんど 0 であるため, SVDPACKC を用いることにより効率的に演算を行うことが可能である.

実装言語は, トークン切りだし部と SVD が C 言語, 司令部が sh スクリプトであり, その他のプログラムはすべて Perl で記述されている. ソースコードの総計は約 4000 行である. ただし, この中には yacc が自動生成した部分, SVDPACKC の部分は含んでいない.

これらのプログラム間では以下の 3 つのデータ形式を利用する.

- list 形式

  トークンと, その数が格納される.

- mat 形式

  行列を格納するファイル形式.

- cl 形式

  さまざまなクラスタを表す形式.

以下, それぞれのプログラム間で使われるデータ形式と, 各プログラムの概略を述べる.

4.1 ファイル形式

4.1.1 list 形式

トークンとその数が格納されるファイル形式. 拡張子は .list
各行は次の要素で構成される.

(トークンの数) (トークン)

例:

```
15 a
35 a_sq
 3 abs
 3 allocate_time
 9 allocated_time
 5 allow_more_time
37 alpha
 8 argc
```

4.1.2 mat 形式

行列を表す. 拡張子は .mat
mat 形式の書式は以下の通り.

1 行目: (行数) (列数)

2 行目以降: (数字)

例:

```
12 9
1 0 0 1 0 0 0 0 0
1 0 1 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0
0 1 1 0 1 0 0 0 0
0 1 1 2 0 0 0 0 0
0 1 0 0 1 0 0 0 0
0 1 0 0 1 0 0 0 0
0 0 1 1 0 0 0 0 0
0 1 0 0 0 0 0 0 1
0 0 0 0 0 1 1 1 0
0 0 0 0 0 0 1 1 1
0 0 0 0 0 0 0 1 1
```

4.1.3 cl 形式

トークンやソフトウェアのクラスタを表す
一行が一つの要素. 改行のみの行が区切り.
例:

- “1”
- “0, 4”
- “6, 8”

という 3 つのクラスタがある場合の cl ファイルは以下のようになる.

```
1

0
4

6
8
```

4.2 トークン切り出しプログラム

入力: ソースコードのファイル名のリスト

出力: ソースコード中のトークン, およびその数 (list 形式)

ソフトウェアのトークンを切りだし, トークンおよびその頻度を list 形式で出力する. 対応言語は C 言語のみである.

字句解析ライブラリとして lex を利用している.

4.3 共起行列作成プログラム

入力: 複数個のリストファイル

出力: リストファイルを一つのソフトウェアとみて, ソフトウェア $times$ トークンの共起行列 (mat 形式)

一つのソフトウェアに含まれるトークンとその数を list 形式として受けとり, ソフトウェア $times$ トークンの共起行列を作成する.

出力される行列は, 各行がソフトウェア, 各列がトークンを表す. すなわち, 入力に与えられた list ファイルの数が n 個, すべての list ファイルのトークンの和集合をとった結果 m 個のトークンが存在したとすると, 出力される行列のサイズは $n \times m$ となる.

4.4 孤立トークン, 普遍トークンの削除を行うプログラム

入力: ソフトウェア $\times$ トークンの共起行列 (mat 形式)

出力: 孤立トークン, 普遍トークンを取りのぞいた共起行列 (mat 形式)

共起行列から, 単一のソフトウェアのみに含まれる孤立トークンや, 半数以上のソフトウェアに含まれる普遍トークンの削除を行う. 結果として出力される行列は, 孤立トークンと普遍トークンの合計数だけ列数が減る.

4.5 LSA を行うプログラム

入力: 共起行列 (mat 形式)

出力: LSA が適用された行列 (mat 形式)

LSA は tf-idf 変換と特異値分解の 2 つのステップから構成される. このプログラムは入力として与えられた行列に対して tf-idf 変換を行い, 特異値分解を行うプログラム SVDPACKC を呼びだす. SVDPACKC は入出力を mat 形式とは異なるフォーマットでやりとりするため, 本プログラムは形式変換を行う.

SVDPACKC は, 疎行列の変換のためのアルゴリズムとして, 以下の 4 つのアルゴリズムを実装している.

