Um Método de Detecção de Plágio para Sistemas Juiz On-line baseado no Comportamento dos Alunos

David B. F. Oliveira [1], Ronem M. Lavareda Filho[1], Elaine H. T. Oliveira[1], Leandro S. G. Carvalho[1], Filipe Dwan Pereira[2], Juan G. Colonna[1], Adria Menezes[1]

[1]Instituto de Computação – Universidade Federal Amazonas (UFAM)

[2]Departamento de Ciência da Computação – Universidade Federal de Roraima (UFRR)

{david,ronem,elaine,galvao,juancolonna,adria}@icomp.ufam.edu.br

filipe.dwan@ufrr.br

Abstract. *Plagiarism is a serious and growing problem in the academic environment, which interferes directly in the quality of teaching. This research is contextualized in the problem the detection of plagiarism in CS1 courses. In these courses, the codes developed by students tend to be simple and small, making it difficult for traditional methods based on code similarity to detect plagiarism. To overcome this difficulty, this work proposes a plagiarism detection method that uses evidences extracted from the logs of online judge systems and which are based on the student's behavior during their attempts to solve the programming exercises. As far as researched, this is the first method in the scientific literature that is based on student behavior, having reached 0.83 in the F-measure during the plagiarism detection process.*

Resumo. *A prática do plágio é um problema grave e crescente no meio acadêmico, que interfere diretamente na qualidade do ensino. Esta pesquisa se contextualiza no problema de detecção de plágio entre códigos-fonte desenvolvidos por alunos de disciplinas de introdução de programação, onde os códigos tendem a ser simples e pequenos, dificultando o processo de detecção de plágio. Para contornar essa dificuldade, neste trabalho é proposto um método de detecção de plágio baseado em técnicas de aprendizagem de máquina que usa evidências extraídas de registros de logs de sistemas juízes online e que são baseadas no comportamento do aluno durante as atividades de programação. Até onde foi pesquisado, esse é o primeiro método da literatura científica que é baseado no comportamento do aluno, tendo alcançado 0.83 na medida-F durante o processo de detecção de plágio.*

1. Introdução

O avanço tecnológico das últimas décadas proporcionou uma série de benfeitorias a várias áreas de nossa sociedade. No contexto educacional, por exemplo, o resultado desse avanço possibilitou o surgimento de uma grande variedade de sistemas computacionais capazes de fomentar a disseminação do conhecimento e auxiliar nos processos de ensino-aprendizagem. Dentre tais sistemas, os juízes on-line voltados para o ensino de programação têm ganhado bastante atenção da literatura científica nos últimos anos [Dwan et al. 2017, Wasik et al. 2018, Watanobe et al. 2020, de Oliveira et al. 2020, Pereira et al. 2021a].

DOI: 10.5753/sbie.2021.218397

Juízes on-line são sistemas de correção automática de códigos-fonte comumente empregados como ferramentas de apoio pedagógico em disciplinas de programação [Galvão et al. 2016, Santos et al. 2020, Pereira et al. 2020b, Costa et al. 2021]. Eles proveem feedbacks instantâneos e automáticos aos estudantes durante suas tentativas de solucionar os exercícios de programação cadastrados por seus professores [Fonseca et al. 2019, Pereira et al. 2020c, Freitas Júnior et al. 2020]. Por conta disso, tais sistemas têm sido cada vez mais adotados pelas universidades, tanto do Brasil como do exterior [Wasik et al. 2018, Lima et al. 2020, Pereira et al. 2021b]. No entanto, apesar das vantagens do uso de juízes online, ainda carecem estudos sobre como detectar e/ou coibir as práticas de plágio que ocorrem nos exercícios compartilhados através desses sistemas [Pereira et al. 2020c, Pereira et al. 2021a]. Porém, no que concerne as disciplinas de introdução à programação (CS1), que são apontadas por muitos trabalhos como disciplinas de alta complexidade para alunos iniciantes [Rossi 2016, Pereira et al. 2020a, Araujo et al. 2021], a prática de plágio pode parecer um artifício válido para alcançar o êxito tão esperado [Lima et al. 2021].

Existem inúmeros métodos de detecção de plágio de códigos-fonte disponíveis na literatura, que podem ser baseados tanto em evidências extraídas dos códigos-fontes como do processo de compilação e/ou execução de tais códigos. Para exemplificar, entre os mais conhecidos estão os métodos baseados em *tokens*, que transformam os códigos-fontes em sequências de *tokens* e usam técnicas de cálculo de similaridade para comparar as sequências geradas nos códigos dos diferentes alunos.

