Vol.2021-SE-207 No.7
2021/3/1

解答固有のソースコード片に着目したプログラミング試験解答からの盗用検出手法の提案

砂田 翼[1]　石尾 隆[1]　新田 章太[2]　松本 健一[1]

概要：ソフトウェア開発者の採用活動において，志望者の技術力を評価するためにプログラミング試験を実施する企業が増加している．遠隔にいる志望者に対してオンラインで実施するプログラミング試験では，志望者が他者の解答をそのまま，あるいは改変して提出する盗用行為が対面での試験よりも容易であり，志望者の技術力を正しく評価できないリスクがある．そのため，提出された多数の解答から互いに類似しているソースコードの組を盗用の可能性ありとして検出する手法が求められており，従来より，最長共通部分列の計算などの類似度計算の手法が盗用検出として活用されている．本研究では，そのような類似度の 1 つである N-gram の Jaccard 係数に加えて，少数の解答にのみ出現する N-gram に重みを付けたコサイン類似度を組み合わせて，ソースコードが類似しているだけでなく，それらの解答にのみ固有の表現が類似しているような解答者の組を，盗用の可能性が高いものとして抽出する手法を提案する．提案手法の性能を評価するために，公開データセット SOCO を用いた実験を行った．その結果，提案手法は既存ツール JPlag と比較して F-measure，Averege Presicion の 2 つの評価指標で既存ツールよりも高い性能を達成した．また，ある企業の採用試験のデータセットに対して提案手法を適用した結果，企業の担当者が解答時間に基づいて疑わしいと考えた受験者の 42%は提案手法の結果と一致したほか，解答時間だけでは見逃していたと考えられる事例を検出できることを確認した．

1. はじめに

システムエンジニアやプログラマ等の優秀な IT 人材を確保するために，採用試験においてプログラミング試験を導入し，志望者のコーディングの能力を測る企業が増加している．従来の入社試験では，志望者を 1 つの会場に集めて会場に用意した環境でプログラミング試験を実施していたが，志望者によっては移動時間や費用が負担となることや，会場に用意されている開発環境が普段の開発環境と異なっている志望者が本来のパフォーマンスを発揮できないなどの課題があった．そこで移動の負担を無くし，受験者が普段使用している環境で試験を受けることのできるオンライン試験が行われる機会が増加している．しかしオンライン試験では，試験監督者が直接見ていない場所で試験を行うため，従来の会場で行う試験よりも不正行為が発生しやすい環境となっている．Marais ら [1] は，大学のプログラミングの課題において，e ラーニング環境で行われる遠隔教育において不正行為の発生が深刻な問題であると報告している．このような不正行為が発生すると，志望者のコーディング能力を正確に測ることができないため，不正行為への対策が必要となる．

本研究では，不正行為のうち，他人のソースコードを盗用や流用して解答を作成する（以下，盗用と呼ぶ）行為に着目する．盗用検出として，目視でソースコードを確認する方法もあるが，多数のソースコードを目視で確認するのは時間がかかる．そこで盗用の検出にかかる時間を削減するために，類似しているソースコードを自動検出するツールが提案されている [2], [3], [4], [5]．盗用の際には，ソースコードのレイアウトの変更やコメントの追加・削除，識別子名の変更などの偽装が行われることが知られており [6], [7], [8]，各ツールはソースコードの比較前にコメントの除去，識別子名正規化などの前処理を行っている．一方で，全員が同じ問題を解くプログラミング試験では，誰もが書きうるようなソースコードが偶然類似してしまうことがあり，それが誤検出につながっている．

本研究では，少数の解答にのみ出現するソースコードは，誰もが書きうるようなソースコードではないと仮定し，既存のソースコードの類似度の 1 つである N-gram 集合の一致度合いに，少数の解答にのみ出現する N-gram に重みを付けたコサイン類似度を組み合わせることで，プログラミング試験の解答から盗用可能性のあるプログラムの組を検

[1] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
[2] 株式会社ギブリー
Givery, Inc.

出する手法を提案する．提案手法の有効性を確かめるために，公開データセット SOCO [9] と企業の採用試験データセットで，検出手法の評価と既存盗用検出ツールとの比較を行なう．

本論文の構成は次のとおりである．2 章ではカンニング検出手法の背景について述べる．3 章では本研究で提案する盗用検出手法について述べる．4 章では公開データセットで行った評価実験について述べる．5 章では企業の採用試験で使用されたデータセットに対して提案手法を適用した結果を述べる．最後に 6 章で，まとめと今後の課題について述べる．

2. 背景：プログラミング試験における盗用検出

プログラミング試験における盗用検出は一般的にソースコードの類似性から検出を行なっている．盗用検出の代表的なツールとして，広く利用可能なものには JPlag[*1] と MOSS[*2] がある．JPlag は Flores ら [2] が開発したツールで，現在公開されている盗用検出ツールで最も使用されている盗用検出ツールの 1 つである [10]．このツールは，ソースコードをトークン列に変換し，最長共通部分列を使用して 2 つのソースコードの類似性を計測し，類似しているソースコードの組の検出を行う．MOSS は Bowyer ら [11] が開発した盗用検出ツールである．このツールもソースコードをトークン列に変換し，トークンの一致度合いと行単位での一致度合いから 2 つのソースコードの類似性を計測し類似しているソースコードの組の検出を行う．

