

Limitações da avaliação baseada em testes: quando códigos não atendem à enunciados de questões de programação em juízes on-line

Rafaela Melo^{1,2}, Davi Aguiar¹, Fernanda Pires¹, Leandro Galvão²,
Horácio Fernandes², Marcela Pessoa¹, David Fernandes²

¹Escola Superior de Tecnologia – Universidade do Estado do Amazonas (EST – UEA)
Manaus – AM – Brasil

²Instituto de Computação – Universidade Federal do Amazonas (ICOMP – UFAM)
Av. General Rodrigo Octávio, 6200, Coroado I – 69080-900 – Manaus – AM – Brasil

{dam.eng23, fpires, mspessoa}@uea.edu.br

{rmelo, galvao, horacio, david}@icompu.fam.edu.br

Abstract. Introduction: Online judges are widely used in programming courses. These systems operate based on test cases, executing the code and indicating whether it is correct or not. Thus, other aspects that may be relevant to learning are not taken into account. For example, it is not possible to identify whether the student's code followed what is requested in the question statement. **Objective:** This work presents a method to detect cases in which the code did not meet the requested requirements. **Methodology:** To this end, machine learning techniques were used. **Results:** The results demonstrated that, among solutions classified as correct, there are cases in which the requirements of the question were not met, highlighting limitations in test-based assessment.

Keywords: Online Judges, Computer Science Education, Programming Learning

Resumo. Introdução: Juízes on-line são muito utilizados em disciplinas de programação. Esses sistemas funcionam com base em casos de teste, executando o código e indicando se está correto ou não. Assim, não são levados em consideração outros aspectos que podem ser relevantes para a aprendizagem. Um exemplo disso, é que não é possível identificar se o código do estudante seguiu o que é solicitado no enunciado da questão. **Objetivo:** Este trabalho apresenta um método para detectar casos em que o código não cumpriu os requisitos solicitados. **Metodologia:** Para tanto, foram utilizadas técnicas de aprendizagem de máquina. **Resultados:** Os resultados indicam que, entre soluções classificadas como corretas, existem casos em que os requisitos da questão não foram cumpridos, evidenciando limitações na avaliação baseada em testes.

Palavras-Chave: Juízes On-line, Educação em Computação, Aprendizagem de Programação

1. Introdução

Em disciplinas de programação é comum o uso de sistemas de avaliação automatizada de códigos, como juízes on-line. Esses sistemas facilitam o trabalho do professor, ao agilizar

a correção de exercícios, principalmente em turmas com grande quantidade de estudantes [Wasik et al. 2018]. Juízes on-line funcionam com base em casos de teste previamente definidos para cada exercício. Quando um estudante submete uma solução, o sistema compila e executa o código utilizando diferentes entradas de teste e compara as saídas produzidas com as saídas esperadas [Liu et al. 2023]. Caso os resultados estejam corretos para todos os casos de teste, a submissão é considerada correta; caso contrário, a solução é identificada como incorreta [Barbosa et al. 2023].

Nesse sentido, a avaliação fica restrita à corretude funcional da solução, deixando de considerar aspectos igualmente relevantes para o processo de aprendizagem, como a qualidade do código e outros critérios pedagógicos estabelecidos pelo professor [Melo et al. 2026]. Além disso, uma solução pode ser aprovada em todos os casos de teste e, ainda assim, não atender integralmente aos requisitos estabelecidos no enunciado. Um exemplo disso são questões que pedem o uso de determinado algoritmo de ordenação, como o *Merge Sort*, mas que acabam sendo resolvidas com algoritmos mais simples, como o *Bubble Sort*, ou até mesmo com o uso de funções prontas da linguagem. Isso acontece porque os casos de teste cadastrados no juiz online são capazes de avaliar apenas o resultado final, sem levar em consideração a estratégia usada pelo aluno para se chegar a essa solução final. Esse tipo de limitação, já discutida em trabalhos anteriores sobre avaliação automática [Ihantola et al. 2010, Keuning et al. 2018], revela que os juízes on-line garantem a correção sintática e funcional, mas não asseguram que os requisitos das questões sejam de fato atendidos.

