特別研究報告

題目

ソースコード特徴量を用いた機械学習による
ソースコードの良否の判定

指導教員

井上 克郎 教授

報告者

槇原 啓介

平成 31 年 2 月 12 日

大阪大学 基礎工学部 情報科学科

平成 30 年度 特別研究報告

ソースコード特徴量を用いた機械学習によるソースコードの良否の判定

槇原 啓介

内容梗概

プログラミングコンテストは，アルゴリズムの問題に対する解答ソースコードを提出し，その優劣を競うコンテストであり，上級者から初級者まで数多くの人が参加している．参加者の成績はその順位によってレーティングが決定される．近年，プログラミングコンテストとレーティングに着目し，解答ソースコードを対象とした研究が行われている．

プログラミングコンテストのレーティングのように，何らかの基準に従ってソースコードの良否を判定することは，プログラミングの学習を支援することができる．例えば，低いレーティングのソースコードを修正し，高いレーティングのソースコードにすることができれば，その修正は適切だったと考えられ，書き手は自分の成長を実感できる．上級者と初級者の間にある差異は，ソースコードの良否の判定や修正の指針に利用できるのではないかと考えた．

そこで本研究では，ソースコード特徴量を用いて，その良否を自動で判定できる機械学習を用いた手法を提案する。具体的には，まずプログラミングコンテストにおいて提出された大量のソースコードに対し，字句解析やメトリクス収集などの静的解析を行い，ソースコード特徴量を取得する．対象となったソースコードに対して，書き手が上級者であるもの，初級者であるものをそれぞれ「良」「否」の 2 種類の値に分類し，ソースコード特徴量と良否を機械学習により学習モデルを作成する．作成した学習モデルに対し，ソースコードのソースコード特徴量を入力すると，自動で良否を出力する．また，判定の結果否とされたソースコードに対して，良となるような修正のための指針を自動的に与える方法について検討を行った．修正の指針については，入力されたソースコードに対して，初級者と比較して上級者の値が大きいソースコード特徴量が存在しない場合，使用を推奨するように提示を行う．

本手法の実装を行い、評価実験として上級者，初級者のソースコードの良否を分類した結果，およそ 90%の精度で判定を行うことができた．また，否と判定されたソースコードを，与えられた指針に従って修正したところ，良と判定される事例を確認した．

主な用語

プログラミングコンテスト，レーティング，機械学習

目 次

1 まえがき **3**

2 背景 **4**

2.1 オンラインジャッジシステム 4

2.2 プログラミングコンテスト 4

2.2.1 ルールと流れ 5

2.2.2 開催規模や参加者 5

2.2.3 レーティングシステム 5

2.3 機械学習 6

3 ソースコード編集における上級者・初級者間の比較調査 **8**

3.1 調査内容 8

3.2 データセット 8

3.2.1 ソースコードデータ 12

3.2.2 提出履歴データ 12

3.2.3 統計情報 13

3.3 上級者・初級者の定義 14

3.4 調査結果 15

3.4.1 予約語利用頻度の調査 15

3.4.2 メトリクスの値に関する調査 18

3.5 本研究への利用 21

4 提案手法 **22**

4.1 目的 22

4.2 ソースコードの良否の判定 22

4.3 修正の指針の提示 27

4.4 実験結果 30

4.4.1 学習精度 30

4.4.2 修正後の良否の判定の変化 33

5 まとめ **36**

謝辞 **37**

参考文献 **38**

1 まえがき

ソースコードを編集することは，ソフトウェアを開発する上で必要不可欠な作業であり，ソフトウェア工学の研究分野として開発者がソースコードに対して行う編集作業の内容に関する研究は数多く行われている [4–6]．開発の過程においては，ソースコードの実装，テスト，修正を繰り返し行うため，これらの作業にかかる労力は大きい．特に，初級者は時間的コストが多くかかるため，行われた編集が適切であるかどうか，判断できることが望ましい．2020 年度から小学校においてもプログラミング教育が必修化される[1]など，プログラミングを学習し始める人の数は今後ますます増加していくことが予想されるため，その基本的な作業の 1 つである編集作業を適切に支援することが今後重要となると考えられる．

ソースコードが与えられた時，その内容の良否を仮に判定できるならば，プログラミングの学習に役立つと考える．仮に否と判定されたソースコードを修正し，その結果良と判定されるようになれば，修正した編集は適切であったと言える．また，結果が良好になることによって、プログラミングを行った作業者は自分の成長を実感できる．さらに良否の判定に用いた基準を利用し，より良い判定を得られるような修正内容を支援することができれば，初級者はそれらを参考にプログラミング学習を進めることができるようになり，効率的な技術力の向上を期待できる．

良いソースコードとはどのようなものかを考えた場合，構造や読みやすさ，実行時間など様々な基準が考えられる．しかしながら人によって基準が異なるため，広く合意できる判断を行うことは難しい．広く受け入れられる基準を得るためには，多くのソースコードを用いて定量的にその良否を判定し、その判断に基づく基準が重要である．ソースコードの編集作業に関して，上級者と初級者間にどのような差異が存在するのかという研究 [1] は既に行われている．その研究によると，両者の間には，予約語利用頻度やメトリクスといった定量的なソースコード特徴量の差異が確認されている．この特徴の差異を利用すれば，書かれたソースコードが上級者に近いか，それとも初級者に近いかということを定量的に判断し，修正の指針を与えることができるのではないかと考えた．

本研究では，ソースコードの良否を定量的かつ自動で判断する手法として機械学習を用いる．プログラミングコンテスト内で提出されたソースコードに対し，上級者・初級者間で差異が見られたソースコード特徴量の値と，ソースコードの「良」「否」を機械学習プログラムに与えることによって学習モデルを作成する．良否に関しては，上級者が書いたソースコードを「良」，初級者が書いたソースコードを「否」と定義する．作成した学習モデルに対し，学習に用いられていないソースコード特徴量の値を入力すると，ソースコードの良否を自動で出力する．出力結果が否の場合は，ソースコード特徴量を用いて修正の指針を提示する．

[1] http://www.mext.go.jp/a_menu/shotou/zyouhou/detail/1375607.htm

2 背景

この章では，本研究における背景として，プログラミングコンテストと，機械学習について述べる． プログラミングコンテストには様々な種類があるが，本研究ではアルゴリズムに関する問題を時間内に解く種類のコンテストについて述べる． 以下ではまず，プログラミングコンテストに用いられるオンラインジャッジシステムと，プログラミングコンテストの概要について説明する．

2.1 オンラインジャッジシステム

プログラミングコンテストにおいて，その採点に利用されるオンラインジャッジシステムについて説明する． オンラインジャッジシステムは，利用者に問題を提示し，問題に対する利用者の回答を受け取る． そして，受け取った回答を採点し，結果を利用者に通知する． オンラインジャッジシステムの一例として，国内で代表される AIZU ONLINE JUDGE[2]，アメリカの Topcoder が存在する．

オンラインジャッジシステムが提示する問題には，問題文，サンプルテストケース，実行時間やメモリ制限等の実行上の制約などが含まれる． 利用者は制約を満たすプログラムを作成し，そのソースコードをシステムに提出する． システムは提出されたソースコードをコンパイルし，予め用意されている複数のテストケースを実行する． 実行後，テストケースを通過すれば正解を，間違った出力や実行制限を違反した場合はその旨を利用者に通知する． システムによってソースコードがコンパイル，実行されるため，システムに環境が用意されている任意の言語で提出することができる．

2.2 プログラミングコンテスト

プログラミングコンテストとは，プログラミングの能力や技術を競い合うコンテストのことである． オンラインジャッジシステムを用いて複数の参加者が同じ問題セットを同じ時間に解く方式で行われる． 参加者の正解問題数や回答時間に応じて参加者の順位が決定し，コンテスト終了後に順位に応じて参加者のレーティング [2,3] が変動する． プログラミングコンテストには ACM ICPC5[3]のように複数の参加者がチームを組んで回答するものも存在するが， 本研究では個人で参加する種類のプログラミングコンテストを対象とする．

[2]http://judge.u-aizu.ac.jp/onlinejudge/
[3]https://icpc.baylor.edu/

2.2.1 ルールと流れ

プログラミングコンテストの実際の流れについて，本研究でデータセットとして用いるCodeforces[4]を例として説明する．Codeforcesにおける一般的なコンテストでは，指定の時間になるとコンテスト参加者に複数の問題が公開される． 参加者は自由な順序・プログラミング言語で問題を解くことができ，解いた問題の難易度と回答までにかかった時間に応じて参加者に得点が加算される． 参加者は各問題に対して何度でも回答を提出することができ，回答ソースコードの正誤やコンパイル可能性については提出の可否に影響しない． 回答ソースコードは提出された時点でCodeforcesに用意されたオンラインジャッジシステムによってコンパイル，事前テストの実行が行われる． 事前テストにはテストケースの一部が用いられ，事前テストが通過したか否かについては提出後に参加者へ通知される． 事前テストの通過は全テストケースの通過を保証しないため，事前テストを通過していても回答が正しくない場合もある． 事前テストの結果を確認し，参加者は再提出を行うか別の問題に回答するかの判断を行う． コンテスト時間終了後に提出ソースコードに対して最終テストが実施され，参加者の最終的な得点が決定する． 得点に応じて各参加者の順位が決定し，レーティングが変動する．