紹介した 2 つの盗用検出ツールを始めとした一般的な盗用検出ツールは，ソースコードの類似性を計測して類似度を求め，類似度の高いソースコードの組を検出する方法が採用されている．しかし同じ課題を同じプログラミング言語で解答している為に盗用を行なっていない受験者間でも問題を解くアルゴリズムが類似した場合に誤検出される可能性があることが知られている．Christian ら [12] は大学のプログラミング課題において，授業で配布された資料に含まれているコード片の一致が多くの学生で起こると仮定し，多くの学生に一致するコード片を除いた後で類似しているソースコードの組を検出する方法を提案している．また Mariani ら [13] は，大学のプログラミング課題において授業資料を参考に課題を解いた場合の模範解答を用意し，模範解答と一致するコード片を除いた後で類似ソースコードの組を検出することで従来方法よりも検出精度が向上したことを報告している．しかしこれらの方法では，授業で配布された資料などのベースとなるソースコードが存在することが前提であり，そのようなソースコードが存在しないデータに対しては適用できない．Ohno ら [14], [15] はコーディングスタイルに着目して，いくつかのコーディングスタイルをあらかじめ定義して，学生と定義されたコーディングスタイルを紐づけておき，その後提出したソースコードのコーディングスタイルが紐づけられたコーディングスタイルと異なると判定した際に盗用を疑う方法を提案している．しかしコーディングスタイルから盗用検出を行う方法では，解答者ごとに複数のソースコードが必要となり，同じ解答者の複数のソースコードが存在しない場合は適用できない．本研究では，盗用検出対象のソースファイル群のソースコードのみを使用して，偶然一致の可能性を考慮した盗用検出方法を提案する．

[*1] http://jplag.ipd.kit.edu/
[*2] https://theory.stanford.edu/ aiken/moss/

3. 提案手法

本研究で提案する手法は，入力として同一のプログラミング言語で書かれた解答ソースファイル群を受け取り，類似度が高く，かつ，偶然一致とは考えられないような少数の解答にのみ共通する内容を持つ解答の組を盗用の可能性として検出する．本研究では，便宜上，1 つの解答を 1 つのファイルとみなして扱うが，これはプログラミング試験の多くの提出者が 1 ファイルのみを用いるためである．複数のファイルを提出している場合は，それらを結合して 1 ファイルとして扱うことで，本手法の適用が可能である．

提案手法は以下の 3 つのステップからなる．

(Step 1) 全体的に類似しているソースコードの組の検出．
ソースコードを N-gram の集合と捉えて，2 つのソースコードの類似度を Jaccard 係数を用いて計測する．

(Step 2) 固有のコード片の一致が多いソースコードの組の検出．
出現頻度の少なさに応じた N-gram に重みを付けて，2 つのソースコードの類似度をコサイン類似度を用いて計測する．

(Step 3) 盗用が疑わしい組の順位付け．
Step 1，2 で計測された類似度結果を用いて，盗用が疑わしい組の順位付けを行う．

図 1 にデータフローを示すように，Step 1 と 2 は互いに独立した基準での類似度の測定である．提案手法の出力は，疑わしい順に並べられた解答ソースファイル名の組の列であり，利用者となる企業の採用試験担当者は，このリストを上位から順に，確認に使うことのできる時間の範囲内で，これらのソースコードの内容を目視で確認する．

以降，各ステップの詳細を述べる．

3.1 全体的に類似しているソースコードの組の検出

Step 1 では，入力されたファイル集合に対して総当たりで類似度を計算し，それを類似度の降順に並べたものを出力する．具体的なファイル単位の類似度として，Ishio らの手法 [16] を用いる．この手法は，オープンソースソフトウェアのファイル単位での再利用を追跡するために提案さ

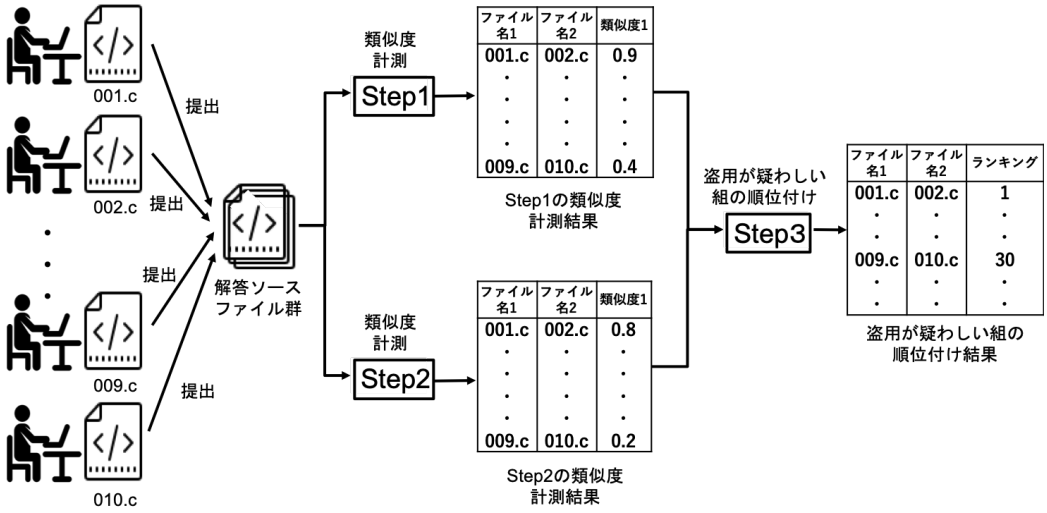

図 1 提案手法の手順
Fig. 1 Overall of the proposed method

れたもので，各ソースファイルをコメントや空白，識別子名の違いを無視して（識別子はすべて単一の文字列 “$p” に変換する形で）トークン列とみなし，各トークンを要素とする 3-gram の多重集合へと変換したうえで，Jaccard 係数を使用して類似度を求める．ソースコード a, b の Jaccard 係数 Jsim(a,b) は，以下の式で求められる．