Entretanto, os códigos desenvolvidos por alunos de turmas de CS1 tendem a ser simples e pequenos, de forma que grande parte dos métodos de detecção de plágio propostos na literatura podem não conseguir extrair uma quantidade suficientemente grande de evidências de tais códigos para viabilizar a detecção de plágio. Além disso, os códigos de alunos dessas turmas tendem a ser muito parecidos, pois os estudantes aprendem a programar com a ajuda dos mesmos professores e instrutores. Como os alunos ainda não tiveram tempo de desenvolver seu próprio estilo de programação, eles acabam copiando os estilos de seus professores, o que aumenta consideravelmente a similaridade entre os códigos dos alunos [Ril Gil et al. 2014].

Neste artigo, propomos um método que não é baseado em evidências extraídas dos códigos-fonte, e sim do comportamento dos alunos durante o processo de desenvolvimento dos códigos. Desta forma, este trabalho não considerará os códigos dos alunos, e sim os *logs* que eles geram durante o desenvolvimento dos códigos em sistemas juízes on-line. Este trabalho parte da premissa de que as ações desempenhadas por alunos (registradas em *logs*) que realmente desenvolveram seus códigos diferem das ações desempenhadas por alunos que copiaram seus códigos de outrem.

Neste trabalho, serão usados os *logs* registrados no sistema juiz on-line CodeBench[1], desenvolvido pelo Instituto de Computação da Universidade Federal do Amazonas. A partir dos *logs* desse sistema, serão coletadas uma série de variáveis capazes de representar o comportamento dos alunos durante o desenvolvimento de seus códigos. Exemplos de variáveis são: número de submissões e testes feitos, proporção entre o número de caracteres colados e a quantidade de caracteres digitados, número de acessos

[1] https://codebench.icomp.ufam.edu.br

ao sistema, tempo de resolução, etc. Após essas variáveis serem extraídas dos *logs*, será adotado um conjunto de técnicas de classificação binária supervisionada para classificar os códigos entre "códigos desenvolvidos"e "códigos plagiados". Ate onde foi pesquisado na literaturam científica, este é o primeiro método de detecção de plágio baseado no comportamento do aluno.

Desta forma, este trabalho visa responder a seguinte questão de pesquisa: (QP1) perfis baseados no comportamento dos alunos durante o processo de desenvolvimento de códigos são úteis para detectar ocorrências de plágio praticadas por alunos iniciantes de programação? (H1) A hipótese é que é possível identificar perfis de comportamento típicos de práticas de plágio, a partir dos *logs* registrados durante o processo de desenvolvimento de códigos em sistemas de juízes on-line.

Este artigo está organizado da seguinte forma: na Seção 2 são apresentados os trabalhos relacionados, na Seção 3 é apresentado método de detecção de plágio proposto neste trabalho, na Seção 4 é mostrado o cenário experimental e o dataset adotado, na Seção 5 é apresentada a análise experimental e os resultados, seguido da conclusão, na Seção 6.

2. Trabalhos Relacionados

Existe uma grande variedade de técnicas de detecção de plágio em códigos-fonte disponíveis na literatura, que normalmente são classificadas de acordo com os tipos de evidências usadas para a detecção [Gomes and Matos 2020, Novak et al. 2019]. Entre os tipos de técnicas mais comuns, podemos citar: baseadas em tokens, baseadas em *birthmark*, baseadas em *fingerprints*, e baseadas em árvores. A seguir, será apresentado alguns exemplos de técnicas de tais classes.

Nas técnicas baseadas em *tokens* [Yuan and Guo 2012, Wan et al. 2018], o código-fonte é transformado em uma série de *tokens*, que são sequências de um ou mais caracteres presentes na gramática da linguagem utilizada. A detecção de plágio é feita por meio das comparações entre as sequências de *tokens* obtidas a fim de encontrar a maior subsequência comum. Um exemplo de método baseado nessa estratégia é apresentado em [Wan et al. 2018], que usa uma técnica de estimativa de similaridade chamada *simhash* [Charikar 2002] para detectar plágio em códigos Verilog HDL.

Marcas de nascença, ou *birthmarks*, são características de códigos-fonte que são muito resistentes a tentativas de ofuscação, onde uma série de mudanças são feitas nos códigos para tentar encobrir a prática de plágio [Yuan et al. 2018, Tian et al. 2015]. Os métodos baseados nessas características procuram detectar ocorrências de plágio através da análise de similaridade entre as marcas de nascença presentes em pares de códigos. Um exemplo de método desta classe é apresentado em [Tian et al. 2015], onde os autores apresentam uma técnica chamada *DYnamic Key Instruction Sequence* (DYKIS) para extrair marcas de nascença dos executáveis (*dinamic birthmarks*), e não dos códigos-fonte (*static birthmarks*).

Os métodos baseados em *fingerprints* procuram gerar valores *hashes* para representar determinados trechos dos códigos-fonte. A detecção de plágio é feita mediante a comparação dos valores hashes extraídos de diferentes documentos. O MOSS [Schleimer et al. 2003] (*Measure of Software Similarity*) é um exemplo de

método baseado em *fingerprints*, que adota uma técnica chamada de *Winnowing* [Schleimer et al. 2003] para gerar os valores *hashes* dos códigos-fonte. Conforme mostrado em [Wan et al. 2018], o MOSS é um dos métodos mais utilizados para detecção de plágio de códigos-fonte, e foi escolhido para ser o baseline do método proposto neste trabalho.