2.2.2 開催規模や参加者

本研究で利用するCodeforcesにおいては,

- 開催頻度: 月に2，3回程度のコンテスト開催
- 参加人数: 毎コンテスト7000人程度
- 国籍: 全世界から参加 (言語はロシア，英語)

となっている． また，過去6か月以内に一度でもコンテストに参加したことのあるユーザーをCodeforcesではアクティブユーザーと定義しているが，Codeforcesにおけるアクティブユーザー数は2019年2月5日現在，52663人となっている．

2.2.3 レーティングシステム

Codeforces は参加者の熟練度を示す指標としてレーティングシステム [2] を用いている．主にチェスやサッカーで用いられるイロレーティング [3] に似た計算方式を採用しており，コンテスト毎に参加者の順位に応じてレーティングを計算する． レーティングが高い参加者ほどコンテストで高い成績を取っているということができる． 本研究においても，この

[4] http://codeforces.com/

レーティングが高い参加者を熟練度が高い参加者とし，アルゴリズムの問題に対して適切にコーディングを行う能力が高いと考える． イロレーティングは 2 者間における強さの指標を表す． A と B のレーティングをそれぞれ r_A, r_B と表すとき, A が B に勝利する確率が,

$$P\left(r_A, r_B\right) = \frac{1}{1+10^{\frac{r_B - r_A}{400}}}$$

となるようにレーティングが調整される． プログラミングコンテストでは，A が B の得点を上回る確率といえる． Codeforces においてはコンテスト終了後の順位を用いてレーティングの変更計算を行う． Codeforces でのレーティング変更計算のアルゴリズムを Algorithm1 に示した． (1) ではおおよそのレーティング変動を計算している． $seed_i$ は，レーティング r_i の参加者の順位期待値を表し，$seed_i$ と実際の順位 $rank_i$ との幾何平均を m_i とする． ここで，$getSeed\left(R_i\right) = m_i$ となるようなレーティング R_i の探索を行い，この R_i と r_i の平均が参加者 i の暫定的なレーティングとなる． また，上位参加者のレーティングがインフレしてしまうことを防ぐために (2) の調整を行っている． (1) と (2) によって計算されたレーティングの変動を r_i に加え，最終的な参加者のレーティングを得られる．

2.3 機械学習

機械学習とは，コンピュータがデータを反復的に学習することによってデータの中にあるパターンを見つけ出し，新たなパターンに当てはめて結果を予測することである．大量の複雑なデータを処理することによって，正確性が高い結果を速やかに提供することができる．機械学習には，教師あり学習と教師なし学習がある．教師あり学習は，既知の入力データと出力データを用いてモデルを訓練し，将来の出力を予測する．既存の出力データがある場合に教師あり学習を使用する．教師なし学習は，入力データの隠れたパターンや固有の構造を見出す．本研究においては，教師あり学習を用いる．

教師あり学習のアルゴリズムは，既存の一連の入力データとそれに対する出力を用いてモデルを訓練し，新たなデータへの出力を合理的に予測できるようにするものである．予測に使用する入力部分のデータを説明変数，予測したい出力部分のデータを目的変数という．予測の精度を確認するために，データを訓練データとテストデータに分割する．訓練データで学習したモデルに対し，テストデータの説明変数を入力し，出力の目的変数が一致するかどうかを確認する．教師あり学習では，分類や回帰の手法を用いて予測モデルを作成する．分類は離散的，回帰は連続的な出力の予測を行う．分類のアルゴリズムには，サポートベクターマシン (SVM)，決定木，k 最近傍法，単純ベイズ，判別分析，ロジスティック回帰，ニューラルネットワークなどが，回帰のアルゴリズムには，線形回帰，非線形回帰，正則化，ステップワイズ回帰，決定木，ニューラルネットワーク，適応ニューロファジー学習などが挙げられる．本研究では，分類のうち，精度が高かった決定木と SVM を採用した．

Algorithm 1：Codeforces におけるレーティング変更計算

```
Data: U：コンテストの全参加者
Data: r_i：参加者 i のコンテスト前のレーティング
Data: rank_i：参加者 i のコンテストでの順位
Result: r'_i：参加者 i の変化後のレーティング
  Function getSeed(r)
    return Σ_{i∈U} P(r_i, r) + 1;
  Function getRatingToRank(r)
    left ← 1;
    right ← 8000;
    while right − left > 1 do
      mid ← (left+right)/2;
      if getSeed(mid) < r then
        right ← mid;
      else
        left ← mid;
    return left;
  sum ← 0;
  for i ∈ U do                                         // (1)
    seed_i ← getSeed(r_i);
    m_i ← √(seed_i * rank_i)
    R_i ← getRatingToRank(r_i);
    d_i ← (R_i+r_i)/2;
    sum ← sum + d_i;
  for i ∈ U do                  // レーティング変動の合計を 0 にする
    d_i ← d_i − (sum/|U| + 1);
                      // (2) 上位のレーティングのインフレを抑える処理
  topU ← U の上位 min(|U|, 4√|U|) 人;
  sum ← Σ_{i∈topU} d_i;
  inc ← min(max(−sum/|topU|, −10), 0);
  for i ∈ U do                        // 最終的なレーティングの決定
    d_i ← d_i + inc;
    r'_i ← r_i + d_i;
```

3　ソースコード編集における上級者・初級者間の比較調査

上級者・初級者間の編集にどのような差異が存在するのかという研究 [1] は既に行われている．この章では，関連研究の調査内容と調査結果のうち，本研究に関連する部分について述べる．

3.1　調査内容

関連研究では，世界最大規模のプログラミングコンテストである Codeforces の参加者を対象に，参加者がコンテストの問題へ回答として提出したソースコードの編集作業について調査を行った．特にソースコードにおける特徴量や修正箇所，修正回数がプログラミングコンテスト上級者・初級者間でどのように変化するかについて分析を行い，ソースコード編集者の熟練度と編集作業の特徴との間にどのような関係があるかを調査している．ソースコード編集作業の特徴調査を行うために，2016/5/19〜2016/11/15 の 6ヶ月間に Codeforces に対して提出された 1,644,636 のソースコードと，それに付随した提出履歴情報を収集しており， このうち，収集ソースコードの 9 割を占める C++でのソースコードが調査対象となった．収集したソースコードデータと提出履歴情報はデータベースに保管しており，オンラインで公開してある[5]．以下で，この研究に用いられたデータセットの説明と調査結果の説明を行う．ここで述べる調査結果とは， 初回提出ソースコードで用いられている予約語の利用頻度の調査，及び Web 上に公開されているソースコードメトリクス計測ツールである SourceMonitor[6]を用いて計測できる 11 種類のメトリクスについての調査結果である．

3.2　データセット

データセットは，参加者が問題への回答として提出したソースコードと，言語やタイムスタンプ等の提出履歴情報データベースの 2 種類からなり，提出履歴情報データベースの内容は表 1 に示される通りである．データセットの統計情報は表 2 にて示しており，本データは 2016/5/19〜2016/11/15 の期間に収集されている．ソースコードのファイル数は 1,644,636 で，プログラミングコンテストにおける提出は 1 つのソースファイルにまとめられるため，これは提出数と一致している．参加者数は，2016/5/19〜2016/11/15 の期間に 1 度以上 Codeforces のコンテストに参加したユーザーの総数である．また，各コンテストには複数の問題が含まれるため，コンテストの数と比較して問題数が多くなっている．

[5]https://sites.google.com/site/miningprogcodeforces/
[6]http://www.campwoodsw.com/sourcemonitor.html

表 1: データセットの内訳

テーブル名	カラム名	キー	説明	データ型
Participant	user_name	PK	Codeforces の参加者名	String
	rating		参加者の現在のレーティング	Integer
	max_rating		2016/11/15 までの最大到達レーティング	Integer
Participant-Submission	user_name	PK FK	*Participant* テーブルにおける *user_name*	String
	files		データセット内における *user_name* の提出数	Integer
File	file_name	PK	データセット上でのソースファイル名	String
	submission_id		Codeforces における提出 ID	Integer
	lang		ソースファイルのプログラミング言語	String
	user_name	FK	ソースファイルの提出者	String
	verdict		提出の正誤	String
	timestamp		提出時刻	Date/Time
	competition_id	FK	*Competition* テーブルにおける *competition_id*	Integer
	problem_id		この提出に対応する *problem_id*	String
	url		Codeforces におけるこのソースファイルの URL	String
	during_competition		この提出がコンテスト時間内に提出されたかどうか	Boolean
Competition	competition_id	PK	コンテスト ID	Integer
	name		コンテスト名	String
	start_time		コンテスト開始時刻	Date/Time
	duration_time		コンテスト時間	Integer
	participants		コンテスト参加者数	Integer