$$Jsim(a,b) = \frac{|3-gram(a) \cap 3-gram(b)|}{|3-gram(a) \cup 3-gram(b)|}$$

この手法を選定した理由は，解答者による偽装に強いと考えたためである．ソースコードを盗用した解答者は偽装を行うことが知られており，それらをまとめた結果が Novak ら [10] によって報告されている．以下がその中の代表的な 4 つの偽装方法である．

(1) インデント，スペース，改行などの空白の変更
(2) コメントの変更，追加，または削除
(3) 識別子の名前変更
(4) コードの並び替え

Ishio らの手法 [16] は，プロジェクト間でのソースコードの再利用を追跡するため，再利用先プロジェクトにおけるコメントの追記やレイアウトの変更，コーディング規約に基づく識別子名の修正を想定している．そのため，偽装方法 (1), (2), (3) の影響を受けない類似度の定義となっている．そして，3-gram の多重集合表現は，トークンの前後関係をある程度無視するため，偽装方法 (4) のコードの並び替えの影響を受けづらい．

この類似度計算の実装は，オープンソースで公開されているものを使用した[*3]．

3.2 固有のコード片の一致が多いソースコードの組の検出

Step2 では，少数の解答に固有のコード片の一致が多いソースコードの組を検出するような類似度を計算する．Step 1 の類似度は，偽装の影響を受けづらいと考えられるが，偶然似たコード量やアルゴリズムで解いたソースコードの組の類似度の値が高くなる可能性がある．そこで，解答ソースファイル群で出現頻度の低い N-gram に大きく重み付けを行った N-gram の出現頻度によるコサイン類似度を用いる．

ある N-gram t の重み $W(t)$ を以下のように定義する．

$$W(t) = log\frac{\text{全解答ソースコード数} + 1}{t\text{ を含むソースコード数} + 1} + 1$$

この定義はいわゆる IDF (Inverse Document Frequency) に相当し，$W(t)$ の値をそのままベクトルの要素とする．

$W(t)$ の値は出現確率の低い N-gram ほど高くなることから，出現確率が低い N-gram を共有したプログラムほど，その一致が偶然ではなく盗用の可能性であると考え，類似度が高くなる．Shirakawa ら [17] による N-gram IDF の定義は，常に一定の順序で出現する単語列の重みを大きくし，偶然発生した可能性の高い単語の並びの重みを小さくするため，盗用検出の特徴量としては合わないと考え，本研究では採用しなかった．

ソースファイル a, b の類似度は，ソースコード a から求めたベクトルを V_a，ソースコード b から求めたベクトルを V_b としたとき，コサイン類似度 $Csim(V_a, V_b)$ として以下のように定義する．

$$Csim(V_a, V_b) = \frac{V_a \cdot V_b}{|V_a||V_b|}$$

類似度が高いほど盗用の可能性が高いと考え，この類似度の値で降順にファイルの組を並べたものを，Step 2 の出力とする．

このステップも類似度を計算する際の識別子の正規化の有無と，N-gram の長さ (N) という 2 つのパラメータを持

[*3] https://github.com/NAIST-SE/CodeHash

表 1 SOCO データセット
Table 1 SOCO dataset

	C	Java
ソースコードファイル数	79 個	259 個
再利用事例	26 組	97 組

つが，Step 1 とは独立に設定可能であるため，パラメータの性能への影響は 4 章で評価する．

3.3 盗用が疑わしい組の順位付け

Step 3 では，Step 1 と Step 2 で求めた類似度によるランキング結果を使用して，盗用が疑わしい組の順位付けを行う．ランキングの統合方法として，本研究では 2 つのランキングでの順位の値の加算を用いた．ソースファイルの組 p が，Step 1 の出力で第 $r_1(p)$ 位，Step 2 の出力で第 $r_2(p)$ 位にいたとすると，その組 p は $r_1(p) + r_2(p)$ のランクを得たものとして，ランクの昇順に出力を並べる．

4. 評価実験

提案手法の盗用検出の性能を確認するために，公開されているデータセットを対象に，Step 1, 2 の個別での性能と，提案手法としてそれらを組み合わせた結果の性能を評価し，既存ツール JPlag との比較を行う．提案手法の Step 2 については，識別子の正規化がある場合とない場合で，N = 3 から 10 まで変化させた際の性能を評価し，Step 3 も同様にそれらを Step 1 と組み合わせた場合の性能を評価する．

4.1 データセット

本研究では，2014 年に行われた再利用して作成されたソースコードを検出するコンテストである「Detection of SOurce COde Re-use」[*4][9] のトレーニングデータセットを使用した．データセットは大学の授業で出された課題を使用しており，各ソースコードには区別のための ID が振られており，3 人の専門家が目視によって検出した同一言語間での再利用事例が記録されている．C 言語に関してはこの公開データセットをそのまま，Java に関してはコードクローン検索の研究 [18] によって更新された版[*5]を使用する．表 1 に言語ごとのファイル数と再利用事例数を示す．このコンテストは 3 つのチーム [19], [20], [21] が参加し，ソースコードの再利用事例の検出を F-Measure の値で競うコンテストであったが，これらのコンテストで使用されたツール自体は公開されていない．