Técnicas baseadas em árvores, também chamadas de *Abstract Syntax Trees* (ASTs), criam uma representação hierárquica da estrutura sintática do código-fonte, formada pelas representações abstratas de cada elemento [Wan et al. 2018]. Assim, uma AST é uma árvore de *tokens* que segue um conjunto de regras específicas de determinada linguagem de programação. A detecção de plágio em uma estratégia AST consiste em encontrar as subárvores comuns aos pares de códigos-fonte por meio de algoritmos de casamento (*matching*) em árvores. Em [Tao et al. 2013] é proposto um método baseado em AST para as linguagens C/C++. Para realizar a detecção, esse método transforma o código em uma representação AST por meio das ferramentas Lex[2] e Yacc[3]. Através dos *tokens* obtidos com o Lex, o Yacc forma as regras de produção e a respectiva estrutura hierárquica do código-fonte. Logo após, é calculado um valor *hash* para cada nó da árvore. Por fim, o cálculo de similaridade é feito com base nesses valores. Nós da árvore que possuem valores *hash* iguais e possuem a mesma quantidade de subnós, são detectados como plágio.

Além das citadas, existem outros tipos de técnicas de detecção de plágio em códigos-fonte, como por exemplo técnicas baseadas em grafos, em casamento de strings, baseadas em nGramas, etc. No entanto, não foram encontradas técnicas baseadas em *logs* de sistemas juízes on-line, tal como a técnica que será apresentada neste trabalho.

3. O Método de Detecção de Plágio

A ideia do método apresentado neste trabalho é usar os registros de *logs* de juízes on-line para gerar um conjunto de evidências capazes de representar o comportamento dos alunos enquanto esses desenvolvem cada uma das questões propostas pelos professores. Neste trabalho, o comportamento de um aluno durante a resolução de um exercício será representado pela expressão $\beta(a, e)$, onde a representa um aluno e e representa o exercício resolvido.

Vale notar que o intuito deste trabalho é identificar se um aluno a cometeu algum comportamento de plágio durante a resolução de um exercício e. Desta forma, $\beta(a, e)$ representa um conjunto de evidências extraídas de *logs* de juízes on-line capazes de representar o comportamento do aluno a durante a resolução do exercício e. A seguir, são apresentadas as evidências utilizadas neste trabalho para compor o comportamento β:

1. **Tamanho do *log*** - os sistemas juízes on-line são capazes de registrar todas as ações dos alunos durante a resolução dos exercícios, e espera-se que os alunos que estão realmente desenvolvendo seus códigos gerem uma quantidade maior de registros de *logs* do que os alunos que simplesmente copiam seus códigos de outros alunos [Pereira et al. 2019a];

[2]Ferramenta para geração de analisadores léxicos, capaz de dividir o código-fonte em *tokens* de acordo com a gramática especificada.

[3]Ferramenta para geração de analisadores sintáticos.

2. **Quantidade de caracteres digitados** - este evento indica quantos caracteres foram digitados durante o desenvolvimento dos códigos, mesmo que tais caracteres sejam apagados posteriormente. A intuição é que, quanto maior o número de caracteres digitado, mais o aluno trabalhou em seu código;
3. **Quantidade de caracteres apagados** - durante o desenvolvimento de uma solução computacional é muito comum a necessidade de apagar e editar trechos do código, e por isso uma variável potencialmente útil é a quantidade de caracteres apagados. As deleções podem ser feitas através das teclas *backspace* e *delete*, através da função *replace*, ou através da seleção de trechos de códigos e posterior inserção de novos caracteres para substituir os trechos selecionados;
4. **Percentual de caracteres colados no código-fonte** - o uso de recursos de *copy* e *paste* (`ctrl+c` e `ctrl+v`) é bastante comum durante as atividades de programação, mesmo para os alunos que estão de fato desenvolvendo seus códigos. No entanto, quando uma parte significativa do código do aluno é proveniente de eventos *paste* (`ctrl+v`), isso pode representar um indício de plágio. Desta forma, esta evidência calcula o número de caracteres colados dividido pelo tamanho do código submetido pelo aluno.
5. **Quantidade de testes executados** - em muitos juízes on-line, os alunos podem testar seus códigos com suas próprias entradas antes de submetê-los para verificação de corretude. Um aluno que constrói suas respostas pode fazer diversos testes durante a resolução dos exercícios de programação, para garantir que as etapas do código estejam corretas [Pereira et al. 2019b];
6. **Número de submissões executadas** - uma submissão é feita quando o aluno deseja que o juiz online verifique a corretude do seu código através dos casos de testes cadastrados pelos professores. Várias submissões podem identificar que o aluno está construindo uma resposta, embora também seja comum o aluno submeter uma única vez sem ter plagiado [Pereira et al. 2019b];
7. **Quantidade de erros de sintaxe nos testes e submissões** - durante o processo de codificação, é comum que o aluno cometa alguns erros de sintaxe até finalizar sua atividade [Pereira et al. 2019b];
8. **Número de erro de lógica nas submissões** - ao construir sua própria solução o aluno poderá fazer várias tentativas de submissões, até que consiga desenvolver um código que atenda a todos os casos de testes cadastrados pelos professores;
9. **Quantidade de *logins*** - um aluno que não costuma acessar o juiz online com frequência, ou seja, não mantém um hábito de treino, possivelmente não conseguirá concluir os exercícios sozinho. Nesses casos, alguns alunos podem recorrer à prática de plágio numa tentativa de garantir uma boa nota nos exercícios [Pereira et al. 2019b];
10. **Proporção entre as notas das avaliações e as notas nas listas de exercícios** - essa evidência procura verificar se as notas das avaliações sem consulta são compatíveis com as notas obtidas nas listas de exercícios, tendo em vista que para obter uma boa nota nas avaliações é importante que o aluno tenha praticado antes;
11. **Média nas avaliações** - alunos com boas médias nas avaliações tendem a dominar bem o conteúdo da disciplina de programação, e por isso não precisa recorrer à práticas de plágio nas listas de exercícios [Pereira et al. 2019b];
12. **Navegação no código** - quando o aluno está de fato trabalhando em uma solução para um exercício, ele costuma navegar no código em construção, seja através as