表は次ページに続く

前ページからの続き

テーブル名	カラム名	キー	説明	データ型
Problem	problem_id	PK	問題 ID	String
	competition_id	FK	この問題が掲載されたコンテストの *competition_id*	Integer
	prob_index		Index of the problem in the competition	String
	points		コンテストにおけるこの問題の得点	Integer
Acceptance	problem_id	PK FK	対応する問題の *problem_id*	String
	submission_in_sample		データセット上におけるこの問題に対する提出数	Integer
	solved_in_sample		データセット上におけるこの問題に対する正答数	Integer
	submission		この問題に対する全提出数	Integer
	solved		この問題に対する全正答数	Integer
	acceptance_rate		$solved/submissions$	Double
	lastmodified		データの最終更新日	Date/Time
Problem-Submission-Statistics	problem_id	PK	対応する *problem_id*	String
	filesize_max		この問題に対する提出ファイルサイズの最大値	Integer
	filesize_min		この問題に対する提出ファイルサイズの最小値	Integer
	filesize_mean		この問題に対する提出ファイルサイズの平均値	Double
	filesize_median		この問題に対する提出ファイルサイズの中央値	Integer
	filesize_variance		この問題に対する提出ファイルサイズの分散	Double
	filesize_max_competition		$filesize_max$ のうちコンテスト中に提出されたもの	Integer

表は次ページに続く

前ページからの続き

テーブル名	カラム名	キー	説明	データ型
	filesize_min _competition		*filesize_min* のうちコンテスト中に提出されたもの	Integer
	filesize_mean _competition		*filesize_mean* のうちコンテスト中に提出されたもの	Double
	filesize_median _competition		*filesize_median* のうちコンテスト中に提出されたもの	Integer
	filesize_variance _competition		*filesize_variance* のうちコンテスト中に提出されたもの	Double
Submission-Distance	file_name	PK	対応するソースファイル名	String
	proble_id		この提出に対応する *problem_id*	String
	next_file		*file_name* の次の提出	String
	submission_index		同じ問題に対してこの提出が何番目の提出か	Integer
	levenshtein_distance		*file_name* と *next_file* とのトークンベースの編集距離	Integer
	add_node		*file_name* から *next_file* にかけての追加ノード数	Integer
	delete_node		*file_name* から *next_file* にかけての削除ノード数	Integer
	update_node		$file_name$ から next_*file* にかけての更新ノード数	Integer
	move_node		*file_name* から *next_file* にかけての移動ノード数	Integer
	node_sum		追加，削除，更新，移動ノード数の合計	Integer

表 2: データセットの統計情報

収集期間	ファイル数	参加者数	コンテスト数	問題数	DB サイズ
2016/5/19〜2016/11/15	1,644,636	14,520	739	3,218	357MB

3.2.1 ソースコードデータ

収集されたソースコードは，Codeforces のプログラミングコンテスト参加者が，コンテストの問題に対する回答として提出したものである．ソースコードは， Codeforces がコンパイル環境を用意している任意の言語で記述することができる．また，回答ソースコードが正答であるか，コンパイル可能かどうかは提出の可否に影響しないため，文法的に不完全なソースコードが含まれる場合もある．

3.2.2 提出履歴データ

提出履歴情報には，参加者のレーティング情報や，どの参加者がどの問題にいつ提出したか，提出が正解であったか等の情報が含まれている．データベースの構成は表 1 に示す通りであり，以下では各テーブルの詳細について述べる．

- **Participant** このテーブルには，2016/5/19〜2016/11/15 の期間中 1 度以上 Codeforces で開催されたプログラミングコンテストに参加したユーザーの情報を含む．各参加者情報には表 1 に示す通り 3 種類の項目が含まれる．Codeforces は一意のユーザー ID を提供しておらず，本データセットにおいてはユーザー名を ID としてある．また，レーティングはコンテストごと変化するが，本データに含まれるレーティングは 2016/11/15 時点のものである．

- **ParticipantSubmission** 各参加者が提出したソースコードのうち，本データセットに含まれるものの数を保管しているテーブルである．

- **File** 2016/5/19〜2016/11/15 の期間中に Codeforces に対して提出されたソースコードの情報を収集したテーブルである．このテーブルには，ソースコードデータの総数と同じ 1,644,636 のデータを含む．各提出履歴に与えられた一意の提出 ID と提出対象である問題の ID をもとに対応するソースコードの URL を構築することができ，url カラムに格納されている．

- **Competition** Codeforces 上で開催されたコンテストのうち，Problemset[7]で公開さ

[7]http://codeforces.com/problemset/

れているコンテストの情報を含む．多くのコンテストは 56 程度の問題を含んでいる．コンテストには月に数回行われる定期的なものや，年に一度開催されるコンテスト等複数の種類があり，コンテストに含まれる問題の難易度も様々である．

- **Problem** このテーブルには 3,218 問のプログラミングコンテストの問題に関する情報が含まれている．問題 ID は，問題が掲載されたコンテストの ID と，コンテスト内で問題を識別するためのアルファベットからなる．例えば問題 ID が 601B である場合，ID が 601 であるコンテストで B 問題として出題されたということである．また，問題には得点が設定されており，この得点は多くの場合に問題の難易度が高くなるほど大きくなるようになっている．ただし，問題の得点は問題作成者によって設定された目安である．

- **Acceptance** このテーブルには，Problem テーブルに含まれる問題への提出数に関する統計情報を格納している．接尾辞が sample であるカラムは，File テーブルに情報が格納されている提出に限定した値である．そうでないカラムの値は，2016/11/15 までに行われた全提出に対する提出数や正解数を表している．また，acceptance_rate は問題に対する正答率を表している．

- **ProblemSubmissionStatistics** このテーブルには，Problem テーブルに格納されている問題に対する提出における，提出ソースコードのサイズを格納している．サイズの単位は Byte である．接尾辞が competition であるカラムについては，コンテスト中に提出されたソースコードに限定して統計がとられている．

- **SubmissionDistance** このテーブルには，連続した提出である file_name と next_file 間における編集距離に関する情報が格納されている．levenshtein_distance はソースコードにおけるトークンベースのレーベンシュタイン距離 [7] を表している．接尾辞が node であるカラムの値は，抽象構文木に対する編集距離である．抽象構文木上の編集距離は GumtreeDiff [8] を用いて計算された値である．

3.2.3 統計情報

図 1 は本データセットのソースコードの，言語別提出数の割合を表している．提出されたソースコードのうち，90%は C++によって記述されており，その次は Java の 4%となっている．また，ソースコード平均サイズは 1.2KB であった．本研究では，提出ソースコードにおいて“GNU C++”,“GNUC++11”，“GNU C++14”のいずれかで記述されたものを対象として実験を行った．対象となったソースコードの数は 1,405,734 で，本データセットの 85%を占める．

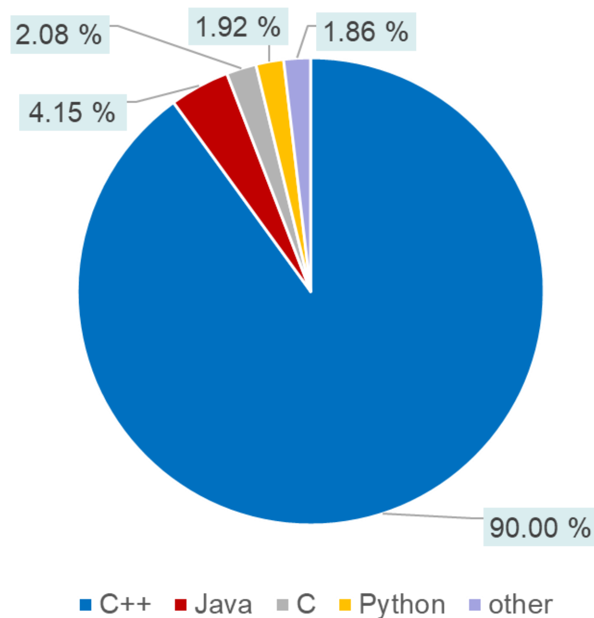

図 1: 提出ソースコードの言語分布

3.3　上級者・初級者の定義

ソースコード編集作業について分析を行う上で，プログラミングコンテスト参加者を上級者と初級者の 2 群に分割して比較調査を行った．具体的には以下のように参加者を分割している．

1. 参加者をレーティングでソート
2. ソートした参加者を人数が等しくなるように 4 分割
3. 4 分割された参加者のうち，最もレーティングが高いグループを上級者，最もレーティングが低いグループを初級者とする

4 分割された参加者のうち，レーティングが中央値付近の 2 群については考慮しない．このとき，上級者・初級者の分布は表 3 のようになった．人数が 2 群で異なるのは，同じレーティングを持つ参加者が境界に複数分布するためである．また，初級者のレーティング最小値が-39 であるが，レーティングの増減に制限はないため，レーティングが負になる場合もある．ただし，レーティングが負である参加者は 1 名のみで，その次に小さい値は 185 であった．初級者のレーティング範囲は-39～1299，上級者は 1573～3367 となる．

表 3: 上級者・初級者の 2 群におけるレーティングの統計情報

		初級者	上級者
レーティング	平均	1171.12	1824.824
	分散	113.62	226.54
	最小値	-39	1573
	中央値	1202	1764
	最大値	1299	3367
人数		3634	3622