4.2 評価指標

本研究では，以下の 2 つの指標を用いる．

- F-measure: Precision と Recall の調和平均の値である．本実験において盗用であるかの閾値は，Step 1, 2 ではそれぞれ解答ソースファイル群の全組み合わせの類似度の値の上位 1 %を抽出するものとした．また，Step 3 ではランキングの上位 1 %を抽出した．

- Average Precision@k (AP@k): AP@k は，上位 k 個の正解データの出現時点での Precision の平均をとった値である．正解がランキング上位に含まれるほど，この値が高くなるため，ランキング上位から検査していく利用者にとっての品質に対応する．k の値としては，利用者が現実的にランキングの確認を行える範囲として，k = 20, 50, 100 の 3 段階で値を計算する．

表 2 Step 1 の F-measure
Table 2 Step 1 F-measure

C 言語			Java		
F-measure	Precision	Recall	F-measure	Precision	Recall
0.607	0.567	0.654	0.329	0.213	0.732

表 3 Step 1 の AP@K
Table 3 Step 1 AP@K

C 言語			Java		
AP@20	AP@50	AP@100	AP@20	AP@50	AP@100
0.909	0.810	0.763	1.000	0.992	0.961

表 4 Step 2 の F-measure
Table 4 Step 2 F-measure

識別子を正規化	N-gram	C 言語			Java		
		F-measure	Precision	Recall	F-measure	Precision	Recall
あり	3-gram	0.500	0.467	0.538	0.445	0.287	0.990
	4-gram	0.536	0.500	0.577	0.450	0.290	1.000
	5-gram	0.571	0.533	0.615	0.450	0.290	1.000
	6-gram	0.571	0.533	0.615	0.450	0.290	1.000
	7-gram	0.571	0.533	0.615	0.450	0.290	1.000
	8-gram	0.607	0.567	0.654	0.450	0.290	1.000
	9-gram	0.607	0.567	0.654	0.450	0.290	1.000
	10-gram	0.571	0.533	0.615	0.450	0.290	1.000
なし	3-gram	0.536	0.500	0.577	0.450	0.290	1.000
	4-gram	0.536	0.500	0.577	0.450	0.290	1.000
	5-gram	0.571	0.533	0.615	0.450	0.290	1.000
	6-gram	0.571	0.533	0.615	0.450	0.290	1.000
	7-gram	0.536	0.500	0.577	0.450	0.290	1.000
	8-gram	0.536	0.500	0.577	0.445	0.287	0.990
	9-gram	0.500	0.467	0.538	0.445	0.287	0.990
	10-gram	0.500	0.467	0.538	0.445	0.287	0.990

4.3 提案手法の評価

提案手法のステップ別での性能の評価結果を示す．表 2 に Step 1 の F-Measure の結果を，表 3 に Step 1 の AP@K の結果を示す．Precision が低めとなっているのは，解答アルゴリズムが類似しているソースコードが類似と判定されている可能性が考えられる．

表 4 に Step 2 の F-Measure の結果を，表 5 に AP@k の結果を示す．Step 1 と比べると，C 言語に関しては性能は低く，Java に関しては Recall が非常に高い結果となった．

[*4] https://pan.webis.de/fire14/pan14-web/soco.html
[*5] https://github.com/UCL-CREST/Siamese

表 5 Step2 の AP@K
Table 5 Step2 AP@K

識別子を正規化	N-gram	C 言語			Java		
		AP@20	AP@50	AP@100	AP@20	AP@50	AP@100
あり	3-gram	0.906	0.715	0.680	1.000	0.982	0.948
	4-gram	0.885	0.731	0.710	1.000	0.993	0.966
	5-gram	0.890	0.775	0.728	1.000	0.994	0.982
	6-gram	0.913	0.804	0.776	1.000	0.994	0.979
	7-gram	0.907	0.811	0.784	1.000	0.994	0.975
	8-gram	0.907	0.831	0.753	1.000	0.994	0.966
	9-gram	0.911	0.837	0.736	1.000	0.994	0.962
	10-gram	0.921	0.797	0.726	1.000	0.994	0.959
なし	3-gram	0.921	0.804	0.752	1.000	0.989	0.953
	4-gram	0.928	0.819	0.764	1.000	0.987	0.956
	5-gram	0.929	0.816	0.739	1.000	0.985	0.955
	6-gram	0.957	0.810	0.705	1.000	0.983	0.953
	7-gram	0.942	0.815	0.735	1.000	0.981	0.953
	8-gram	0.935	0.837	0.721	1.000	0.982	0.954
	9-gram	0.943	0.825	0.732	1.000	0.984	0.957
	10-gram	0.943	0.889	0.739	1.000	0.983	0.958

C 言語では，解答ソースコード数が Java と比較して少ないことから，N-gram の重み付けの効果が小さかった可能性がある．全体としてはパラメータによる大きな性能の差は発生しなかったが，N が大きい場合は，識別子の正規化によって Recall が向上する傾向が見られた．また，AP@k については，N が小さい場合，識別子の正規化がないほうが高い値となる傾向が見られた．これは，N が小さいうちは同一識別子の存在が再利用検出の手がかりとなるが，N が大きくなると構文自体の類似が検出に有効に働いた考えられる．