teclas *right*, *left*, *up*, *down*, seja através de cliques do *mouse*;

13. **Proporção de atividades corretas** - contabiliza o percentual de exercícios corretos submetidos pelo aluno, sejam de listas de exercícios ou das avaliações;
14. **Índice de procrastinação** - verifica se a submissão do aluno foi feita perto do encerramento do prazo de entrega da lista de exercício ou avaliação. O objetivo dessa variável é identificar o comportamento de plágio pelo aluno que não conseguiu desenvolver sua própria resposta e por não ter mais tempo disponível decide plagiar. O valor dessa variável varia de 0 a 1, onde 0 irá indicar que o aluno entregou o exercício assim que o trabalho foi liberado pelo professor e 1 irá indicar que aluno entregou seu código nos minutos finais do prazo de entrega [Pereira et al. 2020b];
15. **Dificuldade do exercício** - verifica a proporção de alunos que conseguiram resolver o exercício. A intuição é que os comportamentos de plágio tendem a ser mais comuns para exercícios mais difíceis [Pereira et al. 2019b];
16. **Tempo do exercício** - visa verificar o tempo que o aluno levou para concluir o exercício. Caso o tempo seja muito curto, pode indicar que a resposta não foi construída pelo aluno, observando que este tempo será contabilizado conforme o tempo gasto na área de programação e não em estadia do site [Pereira et al. 2020b];

Uma vez que o comportamento β, representado pelas 16 variáveis descritas acima, tenha sido definido para um conjunto de alunos, será usado um conjunto de técnicas de aprendizagem de máquina supervisionada para classificar os comportamentos β em dois tipos: (i) comportamento de quem está plagiando uma questão; e (ii) comportamento de quem de fato está desenvolvendo seu código.

4. Cenário Experimental e Dataset

Como o método de detecção de plágio proposto neste trabalho é baseado em técnicas de aprendizagem de máquina supervisionada, foi necessário a confecção de um dataset de códigos-fonte rotulados. Para compor esse dataset, optou-se por utilizar os códigos desenvolvidos por alunos da disciplina de Introdução à Programação de Computadores (IPC) da Universidade Federal do Amazonas durante o primeiro semestre de 2019. Nessa Universidade, os exercícios da disciplina de IPC são disponibilizados através do juiz on-line CodeBench e devem ser solucionados através da linguagem Python. A ementa da disciplina de IPC é dividida em 07 tópicos ou módulos, que abrangem os seguintes conteúdos: (i) variáveis e programação sequencial; (ii) estruturas condicionais, (iii) estruturas condicionais aninhadas, (iv) repetição por condição, (v) vetores e strings; (vi) repetição por contagem; e (vii) matrizes. A seguir, é apresentado um breve algoritmo com uma descrição de todos os passos que foram seguidos para a elaboração do dataset:

1. **Pré-processamento**: Durante esta etapa, todos os códigos da base (no caso do nosso dataset, todos os códigos desenvolvidos por alunos de IPC durante o primeiro semestre de 2019) são limpos de forma a restar apenas os símbolos, *tokens* e funções presentes na gramática da linguagem de programação utilizada. Esse processo de limpeza dos códigos foi proposto originalmente por [Oliveira et al. 2016], e para realizá-lo em nosso dataset foram mantidos nos códigos apenas o seguinte conjunto de termos e caracteres da gramática da linguagem Python: (,), :, [,], Tab, ″, ′, $*$, $-$, $+$, %, !, ., /, $-$, #, $=$, $>$, $<$, and, or, $numpy$, os, sys, $random$, $math$, $False$, $None$, $True$, and, $elif$, $else$, for,