- Subspace Iteration
- Trace minimization Method
- Single-Vector Lanczos Method
- Block Lanczos Method

本研究では, 4 番目のアルゴリズム Block Lanczos Method を利用する. これは入力可能な行列の制限が一番ゆるやかなのが Block Lanczos Method だからである. Block Lanczos Method では, 列数が 5200 まで, 非 0 要素の数が 800,000 個までの行列を演算することが可能である. なお, SVDPACKC には全てのアルゴリズムに列数制限が設けられている. そのため, SVDPACKC にデータを渡す際には一度行列転置を行い, 演算終了後に再び転置を行っている.

4.6 類似度計測, クラスタリングプログラム

入力: 共起行列 (mat 形式)

出力: トークンクラスタ (cl 形式)

LSA の結果を元に単語間, すなわち列ベクトル間の類似度を計測し, クラスタリングを行う. クラスタ間の距離の定義は, 最短距離法と最長距離法の 2 つをサポートする. 本プログラムは, クラスタリングを行った結果を cl 形式で出力する. それぞれのトークンは共起行列の列番号で表す.

通常のクラスタ分析プログラムでは分類対象となる要素間の全ての類似度を保管し, クラスタが生成されるにつれて, 適宜, 類似度行列を更新する方式をとっている. このため, 空間計算量は $O(n^2)$ となる.

しかし, 本手法ではトークンを分類対象としており, 分類対象の数は数万程度である. そのため, 通常の手法では絶対的にメモリ空間が不足してしまう. 仮に要素間の類似度一つに 8

バイトを利用し, トークンが 4 万個存在したとすると, 要素すべての類似度を保存するのに必要なメモリは約 6.4GB に達する.

そこで本プログラムでは, 類似度を全てメモリ上に展開することなく, 類似度の高いトークン組を取りだせるようにする. そして, その結果からクラスタリングを行う. トークン組を類似度の順に取りだす手法の概要を以下に示す.

1. 準備段階
   - 1.a 類似度計算の結果が, ある一定サイズに達するまで類似度を計算する.
   - 1.b ある一定サイズに達したら, それまでに計算した類似度の結果をソートする.
   - 1.c ソートした結果をファイルに書きだす.
2. 類似度並びかえ段階
   - 2.a 準備段階で得られたファイルを全て開く.
   - 2.b それぞれのファイルについて, 各ファイルの先頭, すなわち類似度が最も大きいトークン組を取りだす.
   - 2.c 取りだしたトークン組の中で類似度最大のものを, 最も類似度最大のものとしてクラスタリングアルゴリズムに渡す.
   - 2.d さきほど取りだした要素が入っていたファイルから一つ新しい要素を抜きだす.
   - 2.e 2.c に戻る.

4.7 ソフトウェアクラスタ生成プログラム

入力: トークンクラスタ (cl 形式), 共起行列 (mat 形式)

出力: ソフトウェアクラスタ (cl 形式)

入力のトークンクラスタファイルに含まれるトークンから, それに対応するソフトウェアを共起行列から求める.

4.8 クラスタタイトル生成プログラム

入力: ソフトウェアクラスタ (cl 形式), 共起行列 (mat 形式)

出力: クラスタタイトル (cl 形式)

各ソフトウェアクラスタを読みこみ, 対応する列ベクトルを足しあわせ, 値の大きい要素上位 n 個を出力する.

4.9 クラスタファイル群整形プログラム

入力: 複数のクラスタファイル (cl 形式)

出力: csv ファイル

複数のクラスタファイルの出力を統合し, 一つの表として出力する. 入力されたクラスタファイルを順に $\mathrm{CF}_1, \mathrm{CF}_2, \ldots, \mathrm{CF}_N$ とすると

CF_1 1 番クラスタ	CF_2 1 番クラスタ	$\cdots$	CF_N 1 番クラスタ
CF_1 2 番クラスタ	CF_2 2 番クラスタ	$\cdots$	CF_N 2 番クラスタ
$\vdots$	$\vdots$	$\ddots$	$\vdots$
CF_1 M 番クラスタ	CF_2 M 番クラスタ	$\cdots$	CF_N M 番クラスタ

このような表を csv 形式で出力する.

4.10 統合プログラム

入力: それぞれのソースファイルが入っているディレクトリ名

出力: クラスタタイトル, ソフトウェアクラスタ, トークンクラスタを合成した csv ファイル.