表 6 に Step 3 の F-Measure の結果を，表 7 に AP@k の結果を示す．C 言語のデータにおいては F-Measure，AP@k のどちらの値も Step 1 と Step 2 では同程度であったが，それらを合わせた Step 3 においては Step 1 と Step 2 のどちらよりも高い値が確認できた．これは偽装の影響を受けにくいと考えられる Step 1 の類似度と，誤検出を減らすための Step 2 の類似度の高い組がある程度異なっており，互いの短所を補っている可能性を示した．Java の解答での評価指標の値は Step 2 と比較すると低くなっているが，従来のソースコード類似度を用いた Step 1 よりも効果的に働いている．パラメータの影響を見ると，識別子の正規化あり，N=3 の場合が最も F-measure が高かった．AP@k の値が最も高くなるのは識別子の正規化なし，N=5 であった．

4.4 既存盗用検出ツールとの比較結果

表 8 に，提案手法と JPlag の F-measure の比較結果を示す．提案手法のパラメータには，F-measure が最も高いもの，つまり識別子の正規化あり，N=3 を選択した．JPlag もまた類似度の閾値をパラメータとして要求するため，提案手法と条件を合わせるために，全組み合わせの上位 1 % だけを抽出した．JPlag の性能は，C 言語を対象にした場合は低く，提案手法は性能面で大幅に上回っている．一方，

表 6 Step 3 の F-measure
Table 6 Step 3 F-measure

識別子を正規化	N-gram	C 言語			Java		
		F-measure	Precision	Recall	F-measure	Precision	Recall
あり	3-gram	0.679	0.633	0.731	0.404	0.260	0.897
	4-gram	0.679	0.633	0.731	0.399	0.257	0.887
	5-gram	0.679	0.633	0.731	0.399	0.257	0.887
	6-gram	0.643	0.600	0.692	0.399	0.257	0.887
	7-gram	0.643	0.600	0.692	0.399	0.257	0.887
	8-gram	0.643	0.600	0.692	0.399	0.257	0.887
	9-gram	0.643	0.600	0.692	0.404	0.260	0.897
	10-gram	0.643	0.600	0.692	0.408	0.263	0.907
なし	3-gram	0.571	0.533	0.615	0.413	0.266	0.918
	4-gram	0.571	0.533	0.615	0.413	0.266	0.918
	5-gram	0.607	0.567	0.654	0.413	0.266	0.918
	6-gram	0.607	0.567	0.654	0.413	0.266	0.918
	7-gram	0.607	0.567	0.654	0.413	0.266	0.918
	8-gram	0.607	0.567	0.654	0.413	0.266	0.918
	9-gram	0.571	0.533	0.615	0.413	0.266	0.918
	10-gram	0.536	0.500	0.577	0.413	0.266	0.918

表 7 Step 3 の AP@K
Table 7 Step 3 AP@K

識別子を正規化	N-gram	C 言語			Java		
		AP@20	AP@50	AP@100	AP@20	AP@50	AP@100
あり	3-gram	0.883	0.818	0.748	1.000	1.000	0.976
	4-gram	0.891	0.829	0.781	1.000	1.000	0.978
	5-gram	0.889	0.825	0.752	1.000	1.000	0.973
	6-gram	0.894	0.842	0.743	1.000	1.000	0.980
	7-gram	0.906	0.847	0.727	1.000	1.000	0.975
	8-gram	0.894	0.807	0.740	1.000	1.000	0.974
	9-gram	0.895	0.825	0.761	1.000	1.000	0.974
	10-gram	0.889	0.812	0.771	1.000	1.000	0.977
なし	3-gram	0.931	0.814	0.740	1.000	1.000	0.972
	4-gram	0.930	0.807	0.744	1.000	1.000	0.969
	5-gram	0.945	0.794	0.762	1.000	1.000	0.969
	6-gram	0.953	0.823	0.755	1.000	1.000	0.968
	7-gram	0.927	0.835	0.746	1.000	1.000	0.967
	8-gram	0.913	0.824	0.734	1.000	1.000	0.966
	9-gram	0.902	0.819	0.722	1.000	1.000	0.967
	10-gram	0.901	0.792	0.687	1.000	1.000	0.966

表 8 提案手法 (識別子正規化あり，N=3) と JPlag の F-measure の比較
Table 8 Proposed method (with normalization, N=3) vs. JPlag's F-measure

	C 言語			Java		
	F-measure	Precision	Recall	F-measure	Precision	Recall
Step 1	0.607	0.567	0.654	0.329	0.213	0.732
Step 2	0.500	0.467	0.538	0.445	0.287	0.990
Step 3	0.679	0.633	0.731	0.404	0.260	0.897
JPlag	0.321	0.300	0.346	0.445	0.287	0.990

Java では，JPlag の性能は提案手法の Step 2 と同程度であった．提案手法は Step 3 のランキングの統合によって見逃しが発生し，JPlag の性能を下回っていることから，ランキングの統合方法の改善が今後の課題である．

表 9 に，提案手法と JPlag の AP@k の比較結果を示す．提案手法は JPlag の結果を上回っており，正しいコードの組を上位に提示していることが確認できた．

