情報処理学会論文誌 Vol.59 No.6 1405–1414 (June 2018)

AutoSort：レガシーシステム分析のための プログラミング言語の判定支援手法

岡田 譲二[1,2,a)] 石尾 隆[3,b)] 坂田 祐司[1,c)] 井上 克郎[2,d)]

受付日 2017年7月10日, 採録日 2018年3月6日

概要：レガシーなメインフレームシステムには，拡張子が存在せずプログラミング言語が不明なソースコードファイルが多数存在する．レガシーシステムを分析する際には各ソースコードファイルのプログラミング言語を判定する必要があるが，これを手作業で行うと多大な労力が必要となってしまう．本研究ではプログラミング言語の判定作業を支援するために，手作業で判定すべきファイルの代表をクラスタリングによって選出する手法を提案する．正確な判定を支援するため，提案手法はパターンマッチによる自動判定と，手作業での判定結果を用いた解析による判定誤りの補正をクラスタリングに組み合わせて用いる．提案手法の評価として，人手で正解のプログラミング言語を付与した 2 つの実際のレガシーシステムのファイル集合に対して本手法を適用した．その結果，提案手法は 19 万ファイルのうち，99.49%のファイルを正しく分類できることを確認した．また，これらのファイルに対する人手での判定は 3.3 人月の工数が必要だったが，提案手法は 8 時間の計算時間と，人手による 15 分の確認だけで判定を完了した．

キーワード：レガシーマイグレーション，リバースエンジニアリング，プログラム理解，クラスタリング

AutoSort: A Supporting Method of Programming Language Detection for Analyzing Legacy Systems

Joji Okada[1,2,a)] Takashi Ishio[3,b)] Yuji Sakata[1,c)] Katsuro Inoue[2,d)]

Received: July 10, 2017, Accepted: March 6, 2018

Abstract: Legacy mainframe systems involve many source code files without file extensions. Their programming languages are undocumented. Therefore, their respective programming languages are not easily judged by hand. Although detecting a programming language for each file is necessary for various program analysis tasks, it is time consuming to manually analyze a large amount of files. In this research, we propose a method to support the process of programming language detection employing a clustering technique to select a small number of representatives for manual detection of programming languages. To improve accuracy, our method combines pattern matching and a static analysis using a result of manual detection. In the experiment, we applied our method to two actual legacy systems whose source code files have been manually analyzed. As a result, our method correctly classified 99.49% of the 190,000 files. While a manual detection of programming languages for the systems required 3.3 man-months, our method completed the analysis in eight hours for computation and fifteen minutes for manual checks of programming languages.

Keywords: legacy migration, reverse engineering, program comprehension, clustering

1. はじめに

業務上は重要だが保守の継続が困難なメインフレーム上

[1] 株式会社 NTT データ技術革新統括本部
NTT DATA Corporation, Koto, Tokyo 176–0024, Japan
[2] 大阪大学大学院情報科学研究科
Osaka University, Suita, Osaka 565–0871, Japan
[3] 奈良先端科学技術大学院大学情報科学研究科
Nara Institute of Science and Technology, Ikoma, Nara 630–0192, Japan
[a)] okadaju@nttdata.co.jp
[b)] ishio@is.naist.jp
[c)] sakatayu@nttdata.co.jp
[d)] inoue@ist.osaka-u.ac.jp

の基幹システム（以降はレガシーシステムと呼ぶ）がいまだに多数存在している [1], [2]．レガシーシステムもビジネス変化に対応するためには保守し続けることが必要であり，そのためにソフトウェア構造の可視化や，新しいプログラミング言語や実行環境への移行といったシステム再構築のための活動が行われる [3], [4]．

レガシーシステムの可視化や再構築の際には，現行システムのソースコードを静的解析し，理解するというアプローチが用いられる [5]．静的解析手法の多くはソースコードの構文解析を行っており，解析を行うための前提条件として，対象となるソースコードのプログラミング言語が何であるかをあらかじめ把握しておく必要がある．

本論文では，レガシーシステムを構成するソースコードファイルに使用されているプログラミング言語を判定する問題を取り扱う．一般的なソフトウェアシステムでは，ソースコードファイルの拡張子やディレクトリ構造からプログラミング言語を容易に判定することができるが，レガシーシステムではプログラミング言語の判定は容易ではない．最も大きな理由として，レガシーシステムでは拡張子やディレクトリが存在しないことがあげられる．メインフレームは拡張子という概念が生まれる前から存在しているため，ファイルに拡張子をつけるという文化がなく，実際に多くのメインフレームのソースコードファイルには拡張子が付与されていない [6]．また，ディレクトリという概念のないフラットなファイルシステムが使われており，プログラミング言語ごとにディレクトリを分けるようなファイル管理が行われておらず，システムを構成する複数の言語のソースコードファイルやプログラムが利用するデータファイルが 1 つの場所に混在する状態となっている．

プログラミング言語判定の単純な自動化，たとえば既知の複数の構文解析器を適用して成功したものを選ぶといった方式は，レガシーシステムに多様な言語が用いられることと，プロジェクト独自の言語が含まれている可能性から適用が困難である．長年の保守開発の中では，一時的な利用や性能上の問題解決という名目によって，たとえばスクリプト系の言語やアセンブラなど，そのプロジェクトで主に採用されている言語とは異なる言語も使われている場合が多い．これらの少数の言語は，保守開発者の入れ替わりによって存在すること自体さえも引き継がれることなく今に至ってしまい，現在の保守開発者にとって未知の言語となっていることがしばしばある．また，規模の大きなプロジェクトでは，そのプロジェクト独自の拡張をプリプロセッサやコンパイラに施すことがある．新しいキーワードを追加したり，文法を追加したりするといった独自の拡張がなされたプログラミング言語は，一見すると元々の COBOL だったり，PL/I に見えたりしても，通常の構文解析器ではまったく解析できないソースコードとなっていることがある．