これまでに述べたツールを組みあわせてソフトウェア分類を行う. 具体的には以下の手順を実行する.

1. それぞれのソフトウェアが入っているディレクトリごとに「トークン切りだしプログラム」を実行. それぞれのソフトウェアに含まれるトークンとその数を得る.
2. 各ソフトウェアのトークン頻度リストを「共起行列作成プログラム」に渡して共起行列を得る.
3. 得られた共起行列を「孤立トークン, 普遍トークンの削除を行うプログラム」に入力し, 孤立トークン, 普遍トークンを除いた共起行列を得る.
4. 「LSA を行うプログラム」を利用して, 孤立トークン, 普遍トークンを除いた共起行列に対して LSA を適用する.
5. LSA の結果を「類似度計測, クラスタリングプログラム」に入力しトークンクラスタを得る.
6. 「トークンクラスタからソフトウェアクラスタを求めるプログラム」を実行して, ソフトウェアクラスタを得る.

7. 「ソフトウェアクラタからタイトルクラスタを求めるプログラム」を実行して, 各ソフトウェアクラスタのクラスタタイトルを得る.
8. クラスタタイトルとソフトウェアクラスタを「クラスタファイル群整形プログラム」を利用して csv 形式に整形する.

カテゴリ	ソフトウェア
boardgame	Sjeng-10.0, bingo-cards, btechmux-1.4.3, cinag-1.1.4, faile_1_4_4, gbatnav-1.0.4, gchch-1.2.1, icsDrone, libgmonopd-0.3.0, netships-1.3.1, nettoe-1.1.0, nngs-1.1.14, ttt-0.10.0
compilers	clisp-2.30, csl-4.3.0, freewrapsrc53, gbdk, gprolog-1.2.3, gsoap2, jcom223, nasm-0.98.35, pfe-0.32.56, sdcc
database	centrallix, emdros-1.1.4, firebird-1.0.0.796, gtm_V43001A, leap-1.2.6, mysql-3.23.49, postgresql-7.2.1
editor	gedit-1.120.0, gmas-1.1.0, gnotepad+-1.3.3, molasses-1.1.0, peacock-0.4
videoconversion	dv2jpg-1.1, libcu30-1.0, mjpgTools, mpegsplit-1.1.1
xterm	R6.3, R6.4

表 1: 実験用ソフトウェアの一覧

5 実験

本節では提案手法を用いて実際にソフトウェアの分類を行い, 既存の分類と比較して適正な分類が可能かどうか, 利用しているライブラリなど既存の分類では考慮されていない分類について分類ができているかどうかの確認を行った.

5.1 実験方法

実験対象として分類を行うソフトウェアの収集を行った. SourceForge から無作為に 5 つのカテゴリを選出し, そのカテゴリに含まれる C 言語のプログラムを実験対象とした. 表 1 にその一覧を示す.

5.2 結果

提案手法を適用して, 分類を行った結果を表 2 に示す. 一行が一つのクラスタを表しており, 左から, クラスタタイトル, クラスタに含まれるソフトウェア, クラスタに含まれるトークンの数となっている.

No.	クラスタタイトル	ソフトウェア	トークン数
1	AOP, emitcode, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT, pic14_emitcode, iCode, etype	compilers/gbdk, compilers/sdcc	8597
2	CASE_IGNORE, CASE_GROUND_STATE, screen, CASE_PRINT, CASE_BYP_STATE, Widget, TScreen, CASE_IGNORE_STATE, CASE_PLT_VEC, CASE_PT_POINT	xterm/R6.3, xterm/R6.4	2160
3	YY_BREAK, yyvsp, yyval, DATA, yy_current_buffer, tuple, yy_current_state, yy_c_buf_p, yy_cp, uint32	compilers/gbdk, database/mysql-3.23.49, database/postgresql-7.2.1	223
4	AVI, cinfo, OUTLONG, avi_t, AVI_errno, hdrl_data, OUT4CC, nhb, ERR_EXIT, str2ulong	videoconversion/dv2jpg-1.1, videoconversion/libcu30-1.0, videoconversion/mjpgTools	177
5	domainname, msgid1, binding, msgid2, domainbinding, pexp, __builtin_expect, transmem_list, codeset, codesetp	boardgame/gbatnav-1.0.4, boardgame/gchch-1.2.1	165
6	board, num_moves, ply, pawn_file, npiece, pawns, moves, white_to_move, move_s, promoted	boardgame/Sjeng-10.0, boardgame/cinag-1.1.4, boardgame/faile_1_4_4	154
7	xdrs, blob, DB, UCHAR, XDR, mutex, key_length, logp, page_no, bdb	database/firebird-1.0.0.796, database/mysql-3.23.49	118
8	domainname, N_, binding, gchar, GtkWidget, PARAMS, codeset, gpointer, loaded_l10nfile, argz	boardgame/gbatnav-1.0.4, boardgame/gchch-1.2.1, editor/gnotepad+-1.3.3, editor/peacock-0.4	118