3.4 調査結果

プログラミングコンテスト上級者・初級者間におけるソースコードの編集作業における特徴調査のうち，予約語利用頻度の調査とメトリクスの値に関する調査について以下で述べる．Codeforces 内の同一の問題に対して提出されたソースコードの集合に対し，ソースコード特徴量を標準化するなどの処理を施し，調査を行った．

3.4.1 予約語利用頻度の調査

検出対象となった予約語を表 4 に示す．調査内容の説明のため，以下でいくつかの用語を定義する．

$U_p :=$ 問題 p へ提出を行った参加者の集合

$k_{u,p,t} :=$ 参加者 u，問題 p における初回提出ソースコード内の予約語 t の利用数

$\overline{k_{p,\,t}} := \frac{1}{|U_p|}\sum_{x\in U_p} k_{x,\,p,\,t}$(問題 p における全初回提出ソースコード内の予約語 t の利用数の平均)

$\sigma_{p,t} := \sqrt{\frac{1}{|U_p|}\sum_{x\in U_p}(k_{x,\,p,\,t} - \overline{k_{p,t}})^2}$(問題 p における全初回提出ソースコード内の予約語 t の利用数の標準偏差)

$k'_{u,p,t} := \frac{k_{u,p,t} - \overline{k_{p,t}}}{\sigma_{p,t}}$($k_{u,p,t}$を各問題について平均 0，標準偏差 1 となるよう標準化したもの)

$K'_{t,high} := k'_{u,p,t}$のうち，$u$ が上級者に属する集合

$K'_{t,low} := k'_{u,p,t}$のうち，u が初級者に属する集合

表 4: 調査対象の予約語一覧[8]

alignas	decltype	namespace	static_cast
alignof	default	new	struct
and	delete	noexcept	switch
and_eq	do	not	template
asm	double	not_eq	this
auto	dynamic_cast	nullptr	thread_local
bitand	else	operator	throw
bitor	enum	override	true
bool	explicit	or	try
break	export	or_eq	typedef
case	extern	private	typeid
catch	false	protected	typename
char	final	public	union
char16_t	float	register	unsigned
char32_t	for	reinterpret_cast	using
class	friend	requires	virtual
compl	goto	return	void
concept	if	short	volatile
const	inline	signed	wcha_t
constexpr	int	sizeof	while
const_cast	long	static	xor
continue	mutable	static_assert	xor_eq

問題 p ごとに標準化を行う理由としては，問題によってアルゴリズムが異なるため，同一問題内での相対的な利用頻度を調べることで問題に偏りがある場合でも比較が可能になるからである．上記の用語を用いて，”$K'_{t,\,high}$ と $K'_{t,\,low}$ の母平均が等しい”という帰無仮説を立てた t 検定と，Cohen’s d [9] を用いて効果量を測定した．

[8]http://www.campwoodsw.com/sourcemonitor.html

Cohen's d

2 群の平均値の差を表す値として，Cohen's d [9] を利用する．Cohen's d は x_1 と x_2 の 2 群において，各群の標準偏差に対してどの程度平均が異なるかを表す．Cohen's d は以下の式によって定義される．

$$d = \frac{\overline{x_1} - \overline{x_2}}{s}$$

ただし，

$$s := \sqrt{\frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}}$$

$$n_i := |x_i|$$

$$s_i^2 := x_i \text{の分散}$$

表 5 に上級者と初級者の予約語利用頻度を比較した結果を示す．"$K'_{t,\ high}$ と $K'_{t,\ low}$ の母平均が等しい" という帰無仮説を有意水準 5%で棄却した予約語 t について，Cohen's d の降順に掲載した．表の上部ほど，初級者と比較して上級者の利用頻度が高いと言える．上級者の上位予約語に着目すると，typedef や template，class 等のプログラムの構造化を促進する予約語を多く用いている傾向が確認された．上級者は初級者と比較して，ソースコードの抽象度が高いのではないかと考えられる．また，初級者の上位予約語に着目すると，else，break，if 等の分岐に関する予約語を多く用いる傾向にあることが確認された．このことから，初級者は分岐文を多用し，ソースコードが複雑化する傾向があるのではないかと考えられる．

表 5: 利用頻度が高い予約語

t	$\overline{K'_{t,low}}$	$\overline{K'_{t,high}}$	初級者分散	上級者分散	p 値	Cohen's d
typedef	-0.111	0.118	0.896	1.082	0.000e+00	0.228
template	-0.071	0.070	0.778	1.162	0.000e+00	0.140
return	-0.083	0.047	0.881	1.042	0.000e+00	0.131
typename	-0.063	0.066	0.602	1.225	4.822e-291	0.130
while	-0.049	0.029	0.863	1.088	4.466e-115	0.078
operator	-0.033	0.045	0.793	1.156	1.042e-107	0.077
class	-0.037	0.038	0.883	1.100	1.740e-101	0.074
using	-0.038	0.030	0.720	1.270	7.801e-75	0.066

表は次ページに続く

前ページからの続き

t	$\overline{K'_{t,low}}$	$\overline{K'_{t,high}}$	初級者分散	上級者分散	p 値	Cohen's d
continue	-0.029	0.034	0.914	1.043	8.446e-76	0.063
struct	-0.036	0.017	0.800	1.061	7.348e-59	0.055
do	-0.029	0.018	0.564	0.925	5.805e-65	0.053
public	-0.018	0.012	0.633	1.016	2.597e-22	0.032
namespace	-0.025	0.002	0.885	1.106	4.304e-15	0.027
enum	-0.008	0.012	0.179	1.012	8.389e-14	0.025
asm	-0.005	0.008	0.049	0.772	3.056e-09	0.017
private	-0.009	0.003	0.390	0.814	7.058e-08	0.016
typeid	-0.003	0.004	0.010	0.489	1.143e-08	0.014
friend	-0.007	0.003	0.650	0.844	8.956e-05	0.012
decltype	-0.003	0.004	0.180	0.657	3.321e-04	0.009
try	-0.001	0.002	0.011	0.410	1.004e-02	0.006
catch	-0.001	0.002	0.011	0.408	1.050e-02	0.006
extern	-3.413e-05	-1.058e-04	8.354e-04	0.001	1.259e-61	-0.002
switch	-8.421e-04	-6.476e-03	6.406e-01	0.463	2.068e-03	-0.008
case	-3.696e-04	-6.822e-03	6.454e-01	0.436	3.069e-04	-0.009
goto	9.375e-03	-4.003e-03	8.934e-01	0.657	1.993e-07	-0.015
for	3.069e-02	-1.241e-02	1.005e+00	1.023	1.707e-35	-0.043
if	2.283e-02	-3.016e-02	9.862e-01	1.003	8.989e-55	-0.053
break	5.797e-02	-4.005e-02	1.074e+00	0.942	1.018e-183	-0.098
else	8.760e-02	-9.499e-02	1.061e+00	0.915	0.000e+00	-0.185

3.4.2 メトリクスの値に関する調査

調査対象となったメトリクスの一覧と説明を表 6 に示す．SourceMonitor で計測できる 11 種類のメトリクスである．また，メトリクスの一部で用いた循環的複雑度 [10] について以下で補足する．

表 6: 調査対象メトリクス

メトリクス	説明
avg_complexity	各関数の循環的複雑度の平均値
max_complexity	各関数の循環的複雑度の最大値
avg_depth	各関数のネスト深さの平均値
max_depth	各関数のネスト深さの最大値
methods_per_class	クラス当たりのメソッド数
n_classes	クラス数
n_func	関数の数
n_lines	物理行数
n_statements	セミコロンで区切られた論理行数
percent_branch_statements	全体の論理行数に占める分岐文 (if，else，for，while，goto，break，continue，switch，case，default，return を用いた文) の割合
percent_comments	全体の物理行数に占めるコメントの割合

循環的複雑度

McCabe によって提唱された，ソフトウェアの複雑度を示す指標の一種である．循環的複雑度は以下の式によって定義される．

$$M = E - N + 2P$$

ただし，

$M :=$ 循環的複雑度

$E :=$ 制御フローグラフ ($ControlFlowGraph : CFG$) における辺の数

$N := CFG$ における頂点数

$P := CFG$ におけるコンポーネント (結合しているグラフ) の数

循環的複雑度はソースコード中の分岐の数や組み合わせに依存して増加するため，循環的複雑度が高いほどプログラムが複雑であるといえる．

SourceMonitor で計測した結果を上級者と初級者間で比較する上で，用語を定義する．標

準化の計算方法は 3.4.1 節と同様である．

$$U_p := \text{問題} p \text{へ提出を行った参加者の集合}$$

$$m_{u,p,c} := \text{参加者} u\text{，問題} p \text{における初回提出ソースコード内のメトリクス} c \text{の計測値}$$

$$m'_{u,p,c} := m_{u,p,c}\text{を各問題について平均} 0\text{，標準偏差} 1 \text{となるよう標準化したもの}$$

$$M'_{c,high} := m'_{u,p,c}\text{のうち,} u \text{が上級者に属する集合}$$

$$M'_{c,low} := m'_{u,p,c}\text{のうち,} u \text{が初級者に属する集合}$$

各 c ごとに，$M'_{c,high}$，$M'_{c,low}$ の 2 群について t 検定と Cohen's d の算出を行った．いずれのメトリクスについても t 検定において有意水準 5%で有意差が認められた．結果を表 7 に Cohens' d の降順に掲載した．初級者と比較して上級者の計測値が大きかったメトリクスに着目すると，上級者はソースコードの行数が増える傾向が伺える．関数の数やクラス数も初級者と比較して増加する傾向にあり，処理を関数に分離することで行数の増加につながっている可能性がある． 上級者と比較して初級者の計測値が大きかったメトリクスの傾向として，ネストの深さや複雑度の高さが確認された．