Tais limitações têm impacto direto no acompanhamento da aprendizagem. A ausência de instrumentos que analisem como o estudante pensa e organiza a solução torna mais difícil reconhecer padrões de erro, desvios de raciocínio ou até mesmo identificar casos em que os conceitos não foram compreendidos pelos estudantes (*misconceptions*), que acabam buscando uma alternativa diferente para solucionar o problema [Figueras et al. 2025]. Em turmas grandes, a revisão manual de códigos é inviável [Alves e Jaques 2014], o que reforça a importância de buscar alternativas automatizadas ou semiautomatizadas para analisar a estrutura e a lógica dos programas, e não apenas suas saídas [Pimentel et al. 2025].

Com base nesse contexto, este trabalho buscou avaliar a viabilidade de utilizar técnicas de aprendizagem de máquina para detectar códigos submetidos por estudantes em sistemas juízes on-line que não atendem as especificações previstas em enunciados de problemas de programação. De forma exploratória, investigou-se se seria possível identificar, entre soluções funcionalmente corretas, aquelas que apresentavam estratégias divergentes em relação ao que foi solicitado no enunciado. Como contribuições, este trabalho traz: i) proposição de um método para identificar códigos que não seguem os requisitos do enunciado, mesmo sendo classificados como corretos por juízes on-line; e, ii) investigação das limitações da avaliação baseada exclusivamente em casos de teste, evidenciando que a corretude funcional não garante aderência aos objetivos pedagógicos da questão.

2. Referencial Teórico e Trabalhos Relacionados

Sistemas de avaliação automatizada de código são comumente utilizados em disciplinas de programação como ferramentas de apoio pedagógico [Carvalho et al. 2016]. Esses sistemas facilitam o trabalho do professor, porém possuem um conjunto de limitações

amplamente discutidas na literatura.

No trabalho de Barbosa *et al.* (2023) foi realizada uma investigação acerca dos processos de avaliação que são utilizados por professores e monitores em disciplinas de programação. Os autores aplicaram um questionário com 37 pessoas, para identificar que tipos de ferramentas são utilizadas nas disciplinas e quais características são observadas nos códigos dos estudantes. Os resultados demonstraram que, apesar de ser importante no contexto de ensino e aprendizagem, o uso de juízes on-line possui limitações. Para avaliar somente a corretude funcional do código, esses sistemas são suficientes. Porém, ao considerar outros aspectos, como a aplicabilidade de estruturas de repetição, decisão e operadores lógicos, são necessárias ferramentas auxiliares. Nesse sentido, os autores defendem o uso de um “VAR”, ou seja, um meio auxiliar para avaliar aspectos específicos das soluções.

Figueras *et al.* (2025) realizaram um estudo com especialistas em sistemas de avaliação automatizada, como juízes on-line, para levantar os benefícios e desvantagens de utilizar tais sistemas. Entre os benefícios apontados, destacam-se: i) a facilitação do trabalho do professor, principalmente ao considerar turmas numerosas; e ii) a promoção da autonomia dos estudantes, ao permitir que os códigos sejam testados antes do envio final. Como desvantagens, ao utilizar esses sistemas, os estudantes podem se preocupar apenas em cumprir os requisitos mínimos de uma questão, ou até se preocupar somente em passar nos casos de teste. Nesse contexto, uma das limitações discutidas é o fato desses sistemas considerarem apenas a corretude funcional do código e não outros aspectos relevantes para a aprendizagem de programação.

Diante disso, alguns trabalhos têm investigado o uso de ferramentas complementares aos juízes on-line, com o objetivo de fornecer *feedback* mais rico sobre as soluções dos estudantes. Entre essas iniciativas, destacam-se abordagens que permitem identificar similaridades entre soluções e possíveis indícios de plágio [de Oliveira et al. 2021]. Outras propostas, como o OverCode [Glassman et al. 2015], buscam agrupar códigos semelhantes para facilitar o *feedback* em larga escala. Já em Zhang *et al.* (2022) é apresentado o Clef, uma ferramenta que gera *feedback* para códigos incorretos por meio da produção de versões corrigidas das soluções, em vez de apenas indicar se o código está correto ou não.