$from$, if, $import$, in, is, not, or, $pass$, $while$, abs, all, $bool$, chr, $dict$, $eval$, $float$, $input$, int, len, min, max, ord, pow, $print$, $range$, $round$, str e sum. Para compor esse conjunto, foi feita uma análise do conteúdo programático da disciplina de IPC da Universidade Federal do Amazonas, selecionando todas as funções, *tokens* e símbolos ensinados durante a disciplina. Note que através desse processo serão eliminados todo tipo de conteúdo dos códigos que não faça parte da lista apresentada. Por exemplo, comentários e nomes de variáveis são completamente removidos dos códigos.

2. **Formação de N pares de códigos similares**: através de um mecanismo randômico, é feita uma seleção de N códigos da base pré-processada gerada na etapa anterior. O valor de N representa o tamanho do dataset que se deseja compilar. Após isso, para cada código c dos N sorteados, seleciona-se o código da base pré-processada que é mais similar ao código c e que contém uma solução para o mesmo exercício. A similaridade entre os códigos pode ser calculada, por exemplo, através da distância de *Levenshtein* [Su et al. 2008]. Ao final desse processo, teremos N pares de código, onde cada par contém 2 códigos desenvolvidos para o mesmo exercício e que possuem alta similaridade entre si.
3. **Processo de rotulação**: uma vez selecionado os N pares de códigos que farão parte do dataset, o próximo passo é conduzir uma análise manual de cada um desses pares e rotulá-los entre "plágio"e "não plágio". Para isso, é necessário recuperar o código original (código anterior à etapa de pré-processamento) de cada um dos N pares de códigos selecionados na etapa anterior, e então avaliar manualmente se houve ou não ocorrência de plágio em cada um dos pares de código.

No dataset criado para viabilizar os experimentos que serão mostrados nesse trabalho, optou-se por usar N=100. O processo de rotulação da base contou com três pesquisadores voluntários, onde cada um avaliou todos os 100 pares de códigos do dataset. Como resultado, houve 58 pares de códigos onde os três pesquisadores concordaram que não havia ocorrência de plágio, 21 pares de códigos onde os pesquisadores concordaram que houve ocorrência de plágio, e 21 pares de códigos onde não houve concordância entre os três pesquisadores. A Tabela 1 resume os números do dataset.

Tabela 1. Números do dataset. A primeira coluna indica o número de pares rotulados como plágio, a segunda rotulados como não plágio, e terceira o número de pares onde não houve concordância entre os avaliadores. A quarta coluna mostra o total de pares avaliados.

Plágio	Não plágio	Sem concordância	Total
21	58	21	100

Quando o rótulo de cada par é definido pelo voto da maioria dos avaliadores, obtém-se o resultado da Tabela 2. Nessa tabela, é considerado como plágio todo par de códigos que foi avaliado como plágio por pelo menos dois avaliadores. Da mesma forma, o rótulo não-plágio é empregado quando dois ou mais avaliadores consideraram o par como não-plágio. Todos os experimentos apresentados na Seção 5 foram feitos usando a versão do dataset apresentado na Tabela 2, onde o rótulo de cada par de códigos é definido pelo rótulo assinalado pela maioria dos avaliadores.

O dataset[4] construído para este trabalho é de domínio público, disponibilizado através da licença CC0 (Creative Commons Zero).

Tabela 2. Números do dataset, quando o rótulo é definido pela escolha da maioria dos avaliadores.

Plágio	Não plágio	Total
32	68	100

Em média, os 100 pares de códigos originais (que não passaram pela etapa de pré-processamento) selecionados para compor o dataset possuem, em média, 14,5 linhas. Isso confirma a intuição de que que os códigos desenvolvidos pelos alunos na disciplina de IPC tendem a ser simples e pequenos, conforme discutido na introdução deste artigo.

5. Resultados

Nesta seção, será apresentado os resultados do experimentos que foram conduzidos ao longo deste trabalho.

5.1. Resultados com o método MOSS

Tabela 3. Resultados obtidos com o método MOSS.