表 7: 値が大きいメトリクス

t	$\overline{K'_{t,low}}$	$\overline{K'_{t,high}}$	初級者分散	上級者分散	p 値	Cohen's d
n_statements	-0.139	0.108	0.894	1.078	0.000e+00	0.245
n_func	-0.105	0.089	0.843	1.118	0.000e+00	0.191
n_lines	-0.065	0.043	0.931	1.022	3.037e-220	0.109
methods_per _class	-0.045	0.035	0.567	1.134	1.660e-165	0.084
n_classes	-0.044	0.023	0.768	1.086	8.370e-98	0.069
percent_comments	0.022	-0.035	1.031	0.916	8.580e-61	-0.059
max_depth	0.096	-0.081	1.028	0.971	0.000e+00	-0.179
max_complexity	0.113	-0.100	1.060	0.953	0.000e+00	-0.214
avg_complexity	0.162	-0.137	1.073	0.926	0.000e+00	-0.303
percent_branch _statements	0.195	-0.154	0.993	0.995	0.000e+00	-0.351
avg_depth	0.246	-0.202	1.034	0.944	0.000e+00	-0.457

3.5 本研究への利用

本研究では，与えられたソースコードが，上級者と初級者のどちらに近いか，という観点でソースコードの良否の判定を行う．良否の判定が自動でできるようになると，ソースコードの適切な編集を支援することができる．ソースコードの良否の判定を行うことを考えた場合， ソースコードの構造や読みやすさ，実行時間など様々な基準が考えられるため，判定は難しい．そこで，判定の基準として予約語，メトリクスといったソースコード特徴量が利用できるのではないかと考えた．

調査結果から，上級者は構造に関する記述が，初級者は分岐に関する記述が多いという差異が見られた．上級者・初級者間で差異のある特徴量を意識してソースコードの編集を行えば，良いソースコードとなることが予想される．ただ，上級者のソースコード特徴量のみを利用して編集することは難しい．例えば，初級者には分岐に関する記述が多いという特徴は，あくまで上級者と比較した場合であるため，上級者のソースコードにも分岐に関する記述は当然見られる．ソースコードの一部を切り取れば分岐文が多いソースコードでも，全体として見れば構造化された良いソースコードと言える場合がある．また，ソースコード内での予約語やメトリクスのバランスもあるため，上級者の特徴を意識して編集したソースコードが必ずしも良いソースコードとは限らない．編集後のソースコードに対して，これは良いソースコードである，という判定ができて初めて適切な編集ができたと言える．自動でソースコードの良否を判定できるようになれば，編集の前後での判定の変化によってソースコードの編集が適切かどうかの判断が可能となり，効率的なプログラミング学習につながる．

そこで本研究では，ソースコード特徴量をベクトル化したデータを用いて機械学習によるソースコードの良否の判定を行う．否と判定されたソースコードに関しては，ソースコード特徴量を用いた修正の指針を提示する． 機械学習を用いることで，自動で判定を行うことができる．機械学習に用いるデータとして上級者・初級者間で差異のあるソースコード特徴量を使用することによって，定量的な判定が可能となる．手法に関しては，4 章で具体的に述べる．

4　提案手法

本研究では，ソースコード特徴量を用いた機械学習によってソースコードの良否の判定を行った．本章では，判定の手法について説明する．

4.1　目的

ソースコード編集作業は，開発に際して必然的に行われる作業である．そのため，コンピュータサイエンスにおいてソースコードの編集に関する研究は数多く行われている [4,5]．

ソースコードの良否の判定が自動でできるようになると，ソースコードの適切な編集を支援することができる．編集後にソースコードが良くなったと判定できれば，編集は適切だったと言える．良否の判定に用いた基準を利用すれば，初級者は良いソースコードを参考に学習を進めることができるようになり，効率的な技術力の向上を期待できる．しかし，ソースコードの良否を行うことを考えた場合，構造や読みやすさ，実行時間など様々な基準が考えられるため，一般的な判断は難しい．

本研究では，プログラミングコンテストにおける上級者・初級者間のソースコード特徴量を用いた定量的な基準によってソースコードの良否を判定することで上記の課題に対して解決を図る．本章では，ソースコード特徴量を用いた，機械学習によるソースコードの良否の判定の流れと修正の指針の提示について説明する．ソースコードの良否を判定できれば，プログラミングの学習に役立つ．否と判定されたソースコードを修正し，良と判定されるようになれば，書き手は自分の成長を実感できる．また，ソースコード特徴量を用いた修正のヒントが得ることができれば，効率的に編集を行うことができる．

4.2　ソースコードの良否の判定

本研究では，3 章で述べたオンラインで公開しているデータセットを MySQL 上に保存し，データベースを管理した．ソースコード特徴量を用いた機械学習によるソースコードの良否の判定を行うための流れを図 2 に示す．判定を行うまでの流れは大きく 3 つの段階に分けることができる．まず，プログラミングコンテストにおいて提出されたソースコードを収集し，レーティングによって上級者と初級者に分類する．上級者・初級者の定義は 3.3 節と同様である．次に，分類したソースコードに対して予約語利用頻度やメトリクス値などのソースコード特徴量を取得し，機械学習用の説明変数とする．目的変数はソースコードの良否であり，上級者が書いたソースコードを「良」，初級者が書いたソースコードを「否」と定義する．これらをベクトル化し，機械学習用のプログラムを用いて学習させる．最後に，学習に用いられていない新規のソースコードのソースコード特徴量をベクトル化して入力し，良否の判定を出力する．

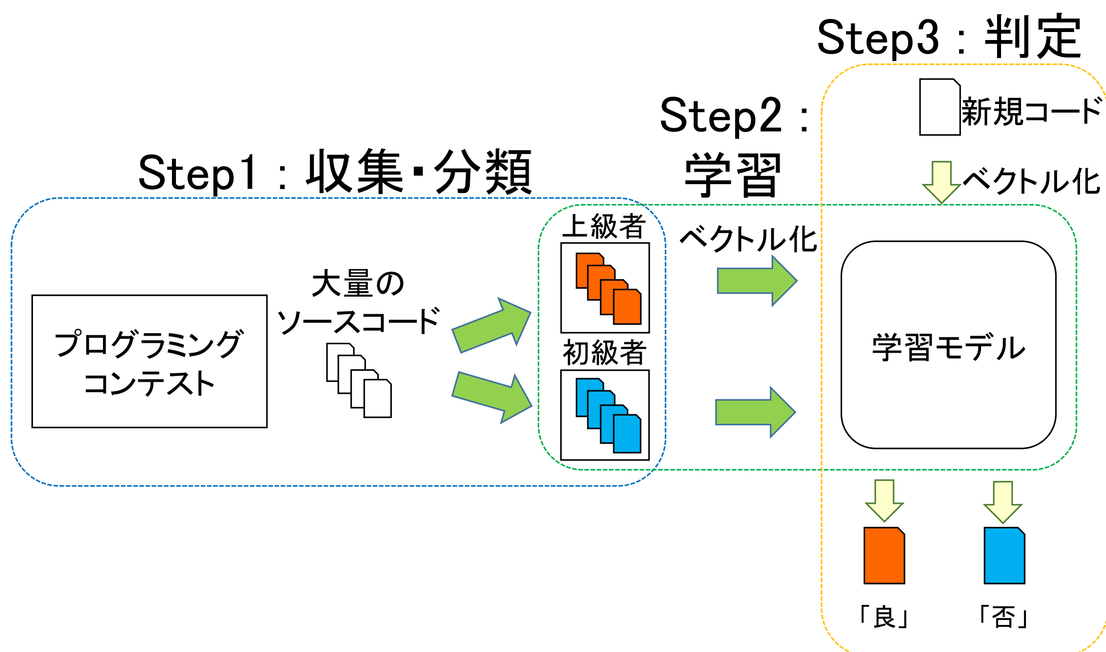

図 2: ソースコードの良否の判定の流れ

3 つの段階それぞれについて，以下で詳細を述べる．

STEP1: 収集・分類

本研究においては，3 章で述べた，オンラインで公開されたデータセットを用いる．プログラミングコンテストシステム Codeforces で 2016/5/19〜2016/11/15 の期間中に提出されたソースコード数は 1,644,636 であり，そのうち C++で記述された 1,405,734 個のソースコードが分類の対象である．表 1 で示したように，データセット内では，File テーブル上にソースコードのファイル名と提出者の名前が存在し，Participant テーブル上に参加者の名前とレーティングが存在するので，これらを組み合わせればファイル名からレーティングを取得し，上級者・初級者に分類することが可能となる．ソースコードを上級者と初級者に分類した結果，上級者が提出したソースコード数は 544,290 であり，初級者が提出したソースコード数は 332,863 であった．これらのソースコードに対してソースコード特徴量を取得し，学習を行う． なお，上級者にも初級者にも該当しないソースコードは本研究では学習の対象