Apesar desses avanços, tais ferramentas ainda apresentam limitações no que diz respeito à verificação da conformidade das soluções com os requisitos descritos nos enunciados. Dessa forma, o presente trabalho explora o uso de técnicas de aprendizagem de máquina para identificar códigos que, embora possam ser considerados corretos por juízes on-line, não seguem os requisitos solicitados nos enunciados de questões de programação.

3. Metodologia

Este estudo possui caráter exploratório e investigativo, buscando analisar a viabilidade do uso de técnicas de cálculo de similaridade e aprendizado de máquina para identificar códigos que não seguem as especificações definidas em enunciados de questões de programação. Para isso, foi estruturado um fluxo metodológico composto por etapas de preparação dos dados, representação textual do código, aplicação de técnicas de detecção e análise qualitativa dos resultados, conforme indica a Figura 1.

Neste trabalho, parte-se da hipótese de que códigos que tentam apenas atender aos casos de teste, sem cumprir as regras do enunciado, tendem a diferir significativamente

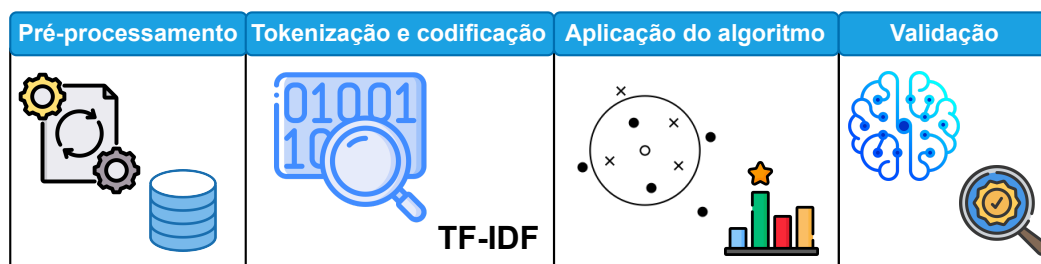


Figura 1. Visão geral do método.

daqueles submetidos pelos demais alunos. Com base nessa premissa, o termo anomalia é utilizado para designar submissões cujas estratégias ou implementações divergem substancialmente das demais soluções de um mesmo exercício.

Para investigar as diferenças entre os códigos, foi aplicado o algoritmo *K-Nearest Neighbors* (KNN) [Aha et al. 1991] para gerar *rankings* de similaridade entre códigos e a amplitude interquartil (AIQ) [Vinutha et al. 2018] para detectar anomalias dentro desses *rankings*. Cada uma das etapas está descrita a seguir.

3.1. Pré-processamento

Antes das etapas de análise, os códigos passam por uma etapa de pré-processamento, visando reduzir ruídos textuais e preservar apenas os elementos relevantes para a lógica de programação. Como as submissões são produzidas por diferentes estudantes, os códigos apresentam grande variabilidade em termos de formatação, comentários, uso de bibliotecas e padrões de estilo, o que pode interferir no cálculo de similaridade, ao introduzir diferenças que não dizem respeito à forma como os problemas foram resolvidos. Para minimizar esses efeitos, devem ser removidos comentários, textos entre aspas, declarações de bibliotecas e espaços redundantes.

O processo segue abordagens utilizadas em trabalhos de detecção de plágio em código-fonte [Oliveira et al. 2016, de Oliveira et al. 2021], que indicam que a remoção desses fragmentos de código contribui para aumentar a precisão na comparação entre soluções. Assim, busca-se preservar apenas os componentes que caracterizam a lógica da implementação, como estruturas de controle (*if*, *for*, *while*, *switch*), tipos de dados, operações, chamadas de função e demais instruções fundamentais para a estratégia de resolução [Burrows e Tahaghoghi 2007].