プログラミング言語の判定においては，ファイル種別の分類も必要である．ファイル種別とは，たとえば C 言語におけるソースファイルとヘッダファイル，COBOL におけるメインファイルとコピーファイルというように，言語としては同一であるが言語処理系にとっての役割が異なるようなファイルの分類である．本研究では，構文解析の起点となるファイルをメインファイル，構文解析の起点とならないファイル（ヘッダファイル，コピーファイルなど）をインクルードファイルと呼んで区別する．インクルードファイルの多くは単独での構文解析が不可能であり，適切な解析を実行するには，これらのファイル種別まで含めたプログラミング言語の判定を行うことが必要である．プログラミング言語に固有のキーワードなどに基づくパターンマッチは，ある程度の自動的な判定は可能であるが，このようなファイル種別の判定には不十分である．

ソースコードファイルが与えられたとき，知識を持った人間なら，そのファイルの内容からプログラミング言語を推定することはそれほど難しくない．しかし，レガシーシステムではソースコードファイルが数万本あることは珍しくなく，すべてのファイルのプログラミング言語を人手で判定するには多大な労力が必要である．第 1 著者が関わったプロジェクトでは，プログラミング言語が不明なソースコードファイルが 17 万ファイル存在しており，そのうち 3 万ファイルは判定開始時に想定されていなかった未知のプログラミング言語のものであった．このファイル集合に対して，ファイル名の命名規約の活用や，ファイル内のキーワードを grep で検索するといった方法で人手で判定を行ったところ，2 人で 1.5 カ月間の作業が必要であった．

本論文では，大規模なレガシーシステムに大量に存在する，プログラミング言語が分からないソースコードファイルについて，そのプログラミング言語とファイル種別を判定する作業を支援する手法を提案する．提案手法のアイディアは，クラスタリングによってファイルを分類し，クラスタの代表として選ばれたファイルだけに対して人間がプログラミング言語の判定を行うというものである．人間の労力を削減するために，パターンマッチによる判定が容易なプログラミング言語は事前に判定を行って人間の判定対象から除外する．また，人間が判定した結果に基づいて構文解析を実行し，得られた情報を用いて誤りを補正することで，分類精度を向上させる．

評価実験として，2 つの実際のレガシーシステムのファイル集合に対して提案手法を適用した．人手で判定した結果を正解として既存手法との正確性の比較を行うとともに，計算時間，作業時間の評価を行うことで，提案手法の有効性を確認した．

以降，2 章では既存手法について紹介し，3 章で提案手法について述べる．4 章で評価実験の方法と結果を説明し，5 章でまとめと今後の課題を述べる．

2. 既存手法

プログラミング言語の判定問題は，レガシーシステムに固有の問題ではない．ドキュメントやメールなどに含まれるソースコード断片に使われているプログラミング言語の識別や，プログラミング言語間で共通の拡張子が用いられている場合への対応といった観点から，プログラミング言語判定の研究が行われている [7]．具体的な判定手法としては，パターンマッチを利用した手法と，教師あり機械学習による判定手法が提案されている．

Linguist [8] は，プログラミング言語ごとに構文の強調表示などを適切に行うことを目的とした，パターンマッチを用いたプログラミング言語の判定ツールである．プログラミング言語の中に含まれる特徴的なキーワードやパターンがあらかじめ列挙されており，それらのキーワードを含む，もしくは決められたパターンにマッチするファイルを，当該のプログラミング言語として判定する．パターンマッチによる判定手法は，他の言語に比べて特徴が明確なプログラミング言語に関しては有効なものの，頻出する単語を共通して持つプログラミング言語を判定する場合，適切なキーワードを作成することが難しい．また，キーワードやパターンを定義していない未知のプログラミング言語に対しては，判定ができない，あるいは誤って既知のプログラミング言語として誤判定してしまうという問題もある．

パターンに頼らないプログラミング言語の判定手法として，教師あり機械学習を用いた手法が提案されている．Klein ら [9] は，ソースファイルの各行に出現する単語や末尾に出現した文字，記号などの特徴を用いた機械学習を提案している．van Dam ら [7] は，ソースコードを単語の列に分解し，n-gram などの機械学習モデルを適用した結果を報告している．これらの手法を適用するには，あらかじめプログラミング言語が判定されているソースコードファイルを多数用意しておく必要があり，拡張子のある一般的なプログラミング言語ではその準備が容易であるが，レガシーシステムでは教師データを準備することが困難である．また，プロジェクトごとに拡張された独自のプログラミング言語などへの対応が困難である．

3. 提案手法：AutoSort

本研究では，人手によるプログラミング言語判定を効果的に行う手法 AutoSort を提案する．提案手法はレガシーシステムのファイル集合を入力として，ファイル名や拡張子の情報を使うことなく，ファイルごとにプログラミング言語とファイル種別のラベルを付与する．ファイル集合にはプログラムの実行時に利用されるデータファイルやパラメータファイルも含まれており，これらは厳密にはプログラミング言語ではないが，プログラミング言語の一種と見なしてラベルをつける．

AutoSort は**図 1** に示すように 4 つのステップから構成される．

(1) パターンマッチによる事前分類
(2) クラスタリング
(3) 人手によるラベリング
(4) 意味解析情報による誤判定補正

提案手法は，まず，パターンマッチによる事前の分類を行う．このステップは，技術的には既存手法と同様である．次に，Klein ら [9] の手法と類似した字句的な特徴を用いたクラスタリングを適用し，人手で判定するべき代表的なファイルを選出する．そして，人手による判定を行ったのち，その結果を用いて呼出関係などの解析を実行し，判定結果と解析結果の一貫性を確認することで誤判定の補正を行う．各手法を段階的に組み合わせることで各手法の弱点を補い，高い精度と少ない工数，未知のプログラミング言語が含まれていたときのロバストネスの実現を目指して設計した手順となっている．