外とする．

STEP2: 学習

本研究においては，機械学習の説明変数はソースコード特徴量，目的変数はソースコードの良否である．機械学習を行うための準備として，対象となった全てのソースコードに対して特徴ベクトルを作成する．学習させる値は，ソースコードごとのソースコード特徴量とソースコードの良否である．ソースコード特徴量に関しては，表 5 及び表 7 に示した予約語とメトリクス値を使用する．ソースコードの良否に関しては，上級者が書いたソースコードを「良」，初級者が書いたソースコードを「否」と定義し，ベクトルに追加する．作成したベクトルを機械学習用プログラムに学習させる．以下で，具体的な説明を行う．

まずは，ソースコード特徴量について説明する．予約語に関しては，ソースコードに対して字句解析を行い，表 5 で示した 29 種類の予約語の利用頻度を取得する．メトリクスに関しては，SourceMonitor を用いて，表 7 で示した 11 種類のメトリクスの値を取得する．これらを組み合わせて，1 つのソースコードにつき 40 次元のベクトルを作成する．図 3 にベクトルの一部を掲載する．1 行が 1 つのソースコードに対応している．

次に良否について説明する．上級者が書いたソースコードを「良」，初級者が書いたソースコードを「否」と定義し，それぞれ値として 1，-1 を与える．ソースコード特徴量のベクトルにこの値を追加し，41 次元のベクトルを作成する．図 4 にベクトルの一部を掲載する．”skill” という変数名で目的変数を追加している．

学習の対象となる全てのソースコードに対して図 4 のようなベクトル化を行い，ソースコード特徴量に対する良否を機械学習プログラムを用いて学習を行う．本研究では，python の機械学習ライブラリである scikit-learn を用いて機械学習用のプログラムを記述した． 使用したアルゴリズムは，決定木と SVM である．両アルゴリズム共に，分類精度は 90%ほどである．

asm	break	case	catch	class	continue
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	2	0	0	0	0
0	4	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

・・・	max_depth	avg_depth	avg_complexity	n_func
	2	0.44	2	3
	3	0.47	3.5	2
	4	0.6	2.5	2
	3	0.37	2	2
	4	0.61	3	2
	4	0.61	3	2
	3	0.94	7	1
	3	1.19	13	1
	3	1.42	25	1
	3	1.42	25	1
	3	1.44	25	1
	4	1.67	6	1

図 3: 説明変数のベクトル

asm	break	case	catch	class	・・・
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	2	0	0	0	
0	4	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	

max_complexity	max_depth	avg_depth	avg_complexity	n_func	skill
3	2	0.44	2	3	1
5	3	0.47	3.5	2	1
3	4	0.6	2.5	2	1
2	3	0.37	2	2	1
4	4	0.61	3	2	1
4	4	0.61	3	2	1
7	3	0.94	7	1	-1
13	3	1.19	13	1	-1
25	3	1.42	25	1	-1
25	3	1.42	25	1	-1
25	3	1.44	25	1	-1
6	4	1.67	6	1	-1

図 4: 目的変数”skill” を追加したベクトル

STEP3: 判定

良否の判定を行いたいソースコードに対し，4.2 節と同様の手順を用いて図 3 のようなソースコード特徴量による 40 次元のベクトルを作成する．学習を行った機械学習プログラムに対し，ベクトルを入力すると，図 5 に示すように,「良」または「否」のいずれかが出力される．

評価結果を出力
「良」

図 5: 良否の判定の出力

4.3　修正の指針の提示

入力ソースコードに対して「否」と判定された場合，ソースコードへ修正を行い,「良」と判定されるようになれば，編集は適切であったと言え，書き手は自分の成長を実感できる．

表 5 や表 7 で示された上級者・初級者間で差異のある特徴量を意識してソースコードの編集を行えば，良いソースコードとなることが予想される．ただ，上級者に多いソースコード特徴量のみを利用し，初級者に多いソースコード特徴量は利用しない，といった単純な修正では良いソースコードにすることは難しい．例えば，初級者には分岐に関する記述が多いという特徴は，あくまで上級者と比較した場合であるため，上級者のソースコードにも分岐に関する記述は当然見られる．ソースコードの一部を切り取れば分岐文が多いソースコードでも，全体として見れば構造化された良いソースコードと言える場合がある．また，ソースコード内での予約語やメトリクスのバランスもあるため，上級者の特徴量を利用して修正したソースコードが必ずしも良いソースコードとは限らない．入力ソースコードの現在の構造を踏まえたうえで，どのように修正を行うのがよいかということを考える必要がある．

そこで，単一のソースコード内における構造の特徴に関する調査を行い，上級者・初級者の差が大きい特徴を修正に利用することを考えた．具体的には，単一のソースコード内における予約語利用頻度を上級者，初級者それぞれで調査を行う．調査結果を基に，入力されたソースコードに対する修正の指針として，初級者と比較して上級者の利用頻度が高い予約語がソースコード内で使用されていなければ，使用するように提示する．単一のソースコードにおける予約語の利用頻度の調査結果を説明するため，以下で用語を定義する．

$$F_{high} := \text{上級者が提出したソースコードの集合}$$

$$F_{low} := \text{初級者が提出したソースコードの集合}$$

$F :=$ ソースコード f が上級者が提出したものであれば F_{high}，初級者が提出したものであれば F_{low}

$k_{f,t} :=$ ソースコード f における予約語 t 利用数

$k_{f,max} :=$ ソースコード f における予約語の利用数の最大値

$k_{f,min} :=$ ソースコード f における予約語の利用数の最小値

$\tilde{k_{f,t}} := \dfrac{2(k_{f,t} - k_{f,min})}{k_{f,max} - k_{f,min}} - 1$(ソースコード f における予約語 t の利用数を -1 から 1 の値に正規化したもの)

$\overline{\tilde{k_t}} := \dfrac{1}{|F|} \sum_{x \in F} \tilde{k_{x,t}}$(予約語 t の利用数を正規化した値の平均)

$\tilde{k_{t,high}} := \overline{\tilde{k_t}}$のうち上級者を起源とする値

$\tilde{k_{t,low}} := \overline{\tilde{k_t}}$のうち初級者を起源とする値

単一のソースコード内での利用頻度が高い予約語を，正規化後の上級者と初級者の値の差が大きい順に表 8 に掲載する．3 章では，ある予約語 t に対し，同一の問題に提出されたソースコードの集合における相対的な予約語利用頻度を調査していた．それに対し，表 8 に示した値は，単一のソースコード内において，ある予約語 t の，利用された全ての予約語における相対的な予約語利用頻度を調査した結果である．他のソースコードでの利用数は使用せず，単一のソースコード内のみで利用数の正規化を行った．これは，単一のソースコードを独立して見た場合の構造の特徴を調べるためである．例えば”for”のように，表 5 における上級者の利用頻度が低くても，表 8 における上級者と初級者の値の差が正になる場合がある．これは，同一の処理を行うソースコードにおける”for”の利用回数は，上級者は初級者と比較して少ないが，単一のソースコード内で利用されているすべての予約語に対する”for”の利用回数は，上級者は初級者と比較して多いということを意味している．この結果からも，表 5 において表の上部の予約語を使用し，下部の予約語は使用しないという単純な修正では良いソースコードにすることが難しいことが分かる．

表 8: 単一ソースコード内において上級者の利用頻度が高い予約語

t	$\tilde{k_{t,high}}$	$\tilde{k_{t,low}}$	$\tilde{K_{t,high}} - \tilde{K_{t,low}}$
return	-0.0363756	-0.1662309	0.1298554
while	-0.6733181	-0.7549128	0.0758102
template	-0.9287975	-0.9848658	0.0560683
for	0.1292167	0.0822340	0.0469827
continue	-0.9180995	-0.9638606	0.0457610
typename	-0.9528854	-0.9919364	0.0390510
struct	-0.9537283	-0.9850745	0.0313462
class	-0.9559435	-0.9871114	0.0311679
operator	-0.9669999	-0.9944450	0.0274451

単一のソースコード内で上級者の利用頻度が高い予約語は，上級者の書く良いソースコードにおいて，全ての予約語の中で利用される割合が高いということである．入力ソースコードにこのような予約語が存在しない場合，上級者の利用頻度が高い予約語を利用することで，良いソースコードに近づく可能性が高まるのではないかと考えた．本研究では，上級者と初級者の値の差が 0.2 以上となる予約語を初級者と比較して上級者の利用頻度が高い予約語と定め，表 8 に掲載した．入力ソースコード内で該当する予約語が使用されていなければ，良否の判定結果とともに図 6 で示すように予約語の利用を促す．

```
"class" is unused. why don't you use it?
"continue" is unused. why don't you use it?
"for" is unused. why don't you use it?
"operator" is unused. why don't you use it?
"struct" is unused. why don't you use it?
"template" is unused. why don't you use it?
"typename" is unused. why don't you use it?
"while" is unused. why don't you use it?
```