3.2. Tokenização e Indexação

Após o pré-processamento, os códigos são transformados em sequências de *tokens* com o objetivo de representar, de forma padronizada, a estrutura sintática das soluções. A tokenização é uma das principais etapas em tarefas de análise automatizada de código, pois reduz a variabilidade textual entre implementações e permite que métodos baseados em similaridade capturem padrões estruturais em vez de características individuais de escrita [Burrows e Tahaghoghi 2007].

O processo consiste em segmentar cada programa em fragmentos correspondentes a elementos sintáticos, como palavras-chave, operadores e delimitadores, e substituí-los por símbolos genéricos contendo apenas dois caracteres. Essa abordagem segue diretrizes apresentadas em pesquisas sobre detecção de similaridade em código-fonte

[de Oliveira et al. 2021], nas quais a tokenização desempenha o papel de abstrair detalhes dispensáveis, como nomes de variáveis ou formatação, mantendo somente a lógica da solução. A Tabela 1 apresenta alguns exemplos de fragmentos de código e seus respectivos símbolos.

Tabela 1. Exemplos de símbolos e fragmentos de código.

Fragmento	Símbolo
int	tb
float	ic
double	rd
char	ub
(	aa
[	ca

Após a tokenização, as sequências resultantes são convertidas em N-gramas utilizando a técnica de janela deslizante (*sliding window*). N-gramas correspondem a subsequências contínuas extraídas de uma sequência maior. Neste trabalho, utiliza-se $N = 5$, de modo que cada N-grama seja composto por cinco *tokens* consecutivos do código.

A janela deslizante consiste em mover uma janela fixa ao longo da sequência original, avançando um *token* por vez e gerando, em cada posição, um novo N-grama. Esse procedimento garante sobreposição entre as subsequências produzidas e preserva as transições originais entre instruções do código. O valor $N = 5$ é escolhido por oferecer um equilíbrio adequado entre granularidade e contexto: valores menores tendem a capturar pouca informação relevante, enquanto valores muito grandes aumentam a especificidade da representação e resultam em vetores esparsos, diminuindo a capacidade de generalização.

Em seguida, cada conjunto de N-gramas é convertido em vetores numéricos por meio do TF-IDF (*Term Frequency–Inverse Document Frequency*) [Sparck Jones 1972]. O TF-IDF é utilizado por ser um método simples, eficiente e amplamente utilizado em tarefas de mineração de texto, permitindo mensurar a relevância de cada N-grama no conjunto de documentos (códigos). Ao final dessa etapa, cada código-fonte é representado como um vetor numérico em um espaço dimensional único formado pelo conjunto total de N-gramas distintos da base. Assim, os códigos podem ser comparados por meio de métricas de distância, e essa representação é utilizada como entrada para o algoritmo KNN na etapa seguinte.

A Figura 2 apresenta um exemplo simplificado do *pipeline* de pré-processamento, tokenização e representação vetorial utilizado neste trabalho.

3.3. Aplicação do Algoritmo

Após a tokenização e indexação, com os códigos sendo representados de maneira vetorial, é adotada uma abordagem baseada em métricas de similaridade para detectar soluções que apresentam padrões estruturalmente distintos do conjunto de submissões de uma mesma questão. Essa etapa tem como finalidade identificar anomalias potenciais, ou seja, códigos cuja lógica ou organização sugerem desvios conceituais em relação às estratégias mais comumente utilizadas pelos estudantes.