3.1 ステップ 1：パターンマッチによる事前分類

ステップ 1 では，他のプログラミング言語に比べて特徴が明確で判定が容易なものを，パターンマッチによる判定を用いて分類する．特徴が明確で判定が容易なプログラミング言語とは，以下のような特徴を持つプログラミング言語である．

- 当該プログラミング言語で記述されたソースコードであれば，必ず記述する予約語や記法が存在する．
- その予約語や記法は当該プログラム特有であり，他の

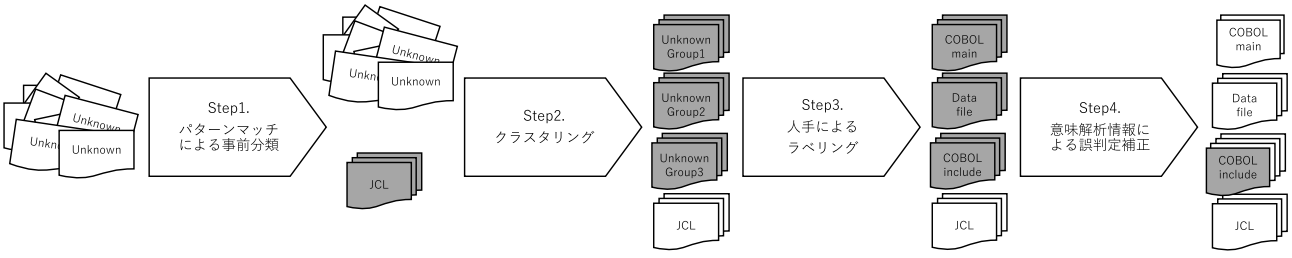


図 1 AutoSort の手順
Fig. 1 Process of AutoSort.

表 1 プログラミング言語判定パターン
Table 1 Patterns of programing language detection.

プログラミング言語・ファイル種別	パターン
アセンブラ	1〜3 文字目がニーモニックキーワードである行数が全行数の 50%を超える
ADL	ダブルクォート内部以外で “␣NAME␣IS” という文字列をファイル中に 1 つ以上含む
JCL メインファイル	“^//.+␣+JOB” と “^//.+␣+EXEC” という文字列をファイル中にそれぞれ 1 つ以上含む
JCL インクルードファイル	“^//.+␣+PROC” という文字列をファイル中に 1 つ以上含む, もしくは “^//.+␣+JOB” という文字列をファイル中に 1 つ以上含むが, “^//.+␣+EXEC” という文字列は 1 つも含まない
Telon	“TRANPORT␣MM/DD/YY␣HH:MM:SS” という文字列をファイル中に 1 つ以上含む

プログラミング言語では（めったに）記述されない．

上記のような特徴を持つ言語どうしの判別であれば，パターンマッチによる判定の問題点が起こらない．パターンマッチで判定する言語を限定的にすることで，世の中に無数に存在するプログラミング言語についての知識がなくとも適切なパターンを作成することができる．

第 1 著者が所属する企業が保守開発する 4 つのレガシーシステムを予備調査した結果，アセンブラ，ADL（Aim Description Language），CA Easytrieve PLUS（以下 EASYPLUS と略す）[10]，JCL，Telon [11] が上記の特徴に当てはまると判断した．特に JCL については，構文解析の起点となる JCL メインファイルと，メインファイルから参照される JCL インクルードファイルという 2 つのファイル種別に分類する判定パターンを定義した．それぞれのプログラミング言語を判定するパターンを**表 1** に示す．表中の文字列はすべて正規表現である．

表 1 の記述順でパターンマッチを適用していき，マッチしたファイルを当該のプログラミング言語として判定する．マッチしなかったファイルを，ステップ 2，3 の判定処理の対象とする．

3.2 ステップ 2：クラスタリング

ステップ 2 のクラスタリングでは，プログラミング言語が持つ様々な特徴を用いて，ステップ 1 のパターンマッチに該当しなかったファイルを k 個のクラスタに分類する．クラスタリングは，未知の言語や独自拡張された言語のソースコードファイルが含まれていても，それらが他のプログラミング言語のファイルと書き方が類似していない限り，異なるクラスタとして認識することができる．なお，ここでの k は手法のパラメータであり，システムに使われているプログラミング言語の数よりも十分に大きい値を設定する．

3.2.1 ファイルの距離

本手法で想定しているファイル集合は，以下のようなものである．

- 単一のディレクトリに存在している
- ファイル名は連番で命名されている
- ファイル名に拡張子が付与されていない

このため本手法では．ソースコードの内容に起因する特徴量のみを選定することとし，名前や拡張子などを特徴量として使用しない．

レガシーシステムにおいては，アルファベットの大文字しか使えない，利用できる記号の種類が少ない，1 行あたりの桁数が固定であるなど，端末の環境の制限が強く，プログラミング言語にもそれらの特徴が反映されているため，これらの特徴に基づいた特徴量を選定することとした．

あるソースファイル f に対して使用する特徴量は以下のとおりである．

- ファイルの行数（空行を含まない）
- 1 行あたりの平均文字数
- 空白を除く 1 ファイル内すべての文字に対する英字の割合
- 空白を除く 1 ファイル内すべての文字に対する数字の割合
- 1 ファイル内すべての英字に対する大文字の割合
- 括弧の 1 行あたりの平均出現数
- 特殊文字（ピリオドやカンマなど）の 1 行あたりの平均出現数
- 他のファイル名と同じ単語の出現数
- 様々な言語の予約語と考えられる単語（if や while など）230 種類それぞれについて，以下の 3 つの特徴量（合計 690 個）．
  - 1 文字目からその単語が出現する行の割合．
  - 各行の前半 1/3 でのその単語の平均出現数．
  - 各行の後半 1/3 でのその単語の平均出現数．