No.	クラスタタイトル	ソフトウェア	トークン数
9	GtkWidget, gchar, gpointer, gint, widget, gtk_widget_show, N_, g_free, dialog, g_return_if_fail	boardgame/gbatnav-1.0.4, editor/gedit-1.120.0, editor/gmas-1.1.0, editor/gnotepad+-1.3.3, editor/peacock-0.4	104
10	AOP, emitcode, esp, IC_RESULT, IC_LEFT, obstack, aop, mov, aopGet, IC_RIGHT	compilers/clisp-2.30, compilers/gbdk, compilers/sdcc	100
11	tuple, uint32, plan, int32, lsn, elm, rec, interp, TCL_ERROR, finfo	database/mysql-3.23.49, database/postgresql-7.2.1	79
12	xdrs, blob, DB, UCHAR, XDR, mutex, key_length, logp, page_no, bdb	database/firebird-1.0.0.796, database/mysql-3.23.49	73
13	UCHAR, relation, stmt, trigger, yyvsp, yyval, t_data, plan, dbname, USHORT	database/firebird-1.0.0.796, database/postgresql-7.2.1	68
14	fout, interp, TCL_ERROR, typ, YY_RULE_SETUP, List, DATA, Tcl_Interp, id, YY_BREAK	compilers/freewrapsrc53, compilers/gbdk, compilers/gsoap2, database/postgresql-7.2.1	50
15	GtkWidget, gchar, gpointer, dlg, gint, g_free, gtk_widget_show, gtk, GList, GTK_BOX	editor/gedit-1.120.0, editor/gmas-1.1.0, editor/gnotepad+-1.3.3	46
16	UCHAR, relation, stmt, trigger, yyvsp, yyval, t_data, plan, dbname, USHORT	database/firebird-1.0.0.796, database/postgresql-7.2.1	43
17	AOP, emitcode, mfp, ic, uchar, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT	compilers/gbdk, compilers/sdcc, database/mysql-3.23.49	36
18	adr, FX, word, stm, ED, xt, REF, prop, term, FP	compilers/gprolog-1.2.3, compilers/pfe-0.32.56	35

No.	クラスタタイトル	ソフトウェア	トークン数
19	AOP, emitcode, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT, pic14_emitcode, iCode, etype	compilers/gbdk, compilers/sdcc, database/firebird-1.0.0.796	31
20	dyn, FPRINTF, process_id, p_offset, ctl, rab, que, io_ptr, prior, PRINTF	database/firebird-1.0.0.796, database/gtm_V43001A	29
21	dyn, FPRINTF, process_id, p_offset, ctl, rab, que, io_ptr, prior, PRINTF	database/firebird-1.0.0.796, database/gtm_V43001A	27
22	regparse, dbp, mech, reginput, flagp, NOTHING, tuple, db, __P, regnode	boardgame/btechmux-1.4.3, database/leap-1.2.6, database/mysql-3.23.49	26
23	rectype, argp, rec, fileid, save_errno, data_len, qp, argpp, int4, dbp	database/gtm_V43001A, database/mysql-3.23.49	26
24	AOP, emitcode, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT, pic14_emitcode, iCode, etype	compilers/gbdk, compilers/sdcc, videoconversion/mjpgTools	26
25	jobject, JNIEnv, JNICALL, JNIEXPORT, jint, jstring, interp, TCL_ERROR, objv, TCL_OK	compilers/freewrapsrc53, compilers/jcom223, compilers/pfe-0.32.56, database/mysql-3.23.49	24
26	entrypoint, USHORT, TEXT, yyvsp, raddr, R, UCHAR, yyval, blob, REQ	compilers/clisp-2.30, database/firebird-1.0.0.796	17
27	int32_t, dbp, cinfo, net, unpack, argp, sinfo, cur1, purpose, mysql	database/mysql-3.23.49, videoconversion/mjpgTools	17
28	AOP, emitcode, mfp, ic, uchar, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT	compilers/gbdk, compilers/sdcc, database/mysql-3.23.49	16