A detecção é realizada em duas etapas complementares: (i) cálculo da similaridade por meio do algoritmo *K-Nearest Neighbors* (KNN) [Aha et al. 1991]; e (ii) identificação

Código original

```
#include <stdio.h>

// leitura do valor
int main() {
    int n;
    printf("Digite n:");
    scanf("%d", &n);

    if (n % 2 == 0) {
        printf("par");
    }
}
```

Após o pré-processamento

```
int main() {
    int n;
    scanf("%d", &n);
    if(n % 2 == 0){ }
}
```

Tokenização

```
tb fn aa fa ac tb id
fn aa str op id fa
if aa id op nm op
nm fa ac fc fc
```

N-gramas (N=5)

```
[tb fn aa fa ac]
[fn aa fa ac tb]
[aa fa ac tb id]
[fa ac tb id fn]
...
```

Representação TF-IDF

```
[0.12, 0.45, 0.00, ...]
```

Figura 2. Exemplo simplificado das etapas de pré-processamento, tokenização e representação vetorial.

de valores atípicos utilizando a amplitude interquartil (AIQ) [Vinutha et al. 2018].

O algoritmo KNN, tradicionalmente utilizado para classificação baseada em vizinhança, é utilizado aqui em sua forma não supervisionada, atuando como um estimador de densidade local. A ideia de utilizar o KNN é que códigos resolvidos segundo estratégias semelhantes tendem a ocupar regiões próximas no espaço vetorial, enquanto soluções que adotam raciocínios diferentes se afastam dessas regiões.

Nesse contexto, o KNN calcula, para cada código, a média das distâncias em relação aos seus k vizinhos mais próximos. Valores altos de distância indicam que o código está isolado ou pouco representado na vizinhança local, o que o torna candidato a uma anomalia. Já valores baixos indicam que o código se encontra em uma região densa, compartilhando características comuns com outras implementações.

A métrica utilizada para o cálculo das distâncias é a distância do cosseno, comumente utilizada em tarefas de comparação textual por capturar adequadamente a similaridade na orientação dos vetores TF-IDF. A escolha do parâmetro k é um aspecto relevante, pois valores muito baixos tendem a tornar o método excessivamente sensível a ruídos locais, enquanto valores muito altos podem suavizar diferenças estruturais importantes entre as representações. Considerando o caráter exploratório do estudo, bem como o número relativamente reduzido de submissões por questão, opta-se por utilizar $k = 5$. Esse valor, frequentemente adotado como configuração inicial em métodos baseados em vizinhança, oferece um equilíbrio adequado entre sensibilidade e estabilidade na detecção de similaridades.

Com as médias de distância calculadas, é aplicada a amplitude interquartil (AIQ) para identificar valores atípicos. A AIQ é uma técnica estatística robusta, muito utilizada em tarefas de detecção de anomalias, por se basear nos quartis da distribuição [Vinutha et al. 2018]. Nesse sentido, valores que se encontram fora dos limites definidos a

partir do intervalo interquartil são interpretados como potenciais anomalias.

Embora, em análises tradicionais de dados numéricos, a identificação de valores atípicos frequentemente se concentre em extremos da distribuição, neste estudo o foco é identificar valores mais distantes dos demais, independentemente da direção. Ou seja, soluções que aparecem distantes de regiões densas (formadas pelos códigos mais similares entre si) podem representar implementações que fugiram das especificações do problema.

3.4. Validação

Após a etapa de detecção preliminar das anomalias, é necessário verificar se os códigos identificados como estruturalmente distintos correspondem, de fato, a casos em que os requisitos solicitados no enunciado da questão não foram atendidos. Para isso, é realizada uma validação qualitativa manual, identificando se a solução segue ou não as especificações da questão, além de observar detalhes de implementação que podem ser relevantes do ponto de vista pedagógico, como por exemplo, aspectos relacionados a estratégia de programação adotada pelo estudante.

Os LLMs têm se mostrado úteis em tarefas relacionadas ao processamento de linguagem natural e à interpretação de códigos-fonte [Nijkamp et al. 2023]. No contexto desta pesquisa, essa característica é particularmente relevante, uma vez que a avaliação proposta envolve a interpretação dos enunciados das questões e a análise das soluções submetidas pelos estudantes. Além disso, trabalhos recentes têm investigado o uso de LLMs em disciplinas de programação [Raihan et al. 2025], indicando o potencial desses modelos para apoiar atividades educacionais relacionadas à análise de código. Com base nessas evidências, os LLMs podem ser utilizados como apoio na etapa de validação qualitativa deste estudo, porém, com um humano validando a avaliação feita pelo modelo.