これらの特徴量を選定した理由を**表 2** に示す．

ファイルからの単語の切り出しにおいては，空白，改行文字，引用符およびピリオドを区切り文字として，それ以外の連続する文字列を単語として切り出している．プログラミング言語ごとに引用符などの扱いのルールは異なるため，引用符で囲まれた範囲の内側，外側という区別は行わず，すべての区切り文字の出現を使用する．

特徴量は全部で 698 次元のベクトルと見なすことができるので，ファイル f_1, f_2 の距離 $d(f_1, f_2)$ は，それらに対応する特徴ベクトル $v(f_1), v(f_2)$ のコサイン類似度によって，以下のように定義する．

表 2 クラスタリングで用いる特徴量の選定理由
Table 2 Selection reasons of feature quantity used in clustering.

特徴量	選定理由
ファイルの行数	メインファイルは比較的長く，インクルードファイルは短いことが多い
1 行あたりの平均文字数	1 行あたりの文字数制限がある言語がある
英字の割合	コメントに英字しか記述できない言語と 2 byte 文字（日本語）を記述できる言語がある
数字の割合	データファイルは数字の割合が高い傾向がある
大文字の割合	COBOL などの言語はすべて大文字で記述する傾向がある
括弧の平均出現数	ブロックを括弧で囲む言語と，括弧ではない予約語で囲む言語がある
特殊文字平均出現数	文の区切り文字が言語によって異なる
他のファイル名と同じ単語の出現数	他のファイルの呼び出しやインクルードなどを多数行う言語と，まったく行わない言語がある
出現位置ごとの予約語の平均出現数	予約語は必ず行頭に記述する言語と，どこに記述してもよい言語がある

$$d(f_1, f_2) = 1 - \cos(v(f_1), v(f_2))$$

3.2.2 K-means++ 法の適用

ファイル集合に対して，非階層型クラスタリング手法の 1 つである K-means++ 法 [12] を適用する．非階層型クラスタリングは，分割の良さを測る評価関数を定め，その評価関数を最適にするように分割する手法である．これに対して階層型クラスタリングは，要素数 1 の N 個のクラスタを初期状態として，最も距離が近い 2 つのクラスタを併合する作業を繰り返すことで階層的なクラスタ構造を得る手法である．非階層型クラスタリングは階層型クラスタリングに比べて計算量が小さく，レガシーシステムで分析すべきファイルは数十万を超えることがよくあることから，本手法では実用的な解析時間で終わることを重視し，非階層型クラスタリング手法を選定した．

K-means++ 法は，K-means 法 [13] の初期値設定を工夫したアルゴリズムである．K-means 法では任意の k 個のクラスタ中心を初期値として設定するが，K-means++ 法では以下の手順でクラスタ中心の選択を行う．

(1) 1 つ目のクラスタ中心 c_1 となるファイルを，解析対象ファイル集合 F からランダムに選択する．
(2) 各ファイル f_j $(j \in 1, 2, \cdots, |F|)$ に対して，最も近い選択済みのクラスタ中心との距離 $D(f_j) = \min_i d(c_i, f_j)$ を求める．
(3) 確率分布 $\frac{D(f_j)^2}{\sum_{f \in F} D(f)^2}$ を用いてランダムに次のクラスタ中心 c_l となるファイルを選択する．つまり，どの選択済みクラスタ中心からも離れているようなファイルを高い確率でクラスタ中心として選択する．
(4) クラスタ中心を k 個選ぶまで，2 から 3 を繰り返す．

クラスタ中心の選択以降は，K-means 法と同一で，以下の手順でクラスタリングを実行する．

(1) 各ファイル f_j $(j \in \{1, 2, \cdots, |F|\})$ を，f_j と最も近いクラスタ中心 c_i が所属するクラスタ C_i に割り当てる．
(2) クラスタに属するファイルの特徴ベクトルの平均 $c_i = \frac{1}{|C_i|} \sum_{f \in C_i} v(f)$ を，新しいクラスタ中心とする．
(3) クラスタに変化がなくなるまで，あるいはパラメータとして指定する最大繰り返し数まで，1 と 2 を繰り返す．

K-means 法は計算コストが小さいという利点があるが，1 でランダムに選択するクラスタ中心の初期値によってクラスタリング結果が大きく異なってしまうという弱点がある．K-means 法よりも解析時間，クラスタリング精度の両面で K-means++ 法の方が有効であるため，本手法では K-means++ 法を採用した．

3.3 ステップ 3：人手によるラベリング

ステップ 3 では，ステップ 2 で k 個に分類された各クラスタについて，1 つずつ代表となるファイルを人間に提示し，人間がどのプログラミング言語なのかという判定を行うことで，ラベリングを行う．

代表ファイルとして，本手法では各クラスタの中心に最も近いファイルを選択する．K-means 法は「各要素は群の平均から離れていない」ことを保証する方法であるため，クラスタに含まれるプログラムの「平均」に近いファイルを選ぶことで，出現する単語の数や種類の特徴を見落としにくいと期待している．K-means++ 法では，各クラスタについてクラスタ中心が得られているため，そのクラスタ中心からクラスタに所属する各ファイルまでの距離を容易に計算することができる．

こうして選ばれた代表ファイルを人間に提示し，提示された人間はそのファイルに対してプログラミング言語を判定する．判定された代表的なファイルのプログラミング言語を，そのクラスタに属するすべてのファイルのプログラミング言語としてラベリングする．つまり，このステップが完了した時点で，すべてのファイルに対してプログラミング言語のラベルが付与された状態となる．