No.	クラスタタイトル	ソフトウェア	トークン数
29	USHORT, UCHAR, blob, REQ, NULL_PTR, hIcon, SCHAR, interp, wndclass, bdb	compilers/freewrapsrc53, database/firebird-1.0.0.796	16
30	optind, nextchar, __P, optstring, last_nonopt, option_index, uchar, optarg, pfound, dbp	boardgame/ttt-0.10.0, compilers/clisp-2.30, database/mysql-3.23.49	15
31	int4, ctl, tn, rec, semid, blkno, ti, oprtype, save_errno, AH	database/gtm_V43001A, database/postgresql-7.2.1	14
32	notify, mech, PyObject, fargs, Node, Name, pset, zone, tprintf, NOTHING	boardgame/btechmux-1.4.3, database/postgresql-7.2.1	11
33	interp, notify, dbp, tuple, mech, PyObject, uint32, plan, int32, buff	boardgame/btechmux-1.4.3, database/mysql-3.23.49, database/postgresql-7.2.1	10
34	adr, stm, AOP, emitcode, operands, ASSERT, IC_RESULT, pred, lg, REF	compilers/gprolog-1.2.3, compilers/sdcc	9
35	yyvsp, yyn, PARAMS, codeset, domainname, msgid1, binding, msgid2, yylsp, domainbinding	boardgame/gbatnav-1.0.4, boardgame/gchch-1.2.1, compilers/clisp-2.30	9
36	ERREXIT, picture, pool_id, USHORT, get_buffer, output_buf, cinfo, xxx, UCHAR, streams	database/firebird-1.0.0.796, videoconversion/mjpgTools	9
37	REF, dyn, USHORT, vec, path_name, clause, STATUS, E, UCHAR, CSB	compilers/gprolog-1.2.3, database/firebird-1.0.0.796	8
38	AOP, emitcode, pfile, ic, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT, pic14_emitcode	compilers/gbdk, compilers/sdcc, database/postgresql-7.2.1	7
39	ic, ply, npiece, score, AOP, pawn_file, uchar, bking_loc, wking_loc, emitcode	boardgame/Sjeng-10.0, compilers/gbdk	7

No.	クラスタタイトル	ソフトウェア	トークン数
40	clause, cinfo, pred, ci, Group, Np, word, X, A, tmp4	compilers/gprolog-1.2.3, database/postgresql-7.2.1, videoconversion/mjpgTools	6

表 2: 41 ソフトウェアの全分類結果

全 40 個のクラスタのうち, 同一ジャンル内のクラスタとなっているのが 18 クラスタを占めた. また, 既存の分類では同一ではないものの, 同一のライブラリを利用している, 同一のアーキテクチャを持っているなどの強い関連を持つクラスタが 8 存在した. 特に結び付きの強い上位 20 クラスタに限定すると, 同一ジャンル, もしくは関連を持つクラスタは 17 クラスタに及ぶ.

既存の分類では同一ではないものの, 強い関連を持つクラスタの例としては, 第 3 クラスタや第 8, 9 クラスタが挙げられる. 前者は YACC を利用したソフトウェアが, 後者には gtk を利用したソフトウェアがそれぞれ分類されている. この他にも

第 22 クラスタ 正規表現ライブラリを利用しているソフトウェア群

第 25 クラスタ JNI を実装しているソフトウェア群

第 30 クラスタ getopt 関数を利用しているソフトウェア群

第 32 クラスタ Python/C を実装しているソフトウェア群

第 35 クラスタ yacc を利用しているソフトウェア群

以上のようなクラスタを得ることができた.