図 6: 修正の指針の提示

4.4　実験結果

4.2 節で行ったソースコードの良否の判定の精度と，4.3 節で示した特徴量を参考に編集を行った後の判定結果の変化について述べる．

4.4.1　学習精度

本研究において，上級者と分類されたソースコード数は 544,290 であり，初級者と分類されたソースコード数は 332,863 である．合計 877,153 のデータに対し，機械学習の精度を以下のような手順で求めた．

1. 上級者・初級者全てのソースコードに対し，4.2 節で述べた 41 次元のベクトルを作成する
2. 1 で作成したベクトルの中からランダムで 1 割抽出し，テストデータとする
3. 残りの 9 割のベクトルを学習データとし，学習モデルを作成する
4. テストデータの説明変数を学習モデルに入力する
5. 入力ソースコードが上級者の場合は「良」，初級者の場合は「否」と出力されれば正解，そうでなければ不正解とする

機械学習の精度の評価指標として，上級者・初級者それぞれについて適合率，再現率，F 値を算出した．以下で，これらの用語について説明する．

適合率 (precision)

予測結果の中に，どの程度正解が含まれるかを示す指標である．本研究においては，良と予測したデータのうち，実際に上級者のソースコードであるものの割合．または，否と予測した予測したデータのうち，実際に初級者のソースコードであるものの割合．

再現率 (recall)

正解のうち，どの程度が予測と合致するかを示す指標である．本研究においては，実際に上級者のソースコードであるもののうち，良と予測されたものの割合．または，実際に初級者のソースコードであるもののうち，否と予測されたものの割る．

F 値 (F-measure)

適合率と再現率の総合的な評価を表す指標である．適合率と再現率の調和平均．

機械学習の2値分類では，評価指標の値を正の場合のみ，つまり本研究における上級者のみで算出を行う場合もあるが，本研究においては上級者・初級者の両方で算出を行う．理由としては，上級者の方がデータ数が多いため，ランダムに抽出したテストデータにも偏りが生じ，判定の能力を表す値として不適切な場合があるからである．具体例を図7に示す．

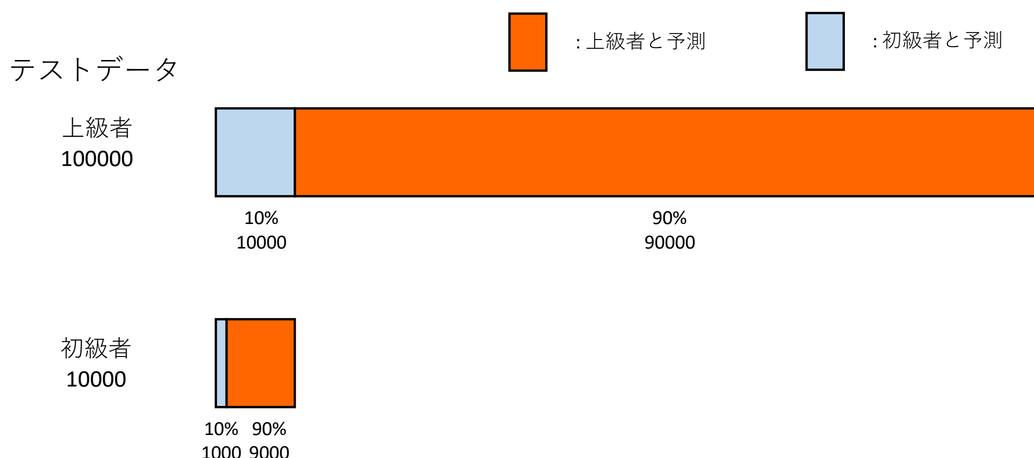

図 7: 評価指標が不適切となる例

上級者のテストデータ数が100000，初級者のテストデータ数が10000であるとし，図7に示す割合で予測を行ったとする．上級者について適合率，再現率，F値は以下のように算出できる．

$$
\begin{aligned}
precision &= \frac{90000}{90000 + 9000} \\
&= 0.909 \\
recall &= \frac{90000}{100000} \\
&= 0.900 \\
F\text{ 値} &= \frac{2 \times pricision \times recall}{recall + precision} \\
&= 0.905
\end{aligned}
$$

この結果を見ると，3つの評価指標全てで90%以上となるため，判定を行う能力が高いように思える．しかし，初級者について同様に算出すると以下のような値になる．

$$
\begin{aligned}
precision &= \frac{1000}{10000+1000} \\
&= 0.091 \\
recall &= \frac{1000}{10000} \\
&= 0.1 \\
F\,値 &= 0.095
\end{aligned}
$$

初級者に関しては，評価指標全てが10%以下となり，初級者に関する判定の信用度は非常に低いことが分かる．したがって，上級者・初級者それぞれについて評価指標の算出を行うことで，判定の能力を適切に評価することができる．

本研究で作成した機械学習プログラムにおける，上級者・初級者それぞれについて算出した適合率，再現率，F 値を表 9 に示す．使用したアルゴリズムは，決定木と SVM である．両アルゴリズム共に，評価指標の値はおおよそ 90%に近い値が多かった．この結果に対する考察を以下で述べる．

ソースコードを上級者・初級者に分類する際に用いたのはレーティングのみであり，ソースコードの構造は無視している．Codeforces においては，制限時間内に問題を解くことができれば得点がつき，終了時の得点が高いほどレーティングが高くなる．つまり，入力に対して正しい出力ができるソースコードを記述することさえできれば，レーティングは上がっていくため，レーティングが高い上級者が提出したソースコードでも，構造を見ると初級者の特徴に近い場合がある．上級者が提出した全てのソースコードの構造が表 5 や表 7，表 8 に示すような上級者の構造に近いわけではない．判定の能力が高いと，テストデータとして構造が初級者に近いソースコードが入力された場合は，提出者が上級者であっても「否」という判定が出力される．初級者に関しても同様のことが言える．したがって，テストデータの判定においてはある程度の誤りが許容できる．実際には，90%という値以上の分類性能を持っている可能性があり，判定の能力は十分にあると言える．

表 9: 機械学習の評価指標

	決定木			SVM		
	適合率	再現率	F 値	適合率	再現率	F 値
上級者	0.918	0.906	0.912	0.940	0.900	0.920
初級者	0.850	0.869	0.860	0.846	0.906	0.875

4.4.2　修正後の良否の判定の変化

本研究においては，ソースコードに対する修正が適切かどうかについて表 10 のように定め，判定が「否」から「良」に変化した場合についてのみ修正が適切であるとした．出力される良否は，入力されたソースコードのソースコード特徴量が上級者と初級者のどちらに近いか，という判定基準で定められる．しかし，どれだけ近いかということは出力結果から判断ができない．例えば，修正前後で判定が「否」のままの場合，修正によって上級者に近づきはしたが依然として初級者に近いのか，より上級者から遠ざかっているのかということは不明である．修正前後で「良」である場合も同様である.「良」から「否」に変化した場合は明らかに修正は不適切である．したがって,「否」と判定されたソースコードに対して表 8 に示した予約語を用いて修正を行い,「良」と判定されるようになれば，修正は適切であったと言うことができる．

表 10: 良否の判定の変化における修正の適切さ

判定	修正前	良		否	
	修正後	良	否	良	否
修正の適切さ		不明	不適切	適切	不明

4.2 節の機械学習プログラムに入力されたソースコードに対し，良否の判定結果とともに，表 8 に示した予約語が存在しない場合，図 6 のように出力する．本研究においては，出力された予約語を用いて行った修正が適切かどうかについて，以下の手順で実験を行った．

1. 4.2 節で作成した機械学習プログラムにソースコードを入力した結果,「否」と判定されたソースコードを修正対象とする
2. 図 6 において出力された予約語を用いてソースコードの修正を行う
3. 再び 4.2 節で作成した機械学習プログラムにソースコードを入力する
4. 判定が「良」であれば，2 で行った修正は適切であると判断する

本研究では,「否」と判定されたソースコードを 2 種類用意した．一方は Codeforces における提出ソースコードのうちデータセットに含まれないソースコードであり，もう一方は Codeforces とは無関係のソースコードである．これらのソースコードについて，修正前後における良否の判定の変化を調査した．図 8，図 9 にそれぞれ修正前後のソースコードのうち一部を示す．修正前後において，ソースコードの処理内容は同じである．具体的な修正の内

容として，図 8 では，”class”，”while”を新たに使用して修正を行った．図 9 では，”class”を新たに使用して修正を行った．その結果，図 8 では SVM を用いて作成した機械学習プログラムにおいて，図 9 では決定木を用いて作成した機械学習プログラムにおいて，それぞれ判定が「否」から「良」に変わることが確認でき，修正は適切であったという結果が得られた．このことから，表 8 に示した予約語を用いて修正の指針を与えることができるということが判明した．