ラベルとしてはプログラミング言語の名前を指定するだけであるが，プログラミング言語によってはファイル種別としてメインファイル，インクルードファイルの区別も同時に指定する．また，データファイルに対しては，ファイル形式の名称とは別に，ファイル種別としてデータファイルであることを指定する．これらのファイル種別の指定

は，次のステップ 4 で使用する情報となる．

3.4 ステップ 4：意味解析情報による誤判定補正

ステップ 4 では，ステップ 3 までで判定したプログラミング言語が，既知の言語かつ，構文解析器や意味解析器を持っているプログラミング言語だった場合，そのプログラミング言語と判定されたファイルに対して構文解析および意味解析を実行し，その結果を用いて誤判定の補正を行う．このステップは必須ではなく，ステップ 3 で判定されたプログラミング言語に対応する構文解析器や意味解析器を持っている場合のみ実施し，対応する構文解析器や意味解析器を持っていない場合，単にこのステップを省略する．

このステップで補正が必要になる主な要因は，ステップ 2 におけるクラスタリングにある．同じ言語のメインファイルとインクルードファイルや，サイズが小さいファイルどうしはそれぞれ特徴量の違いが少なく，異なる種類のファイルが偶然同一のクラスタに分類されることがある．中身に何が含まれていても良いデータファイルについても，偶然ソースコードファイルに似ている場合がありうる．

このようなファイル種別の補正に，ソースコードの意味解析から得られるファイル間の関係を利用する．対象のプログラミング言語の構文解析器や意味解析器を持っている必要があるが，意味解析を行うことでファイル内部に出現する他のファイル名の情報を取得し，ファイル間の関係にともなう制約に基づいて，ファイル種別の正確な判定を行う．

ファイル間の関係情報として，呼び出し関係（A calls B），インクルード関係（A includes B），ファイル操作関係（A reads/writes B）の 3 つを用いる．呼び出し関係とは，プログラミング言語の種類に関係なく別のソースコードファイルを呼び出す命令による関係である．COBOL の CALL 文や JCL の EXEC PGM 文などが該当する．インクルード関係とは，ヘッダファイル，コピーファイルのようなインクルードファイルを埋め込む命令による関係である．COBOL の COPY 文や JCL の INCLUDE 文などが該当する．ファイル操作関係とは，ファイルの入出力を行う命令である．COBOL の READ 文や WRITE 文が該当する．

これらの関係はファイルの構文解析および意味解析によってファイル名の出現位置を調べることによって得られるもので，**表 3** に示すように，呼び出し，インクルード，ファイル操作の対象となるファイルは，それぞれメインファイル，A と同じプログラミング言語のインクルードファイル，データファイルに限られるという制約が存在する．ステップ 3 までの判定結果がこれらの制約を満たしていない場合，判定結果が誤っていると考えられるため，これらを用いて，以下のルールに従って補正を実行する．

- ファイル A から B への呼び出しが存在する場合，B は必ず何らかの言語のメインファイルである（A と B は異なる言語であることもありうる）．
  - B が何らかの言語のインクルードファイルと判定されているのなら，メインファイルとインクルードファイルの種別判定を誤ったものと見なし，B の言語のメインファイルと判定する．
  - B がデータファイルの場合は，何らかの言語のメインファイルであるとして，人間に再度提示し，プログラミング言語の判定のやり直しを行わせる．
- ファイル A が B をインクルードする場合，B を A と同じ言語のインクルードファイルであると判定する．
- ファイル A が B に対して何らかのファイル操作命令を実行する場合，B をデータファイルであると判定する．

表 3 ファイル間の関係に基づく制約
Table 3 Constraints based on relationship among files.

ファイル間の関係	ファイルの種別に関する制約
呼び出し関係 (A calls B)	B は何らかのプログラミング言語のメインファイルである．
インクルード関係 (A includes B)	B はインクルードファイルである． B の記述言語は A と同一である．
ファイル操作関係 (A reads/writes B)	B はデータファイルである．

これらの補正ルールは，あるファイル B が他のファイルから受ける操作に注目しており，通常はファイル B の種別に応じて受付ける操作が異なることから，1 つのファイルに対しては 1 つのルールしか適用されない．1 つのファイルに対して複数のルールが適用される場合は，参照元もしくは参照先のファイルの判別や意味解析が失敗していると考えられるため，参照元と参照先両方のファイルを人間に提示して，プログラミング言語の判定のやり直しを行わせる．

このステップが完了すると，提案手法の出力として，プログラミング言語・ファイル種別によってラベル付けされたファイルの一覧が得られる．

4. 評価実験

実際のレガシーシステムのファイル集合 2 つに対して，提案手法である AutoSort と既存手法との比較実験を行った．評価の観点は，提案手法の正確性と，適用に必要な工数および解析時間である．そのために，すべてのファイルに対して従来のやり方であるファイル内のキーワード検索などに頼った人手でのプログラミング言語の判定を行い，それを正解とした．実行時間の評価の基準となる人手の判別工数は，このときの作業時間を計測したものである．

正確性の評価指標としては，クラス分類器の評価で広く採用されている適合率，再現率を用いる．あるプログラミング言語 l について，

- l が正解であるファイルを l と判別した数を $TP(l)$

表 4 実験に用いたファイル集合
Table 4 File sets used for experiment.