また, 41 アプリケーションのうち, いずれのカテゴリにも分類されなかったソフトウェアが 12 個存在した.

5.3 考察

既存の分類と比較した場合, 得られたクラスタの多数は既存の分類に従っているものの, 既存の分類を十分にカバーできているとは言い難い. これは, どこにも分類されなかったソフトウェアが多いことがその最大の原因と思われる. 本手法ではトークンの出現頻度に基づいて, 統計的処理を用いているため, トークン数の少ないものが, どこにも分類されずその他となってしまう傾向が高い.

既存の分類では考慮されていない分類については, 本手法を用いることで新たな分類を得られることを確認した. このような分類として, GTK や yacc など共通のライブラリを利用したソフトウェア群や, JNI を実装しているソフトウェア群など共通のアーキテクチャに従った分類がある. 特に本手法は分類を行うにあたって前提知識等の入力は一切不要であり, 今後新しいライブラリなどが現れても自動的にこれに追随できることが期待できる.

クラスタタイトルについては, 第 1 クラスタを始め, アプリケーションドメインが同一のものは必ずしもわかりやすいとは言えないものも存在する. しかし, 例えば第 4, 第 6 クラス

タなどは,「AVI」や「board, ply」など非常に分かりやすいタイトルがついている. その他, 使用ライブラリが同一のものなどは分かりやすいタイトルがつく傾向にある.

6　まとめ

本研究では複数個のソフトウェアシステムから, 複数個の分類を見つけだし自動的に分類する手法の提案を行った. 本手法を用いることにより対象ソフトウェアに対する詳細な知識がなくとも自動的に分類が行えることを示した. 特にライブラリやアーキテクチャの検出を自動的に行えるため, 分類対象に新たなライブラリを利用するソフトウェア群が追加されても, 再分類を行うだけで, これらのソフトウェア群を分類することができる.

今後の課題としては, パラメータのよりよい決定方法の模索や, より理解しやすりクラスタタイトルの決定方法の考案, より大規模な実験, そして実行速度およびスケーラビリティの向上が挙げられる.

謝辞

本論文を作成するにあたり，常に適切な御指導を賜わり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 井上 克郎 教授に心より深く感謝致します.

本論文の作成にあたり，適切な御指導および御助言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 楠本 真二 助教授に深く感謝致します.

本論文の作成にあたり，適切な御指導および御助言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 松下 誠 助手に深く感謝致します.

本論文の作成にあたり，御指導や御助言を頂きました Zee Source 社 Pankaj K. Garg 氏に深く感謝致します.

最後に，その他様々な御指導，御助言等を頂い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井上研究室の皆様に深く感謝致します.

参考文献

[1] Nicolas Anquetil and Timothy Lethbridge. Extracting concepts from file names; a new file clustering criterion. In *International Conference on Software Engineering,(ICSE'98)*, pp. 84–93, Apr 1998.

[2] M. W. Berry, T. Do, G. W. O'Brien, V. Krishna, and S. Varadhan. SVDPACKC (Version 1.0) User's Guide. Technical Report CS-93-194, University of Tennessee, Knoxville, TN, April 1993.

[3] Alvin Chan and Tim Spracklen. Discoverying common features in software code using self-organising maps. In *International Symposium on Computational Intelligence (ISCI'2000)*, Kosice, Slovakia, August 2000.

[4] Kunrong Chen and Václav Rajlich. Case study of feature location using dependency graph. In *8th International Workshop on Program Comprehension (IWPC'00)*, pp. 231–239, Limerick, Ireland, June 2000.

[5] J. Dinkelacker and P. Garg. "Corporate Source: Applying Open Source concepts to a corporate environment (Position Paper)". In *Proceedings of the 1st ICSE workshop on Open Source software engineering*, Toronto, Canada, 2001.

[6] J. Dinkelacker, P. Garg, D. Nelson, and R. Miller. Progressive Open Source. In *Proceedings of the International Conference on Software Engineering*, Orlando, Florida, 2002.

[7] D Harman. An experimental study of factors important in document ranking. In *Proceedings of ACM conference on Research and development in information retrieval*, pp. 186–193, Pisa, Italy, September 1986.