修正前

```
int main() {
	(中略)
	fl(n) {
		cin >> a[i];
		if (i > 0 && counter == 0) {
			if (a[i] < a[i - 1] && active == false) {
				start = i - 1;
				active = true;
			}
			else if (active == true && (a[i] > a[i - 1] || i == n - 1)) {
				counter = 1;
				end = (a[i] > a[i - 1]) ? i - 1 : i;
				active = false;
				swap(a[start], a[end]);
				if (start - 1 >= 0) {
					if (a[start - 1] > a[start]) { result = false; }
				}
				if (end + 1 < n) {
					if (a[end] > a[end + 1]) { result = false; }
				}
			}
		}
		else if (counter == 1) {
			if (a[i] < a[i - 1]) {
				result = false;
			}
		}
	}
	if (result == false) { cout << "no"; return 0; }
	if (start == 1 && end == 1) {
	cout << "yes¥n1 1";
	}
	else {
	cout << "yes¥n" << start + 1 << " " << end + 1;
	}
}
```

修正後

```
class swapInfo {(中略)};
inline ll swapInfo::getStart() {(中略)}
inline ll swapInfo::getEnd() {(中略)}
inline bool swapInfo::getIsStart() {(中略)}
inline bool swapInfo::getIsSwapped() {(中略)}
inline void swapInfo::setStart(ll x) {(中略)}
inline void swapInfo::setEnd(ll x) {(中略)}
inline void swapInfo::setIsStart(bool x) {(中略)}
inline void swapInfo::setIsSwapped(bool x) {(中略)}
int main() {
(中略)
	while (!info.getIsStart() && index < n) {
		cin >> a[index];
		if (a[index] < a[index - 1]) {
			info.setStart(index - 1);
			info.setIsStart(true);
			start = info.getStart();
		}
		index++;
	}
	if (!info.getIsStart()) {
		cout << "yes¥n1 1";
		return 0;
	}
	while (!info.getIsSwapped()) {
		if (index < n)
			cin >> a[index];
		else
			index--;
		if (a[index] > a[index - 1] || index >= n - 1) {
			info.setIsSwapped(true);
			info.setEnd((a[index] > a[index - 1]) ? index - 1 : index);
			end = info.getEnd();
			swap(a[start], a[end]);
		}
		index++;
	}
	if ((start >= 1 && a[start - 1] > a[start]) || (end < n - 1 && a[end] > a[end + 1]))
	{
		cout << "no";
		return 0;
	}
	while (index < n) {
		cin >> a[index];
		if (a[index] < a[index - 1]) {
			cout << "no";
			return 0;
		}
		index++;
	}
	cout << "yes¥n" << start + 1 << " " << end + 1;
	return 0;
}
```

図 8: ”class”，”while”を新たに用いた修正例

修正前

```
int main(void){
	(中略)
	while (d < 21){
		for (e = 1; e < u; e++){
			printf(" %d番目の相手の番です。 ¥n", e);
			(中略)
			if (d == 21){
				flag = e;
				break;
			}
		}
		if (d == 21)
			break;
		printf(" あなたの番(%d番目)です。¥n", u);
		(中略)
		if (d == 21){
			flag = u;
			break;
		}
		for (e = u + 1 ; e <= N; e++){
			printf(" %d番目の相手の番です。 ¥n", e);
			(中略)
			if (d == 21){
				flag = e;
				break;
			}
		}
	}
	if (flag == u){
	printf(" あなたの敗北です。¥n");
	return 1;
	}
	else{
	printf(" 相手%dの敗北です。¥n",flag);
	return 1;
	}
	return 0;
}
```

修正後

```
using namespace std;
class member {(中略)};
class master {(中略)};
inline int master::getNumberOfPlayer() {(中略)}
inline int master::getNowDeclaredValue() {(中略)}
inline int master::getNowDeclaringMember() {(中略)}
inline vector<member> master::getParticipants() {(中略)}
inline void master::setNumberOfPlayer(int x) {(中略)}
inline void master::setNowDeclaredValue(int x) {(中略)}
inline void master::setNowDeclaringMember(int x) {(中略)}
inline void master::setParticipants(member player) {(中略)}
inline bool member::getIsPlayer() {(中略)}
inline int member::getOrder() {(中略)}
inline void member::setIsPlayer(bool x) {(中略)}
inline void member::setOrder(int x) {(中略)}
inline int member::declare(int value) {(中略)}
int main(void){
	(中略)
	while (1) {
		int value = gameMaster.getNowDeclaredValue();
		int member = gameMaster.getNowDeclaringMember();
		int order = gameMaster.getParticipants().at(member).getOrder();

		cout << order << "番目の番です." << endl;
		int dValue = gameMaster.getParticipants().at(member).declare(value);
		gameMaster.setNowDeclaredValue(dValue);

		if (gameMaster.getNowDeclaredValue() >= 21) {
		if (gameMaster.getParticipants().at(member).getIsPlayer()) {
		cout << "あなたの敗北です" << endl;
		}
		else {
		cout << order << "番目の敗北です" << endl;
		}
		return 0;
		}

		int nextMember = (member + 1) % gameMaster.getNumberOfPlayer();
		gameMaster.setNowDeclaringMember(nextMember);
		}
}
```

図 9: ”class”を新たに用いた修正例

5　まとめ

本研究では，ソースコード特徴量を用いた，機械学習による良否の判定を行う手法と，ソースコードの修正の指針を提示する手法を提案した．

評価実験として，否と判定されたソースコードを，提示されたソースコード特徴量を用いて修正し，改めて入力ソースコードとしたところ，良と判定される場合があった．したがって，ソースコード特徴量はソースコードの適切な編集への指針となることが判明し，ソースコードの良否の判定を利用することは，プログラミングの学習を支援することにつながると言える．

今後の研究課題として，以下が挙げられる．

- 判定の精度を向上させるために，新たにメトリクスや，コードクローンの数やタイプを特徴ベクトルとして追加することなども考えられる．また，上級者が提出したソースコードのうち，初級者に近いと判定され得るソースコードによる判定ミスをなくすため，テストデータにおいて異なる判定がされたソースコードの目的変数を動的に変更して学習を行うことが考えられる．
- プログラミングコンテストにおいては，予めマクロを自分で用意して利用する場合がある．ソースコードの字句解析を行う際，マクロによって名前が変更された予約語は検出の対象にならないため，特徴ベクトルの作成の際に予約語利用回数に数えられない．そのため，利用しているはずの予約語を利用するように提示されてしまう場合がある．マクロ部分の処理を適切に行うことによってこのような事態は防ぐことができると考えられる．
- 修正の提示は，利用されているかいないかという基準で行っている．全体の構造を加味し，利用されているものに対しても頻度に言及する指針を提示することができれば，より適切な編集につなげることができる．
- 本研究における一連の流れを 1 つにまとめたツールの作成が考えられる．ソースコード特徴量の取得と機械学習は別の方法であるため，手動で行っていた部分も多い．ソースコードを入力し，出力までの全ての過程を自動化できるようになれば，一般的なツールとして本研究の成果を扱うことが可能となる．

謝辞

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 教授には研究及び研究室での生活について適切な御指導及び御助言を賜りました．心より深く感謝しております．

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 准教授には研究の方針について直接の御指導及び御助言をして頂きました．本論文の完成は松下准教授のおかげであると，心より深く感謝しております．

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 神田哲也 助教には，研究において適切な御助言を賜りました．心より深く感謝しております．

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 伊藤 薫 氏には，研究の進め方や論文の書き方の御指導にとどまらず，様々な場面で適切な御助言をして頂きました．心より深く感謝しております．

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 嶋利 一真 氏には，研究の内容や進捗に関して，多くの場面で適切な御助言をして頂きました．心より深く感謝しております．

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 横井 一輝 氏には，研究において様々な場面で適切な御助言を賜りました．心より深く感謝しております．

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 徳井 翔梧 氏には，研究を進めていく過程で，適切な御助言をして頂きました．心より深く感謝しております．

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 藤原 裕士 氏には，字句解析のためのツールを提供して頂くなど，研究において不可欠な御支援をして頂きました．心より深く感謝しております．

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 石田 直人 氏には，本論文の執筆にあたって様々な御支援を賜りました．心より深く感謝しております．

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 白木 秀弥 氏には，本論文の執筆にあたって適切な御助言を賜りました．心より深く感謝しております．

最後に，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は，本論文の執筆にあたって様々な場面で支えて頂きました．皆様のおかげで本論文を完成させることができました．心より深く感謝しております．

参考文献

[1] 堤祥吾. プログラミングコンテスト初級者・上級者間におけるソースコード特徴量の比較, 大阪大学大学院情報科学研究科修士論文, 2018.

[2] Arpad E Elo. *The rating of chessplayers, past and present.* Arco Pub., 1978.

[3] Mikhail Mirzayanov. Codeforces rating system. `http://codeforces.com/blog/entry/102`, 2010.

[4] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proc. of ICSE 2012*, pp. 313, 2012.

[5] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. Discovering bug patterns in javascript. *ACM SIGCSE Bulletin*, pp. 144156, 2016.

[6] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proc. of ICSE 2015*, pp. 913923, 2015.

[7] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10*, pp. 707710, 1966.

[8] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proc. of ASE 2014*, pp. 313324, 2014.

[9] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences, 2nd edition.* Lawrence Erlbaum Associates, 1988.

[10] Thomas J McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, Vol. 2, No. 4, pp. 308320, 1976.