4.5 妥当性への脅威

この実験で使用したデータセットはコンテストのために作られたものであり，現実に発生するカンニング件数に比

表 9 提案手法 (識別子正規化あり，N=3) と JPlag の AP@k の比較

Table 9 Proposed method (with normalization, N=3) VS JPlag's AP@K

	C 言語			Java		
	AP@20	AP@50	AP@100	AP@20	AP@50	AP@100
Step 1	0.909	0.810	0.763	1.000	0.992	0.961
Step 2	0.883	0.818	0.748	1.000	1.000	0.978
Step 3	0.953	0.847	0.781	1.000	1.000	0.980
JPlag	0.866	0.758	0.758	1.000	0.996	0.954

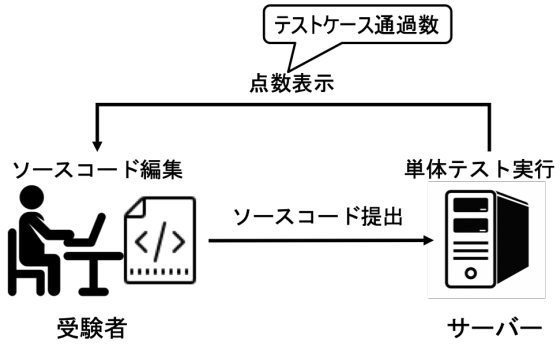

図 2 track の試験環境

Fig. 2 Track test environment

べると，再利用事例を数多く含んでいる可能性がある．盗用の可能性ありとして検出する件数が Precision に大きく影響するため，類似度の閾値の決め方によって F-measure の値は大きく変動する．この実験では，盗用の判定基準を全組み合わせの上位 1 %とする相対的な抽出であるが，特定の値よりも高いものを取り出す，あるいは統計的な外れ値だけを検査するなどの閾値の決定方法もありうる．

5. 企業の採用試験データへの適用

提案手法を企業の採用試験データへ適用し，実際のカンニング検出を試みた結果を述べる．採用試験データは株式会社ギブリー[*6]が提供しているオンラインのプログラミング試験サービス「track（トラック）」[*7]を使用した 2 つの企業の採用試験データである．このサービスはオンラインで試験を行うことを前提に使われており，特定の試験会場ではなく，自宅など受験者本人が受験したい場所から受験者本人が所有している PC の Web ブラウザを使用して，受験を行うことができる環境となっている．図 2 に試験環境の模式図を示す．各課題ごとに制限時間が設けられており，受験者は雛形のコードを受け取り，そのソースコードを編集して解答を作成する．受験者は制限時間内であれば何度でも解答ソースコードを提出することが可能である．各提出ごとに，テストケース通過数が試験の点数として受験者のモニタに表示されるため，受験者は試験途中で点数を確認しながら解答を進めていくことが可能である．

[*6] https://givery.co.jp
[*7] https://tracks.run

表 10 データセット 1 への提案手法の適用結果と企業の判断との比較

Table 10 Results of the proposed method on Dataset 1

判断基準	企業の判断	提案手法での検出
行動データ (1)	23	8
行動データ (2)	1	1
行動データ (3)	10	6
行動データ (4)	16	6
類似度	-	139
合計	50	160

1 つ目のデータセットは，インド人の学生を対象にインドにて行われた採用試験データになっている．8 日間程度の試験を受ける期間が設けられており，受験者はその期間内に受験することが要求されているが，時間や場所に関しての制約は存在しておらず，受験者本人が決めている．課題は盗用行為の防止のために，出題者側が同程度の難易度とした 8 問のうちランダムで選択された 1 問を受験者が解答している．解答するプログラミング言語についての制約はなく，受験者本人が決めている．最終的に収集した解答データは，受験者は 903 名の内，試験期間内に試験課題を一度も開かなかった受験者を除く 588 名の受験データである．

2 つ目のデータセットは，日本で行われた，主に日本人の学生を対象にした採用試験データである．こちらも試験を受ける期間が設けられており，受験者はその期間内に受験することが要求されているが，時間や場所に関しての制約は存在しておらず，受験者本人が決めている．プログラミング言語も受験者が選択可能であるため，本実験では，使用者が多かった Python と C++ の 2 言語を対象とし，152 名の受験データを分析した．

5.1 データセット 1 への提案手法の適用結果

このデータセットでは，企業の担当者がオンライン採用試験環境上で取得できる行動履歴データ，つまり解答時間と提出までのテスト実行回数を基に，以下の基準で不正行為が疑われている人が合計 50 人存在していた．

(1) 課題を開いてから提出まで、1 度しかテスト実行をしていない．つまり，外部エディタなどでプログラミングしたものを貼り付けるなどして，いきなり完成版を提出している可能性が高い．（該当者：23 人）
(2) (1) に加えて，プログラミング知識を問う課題でも疑いリストに入っている．（該当者：1 人）
(3) (1) に加えて，問題を開いてから満点を取得するまでの時間が 2 分未満である．（該当者：10 人）
(4) (1), (2), (3) には該当しないが，他の受験者に比べて明らかに回答時間が短い．（該当者：16 人）

企業の担当者は，(3) が最も強く，(2)，(1)，(4) の順で不正を疑っていた．

データセット 1 に提案手法を適用し，提案手法に基づいて盗用を疑った人数と，企業の担当者が行動履歴データから不正を疑った人数の結果を表 10 に示す．提案手法では 2 人 1 組のペアで検出されるが，企業の担当者の判断ではなさそうな個人で分類されていたため，提案手法も類似ファイルの一方に含まれている人数を数え上げた．提案手法で盗用を疑った人数は 160 人であり，企業の担当者の判断と合致した人は合計 21 人 (42%) であった．不正を疑われた理由別での人数の内訳は (1) が 8 人，(2) が 1 人，(3) が 6 人，(4) が 6 人となっており，行動データに基づく基準とは独立して，異なる視点から盗用疑いを検出していると考えられる．