[8] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. "CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code". *IEEE Transactions. Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[9] T. K. Landauer and S. T. Dumais. A solution to plato's problem: The latent semantic analysis theory of the acquisition, induction, and representation of knowledge. *Psychological Review*, Vol. 104, No. 2, pp. 211–240, 1997.

[10] G. A. Di Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini. Comprehending web applications by a clustering based approach. In *Proc. of 10th International Workshop on Program Comprehension(IWPC'02)*, pp. 261–270, Paris, France, June 2002.

[11] J. I. Maletic and A. Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'00)*, pp. 46–53, November 2000.

[12] 椴山嘉人, 山本晋一郎, 阿草清滋. Fungram に基づくプログラムパターンとその応用. 電子情報通信学会ソフトウェアサイエンス研究会, Vol. Vol.97, No. No.29, pp. pp.31–38, 1997/9.

[13] Open Source Development Lab. `http://www.osdllab.org/`.

[14] SourceCast. `http://www.collab.net/products/sourcecast/`.

[15] SOURCEFORGE.net. `http://sourceforge.net`.

[16] K. Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, Vol. 28, No. 1, pp. 11–21, 1972.

[17] 山本哲男, 松下誠, 神谷年洋, 井上克郎. ソフトウェアシステムの類似度とその計測ツール smmt. 電子情報通信学会論文誌 D-I, Vol. J85 − D − I, No. No.6, pp. 503–511, 2002 年 6 月.

付録

A. Latent Semantic Analysis

A.1 ベクトル空間モデル

A.2 tf, idf

A.3 特異値分解

A Latent Semantic Analysis

ここでは, 提案手法が利用している LSA, および LSA が前提としているベクトル空間モデルについて, その概要を述べる.

A.1 ベクトル空間モデル

一般に文書に含まれる単語間の関連や類似度を計算し, 統計学的にある種の傾向を見出すために, それぞれの単語を何らかの数学的モデル上に変換することが必要となる. ベクトル空間モデルでは, 対象となる文書内に含まれる単語とその頻度を元にして行列を作成し, その行列から単語や文書に対応するベクトルを定義する.

分類対象となる文書の集合 $D = \{d_1, \ldots, d_m\}$ として, d_i に含まれる単語の集合を $W(d_i)$ とする. このとき分類対象全体の語彙 W は以下の式で表される.

$$W = \bigcup_{i=1}^{m} W(d_i)$$

そして, c_{ij} を文書 d_i に単語 $w_j \in W$ が出現した回数として, 文書 d_i に対応する文書ベクトル $\vec{d_i}$ を次の式で定義する. (ただし $|W| = n$)

$$\vec{d_i} = (c_{i1}, c_{i2}, \ldots, c_{in})$$

このベクトルを全ての文書について計算すると, 以下のような $m \times n$ の行列 A が得られる.

$$A = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{bmatrix}$$

このようにして得られた行列 A を共起行列と呼ぶ.

このとき, 行列 A の第 j 列に着目すると, 単語 w_j が各文書に含まれている頻度を並べたベクトルとなる. これを単語 w_j の単語ベクトル $\vec{w_j}$ とする.

このような方法で文書と単語のモデル化を行う. 上記で定義した文書ベクトル $\vec{d}$ や単語ベクトル $\vec{w}$ を用いて文書同士, もしくは単語同士の類似度を計算することができる. ベクトルからの類似度の計算方法としては, 一般に $\cos\theta$ の値が用いられる. 二つのベクトル $\vec{a} = (a_1, a_2, \ldots, a_n)$ と $\vec{b} = (b_1, b_2, \ldots, b_n)$ 間の $\cos\theta$ は以下の式によって求められる.

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}$$

$$= \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2}\sqrt{\sum_{i=1}^{n} b_i^2}}$$

角度が 0 に近い, すなわち $\cos\theta$ が 1 に近いほど類似度が高く, 逆に -1 に近いほど類似度が低い.