	Precisão	Revocação	Medida-F
Plágio	0.502	0.864	0.635
Não-plágio	0.926	0.668	0.776
Média	0.791	0.730	0.730

Conforme descrito na Seção 2, o método MOSS [Schleimer et al. 2003] (*Measure of Software Similarity*) foi selecionado como baseline do método proposto neste trabalho. No entanto, o propósito do MOSS não é classificar pares de códigos entre "plágio"e "não-plágio", e sim estabelecer um percentual de similaridade entre os códigos. Desta forma, para comparar o método MOSS com o método deste trabalho, foi necessário estabelecer um *threshold* de similaridade, de forma que quando o percentual de similaridade entre dois códigos-fonte retornado pelo MOSS for superior ou igual ao *threshold*, dizemos que ele foi classificado como "plágio"; caso contrário, dizemos que ele foi classificado como "não-plágio". Foram testados 5 *thresholds* de similaridade: 50%, 60%, 70%, 80% e 90%, sendo que o método MOSS apresentou melhor resultado quando usamos o *threshold* igual a 90%. Desta forma, neste trabalho, quando o MOSS retorna um índice de similaridade maior ou igual a 90%, classificamos o par como "plágio". Caso contrário, o código é classificado como "não-plágio". O resultado do método é mostrado na Tabela 3.

5.2. Balanceamento do Dataset

Conforme mostrado na Tabela 2, o dataset confeccionado durante este trabalho possui 32 pares de códigos rotulados como "plágio"e 68 pares rotulados como não plágio. Diante desse grande desbalanceamento das classes, optou-se por usar um método de sobreamostragem (oversampling) para gerar artificialmente novas instâncias da classe "plágio". Para

[4]https://codebench.icomp.ufam.edu.br/dataset/

isso, optou-se por usar o método SMOTE [Chawla et al. 2002], disponível no software Weka[5]. O método SMOTE constrói instâncias sintéticas para a classe minoritária através da seleção de instâncias existentes que estão próximas entre si no espaço de evidências, traça uma lista entre as instâncias selecionadas, e então gera uma nova instância em algum ponto aleatório da linha gerada. Foram adotados os parâmetros padrões definidos no Weka para o método SMOTE: *classValue*=0, *nearestNeighbors*=5, *percentage*=100.0 e *randomSeed*=1. Como resultado, foram criados 32 novas instâncias para a classe "plágio", totalizando 64 instâncias para essa classe.

5.3. Resultados de classificação

Foram selecionados três métodos de classificação supervisionados: *Random Forest*, *Support Vector Machine* e *AdaBoost*. Para a validação dos resultados de todos os experimentos conduzidos nesta seção, foi utilizado o *cross-validation 10-fold*. A Tabela 4 mostra os resultados para o método *Random Forest*, que foi aplicado utilizando os parâmetros padrões conforme estabelecidos no software Weka: *maxDepth*=0, *numFeatures*=0, *numTrees*=0 e *seed*=1. Como pode ser constatado na tabela, o método apresentou um valor médio de precisão, revocação e medida-F que são 4.5%, 13.1% e 13.1% maiores que os valores retornados pelo baseline, respectivamente.

Tabela 4. Resultados obtidos com o método *Random Forest*.

	Precisão	Revocação	Medida-F
Plágio	0.806	0.844	0.824
Não-plágio	0.846	0.809	0.827
Média	0.827	0.826	0.826

A Tabela 5 apresenta os resultados obtidos com o método *Support Vector Machine* (SVM). A implementação do SVM usada foi o SMO (Sequential Minimal Optimization), disponível no software Weka. A maioria dos valores de parâmetros utilizados foram os valores padrões conforme definidos no Weka: *c*=1.0, *checksTurnedOff=False*, *epsilon*=1.0E-12, *filterType=Normalize training data*, randomSeed=1. A única mudança nos parâmetros foi em relação ao *Kernel*, onde se optou por utilizar o *NormalizedPolyKernel* com os parâmetros padrões definidos no Weka: *cacheSize*=250007, *checksTurnedOff*=False, *exponent*=2.0 e *useLowerOrder*=False. Como resultado, o modelo retornou valores médios de precisão, revocação e medida-F que foram 1.9%, 10.0% e 10.0% superiores ao baseline, respectivamente.

Tabela 5. Resultados obtidos com o método *SVM*.

	Precisão	Revocação	Medida-F
Plágio	0.771	0.844	0.806
Não-plágio	0.839	0.765	0.800
Média	0.806	0.803	0.803

Por último, a Tabela 6 mostra os valores alcançados pelo método AdaBoost. A implementação utilizada foi o AdaBoostM1, disponível no software Weka. Os parâmetros

[5]https://www.cs.waikato.ac.nz/ ml/weka/

desse método também foram carregados com os valores padrões do software Weka: *numInterations*=10, *seed*=1, *useResampling*=False e *weightThreshold*=100. Como resultado, o método alcançou um valor médio de precisão, revocação e medida-F que são 4.2%, 11.1% e 10.8% maiores que os valores retornados pelo baseline, respectivamente.

Tabela 6. Resultados obtidos com o método *AdaBoost*.

	Precisão	Revocação	Medida-F
Plágio	0.753	0.906	0.823
Não-plágio	0.891	0.721	0.797
Média	0.824	0.811	0.809

Os resultados alcançados validam a hipótese H1 estabelecida na Introdução deste artigo, de que é possível identificar perfis de comportamento típicos de práticas de plágio, usando *logs* de sistemas juízes online.