企業の判断と提案手法の出力の差異の 1 つ目は，提案手法の類似度で高い値が算出され，盗用が疑われたが，企業の担当者が行動履歴データから不正を疑われなかったパターンである．139 名がこれに該当した．これは，実際に解答を開始してから，盗用した際の偽装に時間をかけたために，解答時間の短さや提出回数の少なさでといった行動履歴データでは盗用として企業の担当者は検出できなかったと考えられる．実際に，提案手法の Step 1 で類似度が 1.0 となるような，コメントや識別子名以外が完全一致している解答ファイルの組が，企業の担当者が疑った 50 人に入っていなかったことを確認した．

差異の 2 つ目は，企業の担当者が行動履歴データから不正を疑われたが，提案手法の類似度で高い値が出なかったものである．29 件がこれに該当する．この差異が発生した可能性として考えられるのは 3 点ある．1 点目はインターネットなどで出題された課題と類似した問題を発見し，コピーして提出した可能性である．この場合，他の解答者との類似度を計測している提案手法では，解答の類似を検出できない．2 点目は，提案手法の類似度では高い値が算出されない偽装方法が使用された可能性で，3 点目は，不正行為はしておらず本当に短い時間で完成させた可能性である．データセット 1 の課題は基礎的な問題で，少ないコード量 (16 行) でも解答は可能な問題であったことから，実際に短い時間で解答した可能性が考えられる．この結果から，ソースコードの類似度に基づく提案手法は有効ではあるが，受験者の行動履歴データと組み合わせた総合的な判断が，現実的なカンニング検出では重要になると考えられる．

5.2 データセット 2 への提案手法の適用結果

データセット 2 に対して提案手法を適用し，盗用の疑いのあるソースコードとして 26 組を検出した．しかし，データセット 1 と 2 の Step 1 における類似度の箱ひげ図を図 3 に示すが，データセット 1 に関しては Python も C++ での回答も類似度が 1.0 に近い値が記録しているものがあるが，データセット 2 に関しては最大の組でも 0.6 程度と

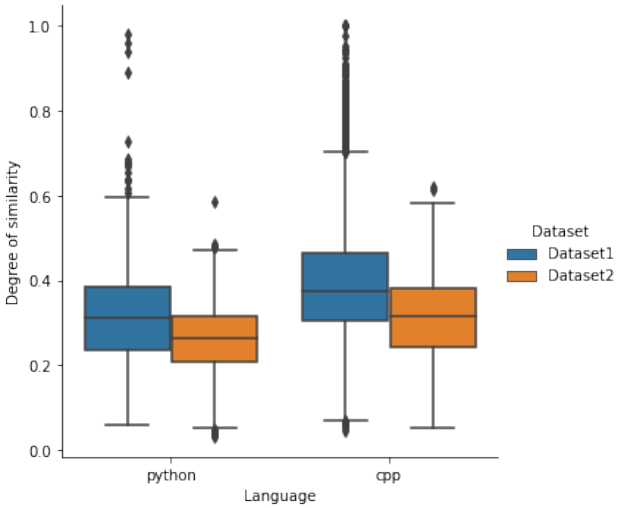

図 3 各データセットの Step 1 の類似度の分布
Fig. 3 Distributions of similarity of Step 1 for each dataset

なっており，類似度が 1.0 に近いような盗用の疑いが特に強い組は検出されなかった．

実際に，この結果を企業の採用担当者が目視で確認した結果，部分的な類似は見受けられるが，26 組すべて，盗用により作成したソースコードとはみなさないとの回答を得た．ソースコードの盗用を行っている者がいる可能性は低いという類似度から得られた予想は裏付けられたが，類似度の値がどのぐらいであれば盗用を疑うべきかという適切な閾値の決定は，今後の課題である．

6. まとめ

本論文では解答固有のソースコード片を考慮したプログラミング試験解答の盗用検出手法を提案した．提案手法は類似しているソースコードを 2 つの類似度指標を組み合わせて盗用を検出する仕組みとなっており，既存研究で定義された N-gram の Jaccard 係数に基づく類似度に，少数の解答にのみ出現する N-gram に重みを付けたコサイン類似度を組み合わせた．提案手法の評価を公開データセット SOCO を対象に行なった結果，公開データセットにおいて，提案手法は既存ツール JPlag と比較して F-measure, Averege Presicion のどちらの評価指標においても高い性能を達成した．この結果から，ソースコードの少数の解答にのみ出現するソースコード片を重視するというアイディアの有効性を確認した．また，企業の採用試験で取得したデータセットに対する適用結果から，提案手法は有効である一方で，企業の担当者が盗用行為を疑った人と提案手法で盗用行為を疑った人との間には差異があることを確認した．受験者の行動履歴データと組み合わせた総合的な判断が，実際の適用では重要になると考えられる．