A.2 tf, idf

類似度算出を行う際には, 共起行列 A の各要素として単語の出現頻度そのものを利用するかわりに, 単語頻度 (tf:Terminal Frequency) の正規化を行ったり, ある単語が文書空間全体の内でその単語がどれだけ出現したか (df: document frequency) を考慮した変換を行うことも一般的に行われる. df は単語の普遍性を表わしており, df が低いほど, その単語は特徴的であると言える. そのため, df の逆数, すなわち idf を tf に掛けた値 tf $*$ idf を共起行列の値とする.

tf, idf として様々な式が提案されているが, 本研究では tf として Harman[7], idf として Sparck Jones[16] で提案されている式を利用する.

$$tf_{ij} = \frac{\log_2(a_{ij}+1)}{\log_2(|T(s_i)|)}$$

$$idf_j = \log_2\frac{|D|}{|D'|} + 1 \text{ (ただし } D' = \{s|count(s,t_j) > 0\})$$

A.3 特異値分解

ベクトル空間モデルにおいて, 分類対象の文書やそこに含まれる単語をベクトルに変換する手法はすでに述べた. しかし, 上述の手法で定義されるベクトル空間はかなり疎 (sparse) であり, そのままの状態では必ずしも適正な類似度が算出できるとは限らない.

例えば文書間の類似度を計算した場合, それらの文書に全く同じ単語が含まれていない限り類似度は 0 である. そのため同一の単語はなくとも類似語を数多く含んだ文書間の類似度が適正なものとならない. また, 直接の関連はなくとも, 間接的に関連づいている文書もこのベクトル空間モデルでは適切に扱うことができない.

LSA では共起行列 A に対して特異値分解 (Singular Value Decomposition:SVD) を利用することにより, この問題の解決を図っている. SVD を行うと $m \times n$ 行列 A を次のような行列 U, S, V という 3 つの行列に一意に分解することができる. (ただし, $l = min(m,n), U : m \times l, S : l \times l, V : n \times l$)

$$A = USV^T$$

$$\begin{aligned} A &= USV^T \\ &= \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1l} \\ u_{21} & u_{22} & \cdots & u_{2l} \\ \vdots & \vdots & \ddots & \vdots \\ u_{m1} & u_{m2} & \cdots & u_{ml} \end{pmatrix} \begin{pmatrix} s_1 & & & 0 \\ & s_2 & & \\ & & \ddots & \\ 0 & & & s_l \end{pmatrix} \begin{pmatrix} v_{11} & v_{12} & \cdots & v_{1n} \\ v_{21} & v_{22} & \cdots & v_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ v_{l1} & v_{l2} & \cdots & v_{ln} \end{pmatrix} \end{aligned}$$

ここで, U は左特異行列 (left singular matrix) と呼ばれる $m \times l$ の正規直行行列 ($UU^T = I$, ただし I は単位行列), S は特異値行列 (singular value matrix) と呼ばれる $l \times l$ の対角行列, V は右特異行列 (right singular matrix) と呼ばれる $l \times n$ の正規直行行列 ($VV^T = I$) である. S の対角要素に並ぶ値は特異値と呼ばれ, s_1 が最も大きく, 順に小さくなっていって s_l が最も小さな値となる.

こうして特異値分解によって分解された行列には,「上位 r 個の特異値のみを使って USV^T を掛け合わせた結果は, 元の行列 A の rank r における最小二乗誤差になる」という性質がある. すなわち, $\hat{U} : m \times r, \hat{S} : r \times r, \hat{V} : n \times r$

$$\begin{aligned} \hat{U} &= \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1r} \\ u_{21} & u_{22} & \cdots & u_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ u_{m1} & u_{m2} & \cdots & u_{mr} \end{pmatrix} \\ \hat{S} &= \begin{pmatrix} s_1 & & & 0 \\ & s_2 & & \\ & & \ddots & \\ 0 & & & s_r \end{pmatrix} \\ \hat{V}^T &= \begin{pmatrix} v_{11} & v_{12} & \cdots & v_{1n} \\ v_{21} & v_{22} & \cdots & v_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ v_{r1} & v_{r2} & \cdots & v_{rn} \end{pmatrix} \end{aligned}$$

としたとき, $\hat{A} = \hat{U}\hat{S}\hat{V}^T$ は A の rank r における最小二乗誤差である.

このように, 誤差を最小に保ったまま行列の rank をあえて削減することによって, より関連の強い単語ベクトルが同一次元に縮退され, 類似した値に近似されることが期待できる. そ

のため, 従来のベクトル空間モデルでは適正な類似度が得られなかった, 間接的に関連のある単語間についても高い類似度を得ることができる.