6. Conclusão e Trabalhos Futuros

Este trabalho propõe um método de detecção de plágio de códigos-fonte voltado para disciplinas introdutórias de programação. O método utiliza como evidências um conjunto de variáveis capazes de representar o comportamento dos alunos durante o desenvolvimento de seus códigos, extraídas de registros de *logs* de sistemas juízes on-line. Usando essas evidências, foram testados três métodos de classificação supervisionados: *Random Forest*, *Support Vector Machine* e *AdaBoost*. O melhor resultado foi obtido com o método *Random Forest*, que alcançou um valor de medida-F igual a 0.826, que é 13.1% superior ao alcançado pelo baseline.

Como trabalhos futuros, pretende-se aplicar técnicas de aprendizagem supervisionada sobre um conjunto de evidências extraídas tanto da análise de similaridade entre pares de códigos, bem como extraídas dos *logs* de sistemas juízes online. Como o método apresentado neste trabalho usa evidências não consideradas pelas abordagens tradicionais, acredita-se que um resultado ainda melhor pode ser alcançado quando combinamos as duas fontes de evidências.

Agradecimentos

Esta pesquisa, realizada no âmbito do Projeto Samsung-UFAM de Ensino e Pesquisa (SUPER), nos termos do artigo 48 do Decreto nº 6.008/2006 (SUFRAMA), foi parcialmente financiada pela Samsung Eletrônica da Amazônia Ltda., nos termos da Lei Federal nº 8.387/1991, por meio dos convênios 001/2020 e 003/2019, firmados com a Universidade Federal do Amazonas e a FAEPI, Brasil, além do apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001 e da PROPESP/UEA através do PBICT.

Referências

Araujo, A., Zordan Filho, D. L., Oliveira, E. H. T., Carvalho, L. S. G., Pereira, F. D., and Oliveira, D. B. F. (2021). Mapeamento e análise empírica de misconceptions comuns em avaliações de introdução à programação. In *Anais do Simpósio Brasileiro de Educação em Computação*, pages 123–131. SBC.

Charikar, M. S. (2002). Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, page 380–388, New York, NY, USA. Association for Computing Machinery.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357.

Costa, T. L., de Oliveira, E. H. T., Passito, A., de Souza Pinto, M. A., de Carvalho, L. S. G., de Oliveira, D. B. F., and Pereira, F. D. (2021). Material didático interativo para a disciplina de introdução à programação de computadores. In *Anais Estendidos do Simpósio Brasileiro de Educação em Computação*, pages 41–42. SBC.

de Oliveira, J., Salem, F., de Oliveira, E. H. T., Oliveira, D. B. F., de Carvalho, L. S. G., and Pereira, F. D. (2020). Os estudantes leem as mensagens de feedback estendido exibidas em juízes online? In *Anais do XXXI Simpósio Brasileiro de Informática na Educação*, pages 1723–1732. SBC.

Dwan, F., Oliveira, E., and Fernandes, D. (2017). Predição de zona de aprendizagem de alunos de introdução à programação em ambientes de correção automática de código. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*, volume 28, page 1507.

Fonseca, S., Oliveira, E., Pereira, F., Fernandes, D., and Carvalho, L. S. G. (2019). Adaptação de um método preditivo para inferir o desempenho de alunos de programação. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*, volume 30, page 1651.

Freitas Júnior, H. B., Pereira, F. D., Oliveira, E. H. T., Oliveira, D. B. F., and Carvalho, L. S. G. (2020). Recomendação automática de problemas em juízes online usando processamento de linguagem natural e análise dirigida aos dados. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação*, pages 1152–1161. SBC.

Galvão, L., Fernandes, D., and Gadelha, B. (2016). Juiz online como ferramenta de apoio a uma metodologia de ensino híbrido em programação. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*, volume 27, page 140.

Gomes, K. and Matos, S. (2020). Detection of programming plagiarism in computing education: A systematic mapping study. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação*, pages 1633–1642, Porto Alegre, RS, Brasil. SBC.

Lima, M., Carvalho, L. S. G., de Oliveira, E. H. T., Oliveira, D. B. F., and Pereira, F. D. (2020). Classificação de dificuldade de questões de programação com base em métricas de código. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação*, pages 1323–1332. SBC.

Lima, M. A. P., Carvalho, L. S. G., de Oliveira, E. H. T., Oliveira, D. B. F., and Pereira, F. D. (2021). Uso de atributos de código para classificação da facilidade de questões de codificação. In *Anais do Simpósio Brasileiro de Educação em Computação*, pages 113–122. SBC.

Novak, M., Joy, M., and Kermek, D. (2019). Source-code similarity detection and detection tools used in academia: A systematic review. *ACM Trans. Comput. Educ.*, 19(3).