今後の課題としては，まず，適切な類似度の閾値の決定が挙げられる．本研究で実施した実験ではランキングの上位 1 %という相対的な値を用いたが，利用者は，実際に盗

用の可能性が低くとも常に一定の件数を検査するというコストが発生する．また，受験者の行動データを用いたさらなる分析が必要である．企業の担当者が盗用行為を疑った人で提案手法で盗用を疑った人には差異が発生したが，特に企業の担当者が盗用行為を疑った人で，提案手法では盗用行為を疑われなかった人に関しては，インターネットに公開されたソースコードの流用など，他の可能性を考慮した性能の評価が必要となる．

謝辞

本研究は JSPS KAKENHI No.20H05706 の助成を受けたものです．

参考文献

[1] Marais, E., Minnaar, U. and Argles, D.: Plagiarism in e-learning systems: Identifying and solving the problem for practical assignments, *Proceedings - Sixth International Conference on Advanced Learning Technologies, ICALT 2006*, Vol. 2006, pp. 822–824 (online), DOI: 10.1109/icalt.2006.1652567 (2006).

[2] Prechelt, L., Malpohl, G. and Philippsen, M.: Finding Plagiarisms among a Set of Programs with JPlag, Vol. 8, No. 11, pp. 1016–1038 (2002).

[3] Joy, M. and Luck, M.: Plagiarism in Programming Assignments, *IEEE Transactions on Education*, Vol. 42, pp. 129 – 133 (online), DOI: 10.1109/13.762946 (1999).

[4] Ahtiainen, A., Surakka, S. and Rahikainen, M.: Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises, *ACM International Conference Proceeding Series*, Vol. 276, pp. 141–142 (online), DOI: 10.1145/1315803.1315831 (2006).

[5] Chuda, D. and Navrat, P.: Support for checking plagiarism in e-learning, *Procedia - Social and Behavioral Sciences*, Vol. 2, pp. 3140–3144 (online), DOI: 10.1016/j.sbspro.2010.03.478 (2010).

[6] Grier, S.: A tool that detects plagiarism in Pascal programs, *ACM SIGCSE Bulletin*, Vol. 13, pp. 15–20 (online), DOI: 10.1145/953049.800954 (1981).

[7] Arwin, C. and Tahaghoghi, S.: Plagiarism detection across programming languages, Vol. 48, pp. 277–286 (online), DOI: 10.1145/1151699.1151730 (2006).

[8] Cosma, G. and Joy, M.: Source-code plagiarism: A UK academic perspective (2006).

[9] Re-us, S. C.: UAM @ SOCO 2014 : Detection of Source Code Re-use by mean of Combining Different Types of Representacions, pp. 2–6 (2014).

[10] Novak, M., Joy, M. and Kermek, D.: Source-code similarity detection and detection tools used in academia: A systematic review, *ACM Transactions on Computing Education*, Vol. 19, No. 3 (online), DOI: 10.1145/3313290 (2019).

[11] Bowyer, K. W. and Hall, L. O.: Experience using 'MOSS' to detect cheating on programming assignments, *29th ASEE/IEEE Frontiers of Education Conference*, pp. 18–22 (1999).

[12] Domin, C., Pohl, H. and Krause, M.: Improving plagiarism detection in coding assignments by dynamic removal of common ground, *Conference on Human Factors in Computing Systems - Proceedings*, Vol. 07-12-May-, pp. 1173–1179 (online), DOI: 10.1145/2851581.2892512 (2016).

[13] Mariani, L. and Micucci, D.: AuDeNTES: Automatic detection of teNtative plagiarism according to a rEference solution, *ACM Transactions on Computing Education*, Vol. 12, No. 1 (online), DOI: 10.1145/2133797.2133799 (2012).

[14] Ohno, A. and Murao, H.: A Quantification of Students Coding Style Utilizing HMMBased Coding Models for In-Class Source Code Plagiarism Detection, pp. 553 – 553 (online), DOI: 10.1109/ICICIC.2008.614 (2008).

[15] Ohno, A. and Murao, H.: A two-step in-class source code plagiarism detection method utilizing improved CM algorithm and SIM, *International Journal of Innovative Computing, Information and Control*, Vol. 7 (2011).

[16] Ishio, T., Sakaguchi, Y., Ito, K. and Inoue, K.: Source File Set Search for Clone-And-Own Reuse Analysis, *IEEE International Working Conference on Mining Software Repositories*, pp. 257–268 (online), DOI: 10.1109/MSR.2017.19 (2017).

[17] Shirakawa, M., Hara, T. and Nishio, S.: IDF for Word N-grams, *ACM Transactions on Information Systems*, Vol. 36, No. 1, pp. 5:1–5:38 (online), DOI: 10.1145/3052775 (2017).

[18] Ragkhitwetsagul, C. and Krinke, J.: Siamese: scalable and incremental code clone search via multiple code representations, *Empirical Software Engineering*, Vol. 24, No. 4, pp. 2236–2284 (online), DOI: 10.1007/s10664-019-09697-7 (2019).

[19] Ganguly, D. and Jones, G. J.: DCU@FIRE-2014: An information retrieval approach for source code plagiarism detection, *ACM International Conference Proceeding Series*, Vol. 05-07-Dec-, pp. 39–42 (online), DOI: 10.1145/2824864.2824887 (2014).

[20] Re-us, S. C.: UAM @ SOCO 2014 : Detection of Source Code Re-use by means of Combining Different Types of Representacions, pp. 2–6 (2014).

[21] García-Hernández, R. A. and Ledeneva, Y.: Identification of Similar Source Codes based on Longest Common Substrings, *Fire2014* (2014).