システム名	システム A	システム B
総ファイル数	18,399	174,735
言語の一覧	アセンブラ EASYPLUS COBOL CICS MAP Format FORTRAN SQL DATA	アセンブラ EASYPLUS COBOL JCL PL/I ADL DATA Telon
言語の総数	8	8
判別するクラスの一覧	アセンブラ EASYPLUS COBOL MAIN COBOL INC CICS MAP Format FORTRAN SQL DATA	アセンブラ EASYPLUS COBOL MAIN COBOL INC JCL MAIN JCL INC PL/I MAIN PL/I INC ADL DATA Telon
判別するクラス数	9	11
人手による判別工数 (人・時間)	56	485

- l が正解以外のファイルを l 以外と判別した数を $TN(l)$
- l が正解であるファイルを l 以外と判別した数を $FN(l)$
- l が正解以外のファイルを l と判別した数を $FP(l)$

とすると，適合率は $\frac{TP(l)}{TP(l)+FP(l)}$，再現率は $\frac{TP(l)}{TP(l)+FN(l)}$ で表される．また，あるシステム x 全体のプログラミング言語種別の集合を $L(x)$ とし，システム全体のファイル総数を $File(x)$ とするとき，$\frac{\sum_{l\in L(x)} TP(l)}{File(x)}$ をシステム x の全体正解率と呼ぶこととする．

実行時間の評価としては，クラスタリングにおける人手でのラベリングの工数と解析時間を指標として用いる．

4.1 実験方法

実験に用いたファイル集合は，第 1 著者が所属する企業で実際に保守開発を行っているやや小規模なレガシーシステムと，大規模なレガシーシステムの 2 つである．2 つのファイル集合の概要を**表 4** に示す．

判別するクラスとは，言語とファイル種別を足しあわせて重複を省いたものである．言語にメインファイルとインクルードファイルの種別があるものについては，表中では言語名の後ろにそれぞれ MAIN，INC と表記した．たとえばシステム A でいうと，アセンブラ，EASYPLUS，COBOL メインファイル（COBOL MAIN），COBOL インクルードファイル（COBOL INC），CICS MAP，Format，FORTRAN，SQL，DATA の 9 個である．なお，システム A の Format はこのプロジェクト独自の DSL であり，DATA はプログラムの実行時に利用されるデータファイルやパラメータファイルのことである．

提案手法の効果を評価するために，各ファイル集合に対して，以下の 3 つの手法を適用した．

AutoSort 提案手法．クラスタリングにおけるパラメータは，クラスタ数 k を 40，繰り返し数を 10 とした．この k の値は，過去の経験から，多くのシステムで用いられるプログラミング言語を上回ることが想定される値である．

Simon-Weber 機械学習によって言語判定を行う Klein らの手法 [9] に対応する実装 [14] である．この手法はあらかじめ言語の特徴を教師あり機械学習する必要があるため，$K = 10$ で K-分割交差検証を行い，10 回の検証結果の平均を評価した．このとき，ファイル集合は判別するクラスごとに 10 個ずつに分割しており，ファイルの割合は維持したものとなっている．

Clustering 提案手法のクラスタリングのみ，すなわちステップ 2 と 3 のみを単純に適用したもの．特徴量，距離，クラスタリングのパラメータは提案手法と同一である．

AutoSort のステップ 4 で利用する構文解析器・意味解析器としては，JCL，COBOL，PL/I のみを持っているとして，これらの構文解析器・意味解析器を用いる．

AutoSort と Simon-Weber の比較によって既存手法との違いを調査し，また，AutoSort と Clustering の比較によって本研究で導入しているパターンマッチや誤判定の補正の効果を調査する．

AutoSort および Clustering の処理は Python を用いて実装した．実行時間の計測環境は，CPU として Intel Xeon E7-2860 2.27 GHz を搭載し，256 GB RAM，15,000 rpm HDD を記憶領域として備えた計算機である．

4.2 実験結果

4.2.1 正確性の評価

正確性の比較実験結果を**表 5** と**表 6** に示す．提案手法はほとんどのプログラミング言語・ファイル種別で適合率，再現率が 95%以上となった．また，全体正解率は両システムで 99%以上，合計で 99.49%となっており，従来手法や単純なクラスタリングの適用に比べて適合率，再現率はいずれも大きく向上していることが分かる．

提案手法の各ステップ完了時点での適合率の値を**表 7** に，再現率の値を**表 8** に示す．これらの表は紙面の都合上システム B についての結果のみを示すが，ステップ 3 までである程度高い適合率，再現率を達成していることが分かる．仮にステップ 4 で用いた JCL，COBOL，PL/I の構文解析器を持っていなかった場合は，このステップ 3 までの

表 5 正確性の比較実験結果：システム A
Table 5 Comparative experiment result on accuracy: System A.

		AutoSort		Simon-Weber		Clustering	
判別するクラス	Correct	適合率 (%)	再現率 (%)	適合率 (%)	再現率 (%)	適合率 (%)	再現率 (%)
アセンブラ	204	96.52	95.10	69.66	79.90	100.00	45.59
EASYPLUS	337	100.00	98.81	30.66	87.24	14.46	10.68
COBOL MAIN	8,858	99.92	100.00	92.31	27.51	99.95	97.53
COBOL INC	5,891	100.00	100.00	44.26	74.20	98.15	92.65
CICS MAP	2,769	100.00	99.53	85.66	97.91	99.70	96.86
Format	198	85.53	98.48	63.52	74.75	97.93	95.45
FORTRAN	18	100.00	100.00	1.86	72.22	77.78	77.78
SQL	23	100.00	100.00	37.93	95.65	100.00	95.65
DATA	101	92.86	77.23	15.12	79.21	10.75	99.01
全体正解率			99.71		55.65		93.66

表 6 正確性の比較実験結果：システム B
Table 6 Comparative experiment result on accuracy: System B.