Oliveira, A. M. d. et al. (2016). Um método de detecção de plágio em códigos-fonte para disciplinas iniciais de programação.

Pereira, F. D., Fonseca, S. C., Oliveira, E. H., Cristea, A. I., Bellhäuser, H., Rodrigues, L., Oliveira, D. B., Isotani, S., and Carvalho, L. S. (2021a). Explaining individual and collective programming students' behaviour by interpreting a black-box predictive model. *IEEE Access*.

Pereira, F. D., Fonseca, S. C., Oliveira, E. H., Oliveira, D. B., Cristea, A. I., and Carvalho, L. S. (2020a). Deep learning for early performance prediction of introductory programming students: a comparative and explanatory study. *Brazilian journal of computers in education.*, 28:723–749.

Pereira, F. D., Oliveira, E., Cristea, A., Fernandes, D., Silva, L., Aguiar, G., Alamri, A., and Alshehri, M. (2019a). Early dropout prediction for programming courses supported by online judges. In *International Conference on Artificial Intelligence in Education*, pages 67–72. Springer.

Pereira, F. D., Oliveira, E. H., Oliveira, D. B., Cristea, A. I., Carvalho, L. S., Fonseca, S. C., Toda, A., and Isotani, S. (2020b). Using learning analytics in the amazonas: understanding students' behaviour in introductory programming. *British Journal of Educational Technology*.

Pereira, F. D., Oliveira, E. H. T., Fernandes, D., and Cristea, A. (2019b). Early performance prediction for cs1 course students using a combination of machine learning and an evolutionary algorithm. In *2019 IEEE 19th International Conference on Advanced Learning Technologies (ICALT)*, volume 2161-377X, pages 183–184.

Pereira, F. D., Pires, F., Fonseca, S. C., Oliveira, E. H., Carvalho, L. S., Oliveira, D. B., and Cristea, A. I. (2021b). Towards a human-ai hybrid system for categorising programming problems. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 94–100.

Pereira, F. D., Souza, L. M., Oliveira, E. H. T., Oliveira, D. B. F., and Carvalho, L. S. G. (2020c). Predição de desempenho em ambientes computacionais para turmas de programação: um mapeamento sistemático da literatura. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação*, pages 1673–1682. SBC.

Ril Gil, Y., Toll Palma, Y. d. C., and Lahens, E. F. (2014). Determination of writing styles to detect similarities in digital documents. *RUSC: Revista de Universidad y Sociedad del Conocimiento*, 11(1).

Rossi, R. G. (2016). *Classificaçao automática de textos por meio de aprendizado de máquina baseado em redes*. PhD thesis, Universidade de São Paulo.

Santos, I. L., Oliveira, D. B. F., Carvalho, L. S. G., Pereira, F. D., and Oliveira, E. H. T. (2020). Tempos de transição em estados de corretude e erro como indicadores de desempenho em juízes online. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação*, pages 1283–1292. SBC.

Schleimer, S., Wilkerson, D. S., and Aiken, A. (2003). Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, page 76–85, New York, NY, USA. Association for Computing Machinery.

Su, Z., Ahn, B.-R., Eom, K.-Y., Kang, M.-K., Kim, J.-P., and Kim, M.-K. (2008). Plagiarism detection using the levenshtein distance and smith-waterman algorithm. In *2008 3rd International Conference on Innovative Computing Information and Control*, pages 569–569.

Tao, G., Guowei, D., Hu, Q., and Baojiang, C. (2013). Improved plagiarism detection algorithm based on abstract syntax tree. In *Emerging Intelligent Data and Web Technologies (EIDWT), 2013 Fourth International Conference on*, pages 714–719.

Tian, Z., Zheng, Q., Liu, T., Fan, M., Zhuang, E., and Yang, Z. (2015). Software plagiarism detection with birthmarks based on dynamic key instruction sequences. *IEEE Transactions on Software Engineering*, 41:1–1.

Wan, H., Liu, K., and Gao, X. (2018). Token-based approach for real-time plagiarism detection in digital designs. In *2018 IEEE Frontiers in Education Conference (FIE)*, pages 1–5.

Wasik, S., Antczak, M., Badura, J., Laskowski, A., and Sternal, T. (2018). A survey on online judge systems and their applications. *ACM Comput. Surv.*, 51(1).

Watanobe, Y., Chowdhury, I., Cortez, R., and Vazhenin, A. (2020). Next-generation programming learning platform: Architecture and challenges. *SHS Web of Conferences*, 77:01004.

Yuan, B., Wang, J., Fang, Z., and Qi, L. (2018). A new software birthmark based on weight sequences of dynamic control flow graph for plagiarism detection. *The Computer Journal*, 61(8):1202–1215.

Yuan, Y. and Guo, Y. (2012). Boreas: An accurate and scalable token-based approach to code clone detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 286–289, New York, NY, USA. ACM.