		AutoSort		Simon-Weber		Clustering	
判別するクラス	Correct	適合率 (%)	再現率 (%)	適合率 (%)	再現率 (%)	適合率 (%)	再現率 (%)
ADL	3,704	100.00	100.00	0.00	0.00	98.43	96.44
アセンブラ	277	98.58	100.00	4.04	16.25	0.00	0.00
EASYPLUS	292	99.65	98.29	2.57	34.59	0.00	0.00
COBOL MAIN	20,960	100.00	99.99	60.54	9.95	100.00	100.00
COBOL INC	19,435	99.83	100.00	37.90	57.93	97.03	97.39
JCL MAIN	3,187	100.00	99.34	8.51	50.89	41.17	54.47
JCL INC	21,062	99.75	100.00	61.55	67.51	93.00	85.81
PL/I MAIN	22,312	99.38	98.24	58.34	63.83	96.69	93.96
PL/I INC	52,307	99.76	99.76	74.01	53.35	97.18	93.74
Telon	463	100.00	100.00	11.77	67.17	0.00	0.00
DATA	30,736	99.87	98.78	69.86	67.36	86.59	98.18
全体正解率			99.47		52.93		93.54

表 7 各ステップでの適合率：システム B
Table 7 Precision at each step: System B.

ステップ	1	2 , 3	4
ADL	100.00	100.00	100.00
アセンブラ	98.58	98.58	98.58
EASYPLUS	99.65	99.65	99.65
COBOL MAIN	0.00	100.00	100.00
COBOL INC	0.00	99.82	**99.83**
JCL MAIN	100.00	100.00	100.00
JCL INC	99.75	99.75	99.75
PL/I MAIN	0.00	85.45	**99.38**
PL/I INC	0.00	98.66	**98.76**
Telon	100.00	100.00	100.00
DATA	0.00	95.93	**99.87**

表 8 各ステップでの再現率：システム B
Table 8 Recall at each step: System B.

ステップ	1	2, 3	4
ADL	100.00	100.00	100.00
アセンブラ	100.00	100.00	100.00
EASYPLUS	98.29	98.29	98.29
COBOL MAIN	0.00	99.99	99.99
COBOL INC	0.00	95.12	**100.00**
JCL MAIN	99.34	99.34	99.34
JCL INC	100.00	100.00	100.00
PL/I MAIN	0.00	98.24	98.24
PL/I INC	0.00	92.32	**99.76**
Telon	100.00	100.00	100.00
DATA	0.00	98.78	98.78

適合率，再現率となる．

提案手法のステップ 4 によって，クラスタリングの際に誤判定されたインクルードファイルが正しい分類に再分類されている．ステップ 4 は工数をかけて構文解析器を多種類用意できれば，より判定精度が上がることが予想される．ステップ 3 までの結果を基に，ファイル数が多そうなプログラミング言語から優先的に構文解析器を用意すると効率良く判定精度を向上させることができると考えられる．

適合率，再現率の高さは，第 1 著者が所属する企業では十分に実用的であると判断された．適合率，再現率が

表 9 工数および解析時間の比較結果：システム A
Table 9 Comparative experiment result on utility: System A.

手法	AutoSort	S.-Weber	Clustering	人手
人手時間	7 m 41 s	0 m 00 s	7 m 37 s	6 人日
解析時間	46 m 49 s	11 m 42 s	21 m 03 s	-

表 10 工数および解析時間の比較結果：システム B
Table 10 Comparative experiment result on utility: System B.

手法	AutoSort	S.-Weber	Clustering	人手
人手時間	7 m 00 s	0 m 00 s	7 m 12 s	3 人月
解析時間	7 h 10 m 40 s	2 h 32 m 14 s	3 h 16 m 48 s	-

100%にならないことを前提とした本手法の利用プロセスの確立が議論され，レガシーシステムの分析サービスを受ける顧客への事前説明を行ったうえで現場が利用するツールの 1 つとして採用された．

ただし，基幹システムの移行作業においては，システムの機能を見落とすことが多大な影響を及ぼす可能性があり，再現率について 100%を求められる場合も存在する．このような場合には人間がすべてのファイルを目視する作業をなくすことができない．しかしその場合であっても，提案手法によってソースファイルにラベル付けされていれば，作業者はほとんどのソースファイルのラベルを確認する作業だけで済むため，一定の作業時間短縮の効果はあるものと思われる．判定結果として複数のプログラミング言語を候補として出力するなど，再現率を 100%とするような拡張を行うことが今後の課題である．

4.2.2 適用に必要な工数および解析時間の評価

提案手法の実行および人手での分析に要する工数の比較実験結果を**表 9** と**表 10** に示す．人手による分析作業の場合，1 ファイルあたりの作業時間は平均で 10 秒程度であるが，ファイル数に比例した工数が必要であった．提案手法ではファイル数とは関係なく，クラスタ数に比例する工数で実施できる．提案手法における分析でも 1 ファイルあたり 10 秒程度という工数は変わらず，$k = 40$ とした本実験ではシステム A，システム B ともに約 7 分であった．全自動で動作する機械学習手法に比べると作業時間は必要であるが，十分に実用的として現場でも受け入れられた．

なお今回の実験では，手作業での実験開始時に分析者にとって未知の言語であったもの（Format，FORTRAN）が含まれていたため，実作業時にはこの言語が何であるかを有識者に数時間インタビュしているが，本実験の工数にこの時間は含めていない．この工数はどの手法を採用するにしても必要な工数であり，ファイル数とは関係なく一定の工数が必要であると考える．

4.2.3 本手法の適用範囲

本手法で選定した特徴量は，レガシーシステムの端末環境に由来するプログラミング言語の特徴を反映したものであるため，レガシーシステムのプログラミング言語であれば精度良く分類ができると考えている．

一方で，レガシーシステム以外でよく用いられるプログラミング言語（C 言語や Java など）に関しては，本手法で提案した特徴量以外に，新しい特徴量を導入する必要があるかもしれない．しかし，クラスタリングによって代表ファイルを選出するというアイディア自体はレガシーシステムに固有というわけではないため，文書中に埋め込まれたソースコード断片などのプログラミング言語判定に応用することも考えられる．

プログラミング言語によっては，複数のバージョン，派生言語，方言などが存在する．そのようなプログラミング言語が複数使用されているようなシステムを対象として，それらの違いを判定しなければいけない場合，ファイル間での特徴量の差異が小さくなり，分類精度が低下すると考えられる．

5. おわりに

本論文では，レガシーシステム内に雑多に混在する複数のプログラミング言語で記述されたファイル集合から，それらのプログラミング言語およびファイル種別を判定する手法 AutoSort を提案した．提案手法は，クラスタリングによって特徴が類似するファイルをまとめ，そこから人間が判定すべき代表ファイルを選出することで，効果的な判定を可能とした．また，作業者の労力を減らすために，パターンマッチによる一部の言語の分類と，事後的な解析による誤判定の補正を組み込んだ．

提案手法と既存手法との比較実験では，クラスタリングが機械学習手法よりも正確な結果となること，また，パターンマッチおよび誤判定の補正が提案手法の正確さに貢献していることを確認した．適用する場合の作業工数も十分に現実的なものであり，現場で使用するツールとして受け入れられた．

今後の課題としては，再現率について 100%を求められるような状況に対応するための再現率を重視した提案手法の拡張があげられる．また，クラスタリングによって代表ファイルを選出するというアイディア自体はレガシーシステムに固有というわけではないため，文書中に埋め込まれたソースコード断片などのプログラミング言語判定に応用することも考えられる．

謝辞 今野有一氏，NTT データ数理システム阿部裕介氏には本研究についてアドバイスを賜った．またニューソン株式会社永田宏樹氏には実験にご協力していただいた．ここに謝意を表する．

参考文献

[1] Bennett, K.: Legacy systems: Coping with success, *IEEE Software*, Vol.12, No.1, pp.19–23 (1995).

[2] Khadka, R., Batlajery, B.V., Saeidi, A.M., Jansen, S. and Hage, J.: How do professionals perceive legacy systems and software modernization?, *Proc. 36th International Conference on Software Engineering*, pp.36–47 (2014).
[3] Bisbal, J., Lawless, D., Wu, B. and Grimson, J.: Legacy information systems: Issues and directions, *IEEE Software*, Vol.16, No.5, pp.103–111 (1999).
[4] Ganesan, A.S. and Chithralekha, T.: A Survey on Survey of Migration of Legacy Systems, *Proc. International Conference on Informatics and Analytics*, pp.72:1–72:10 (2016).
[5] Bisbal, J., Lawless, D., Wu, B., Grimson, J., Wade, V., Richardson, R., O' Sullivan, D.: A survey of research into legacy system migration, Technical Report, Trinity College Dublin (1997).
[6] 神居俊哉，高尾 司：メインフレーム実践ハンドブック：z/OS (MVS), MSP, VOS3 のしくみと使い方，リックテレコム (2009).
[7] van Dam, J.K. and Zaytsev, V.: Software Language Identification with Natural Language Classifiers, *Proc. 23rd International Conference on Software Analysis, Evolution, and Reengineering*, pp.624–628 (2016).
[8] Github: Linguist, available from ⟨https://github.com/github/linguist⟩.
[9] Klein, D., Murray, K. and Weber, S.: Algorithmic Programming Language Identification, *CoRR*, Vol.abs/1106.4064 (2011) (online), available from ⟨http://arxiv.org/abs/1106.4064⟩.
[10] CA Technologies: CA Easytrieve PLUS, available from ⟨http://www.ca.com/jp/devcenter/ca-easytrieve.aspx⟩.
[11] CA Technologies: CA Telon, available from ⟨http://www.ca.com/us/products/detail/ca-telon.aspx⟩.
[12] Arthur, D. and Vassilvitskii, S.: K-means++: The Advantages of Careful Seeding, *Proc. 18th ACM-SIAM Symposium on Discrete Algorithms*, pp.1027–1035 (2007).
[13] MacQueen, J.: Some methods for classification and analysis of multivariate observations, *Proc. 5th Berkeley Symposium on Mathematical Statistics and Probability*, pp.281–297 (1967).
[14] Weber, S.: Programming-Language-Identification, available from ⟨https://github.com/simon-weber/Programming-Language-Identification⟩.

岡田 譲二 （正会員）

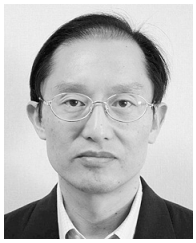
2006 年名古屋大学工学部電気電子情報工学科卒業．2008 年同大学大学院博士前期課程修了．同年株式会社 NTT データ入社．プログラム解析技術の研究開発および現場適用に従事．

石尾 隆 （正会員）

2003 年大阪大学大学院基礎工学研究科博士前期課程修了．2006 年同大学大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員（PD）．2007 年大阪大学大学院情報科学研究科助教．2017 年奈良先端科学技術大学院大学情報科学研究科准教授．博士（情報科学）．プログラム解析，プログラム理解に関する研究に従事．

坂田 祐司 （正会員）

1996 年東京大学大学院工学系研究科材料学科博士前期課程修了．同年 NTT データ通信株式会社（現，株式会社 NTT データ）入社．プログラム解析，システムモダナイゼーションに関する研究に従事．



井上 克郎 （正会員）

1979 年大阪大学基礎工学部情報工学科卒業．1984 年同大学大学院博士課程修了．同年同大学基礎工学部助手．1984～1986 年ハワイ大学マノア校情報工学科助教授．1989 年大阪大学基礎工学部講師．1991 年同助教授．1995 年同教授．2002 年大阪大学大学院情報科学研究科教授．工学博士．ソフトウェア工学，特に，ソフトウェア開発手法，プログラム解析，再利用技術の研究に従事．本会フェロー．