4. Resultados e discussão

Esta seção apresenta os resultados obtidos a partir da aplicação do método apresentado, incluindo a base de dados utilizada na pesquisa.

4.1. Base de dados

A base de dados utilizada no estudo é composta por códigos na linguagem C submetidos ao juiz on-line CodeBench¹ por estudantes das disciplinas Algoritmos e Estruturas de Dados I e Algoritmos e Estruturas de Dados II da Universidade Federal do Amazonas (UFAM).

Para a realização do experimento, foram selecionadas oito questões de programação, abordando temas como ordenação, manipulação de vetores e matrizes, uso de funções e recursividade. A escolha das questões ocorreu por conveniência, considerando critérios como a clareza dos enunciados, diversidade de estruturas algorítmicas solicitadas e presença de restrições explícitas (por exemplo, a obrigatoriedade do uso de determinado algoritmo ou técnica). As questões foram selecionadas dessa forma para facilitar a verificação de conformidade entre o enunciado e o que foi efetivamente implementado pelo estudante.

Cada questão conta com múltiplas submissões de diferentes estudantes, conforme mostrado na Figura 3. Apesar de ser um número pequeno de questões, foi possível avaliar, ainda que em escala reduzida, a aplicabilidade dos métodos de detecção de anomalias em cenários reais de ensino de programação.

¹<https://cb.icomp.ufam.edu.br/>

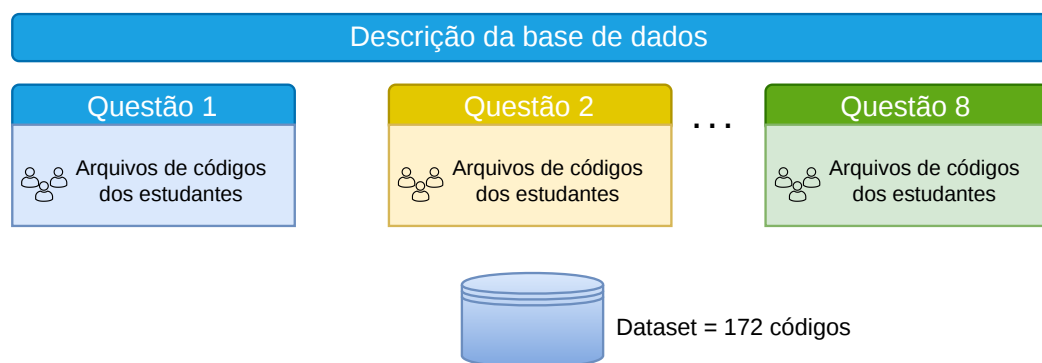


Figura 3. Base de dados utilizada no estudo.

4.2. Análise dos resultados

Foi realizada uma validação qualitativa manual, conduzida por um estudante do sétimo período do curso de Engenharia da Computação, com experiência em desenvolvimento de software. Cada código identificado como anômalo foi inspecionado individualmente, e os resultados da análise foram registrados em uma planilha, indicando se a solução atendia ou não aos requisitos da questão, além de observações sobre as escolhas realizadas no código para resolver os problemas.

Em alguns casos, um *Large Language Model* (LLM) foi utilizado como ferramenta de apoio à análise, especialmente em códigos extensos que dificultavam a inspeção manual. O LLM que serviu de apoio foi o Claude Sonnet 4.6², que também auxiliou na investigação dos motivos que levaram determinados códigos, mesmo corretos em relação às especificações do enunciado, a serem identificados como anômalos.

A interação com o modelo seguiu um formato de *prompt* simples, composto pela apresentação do enunciado da questão e do código submetido pelo estudante, seguidos da pergunta objetiva "Esse código segue o enunciado?", sem o uso de técnicas adicionais de engenharia de *prompt*. Nesses casos, o modelo contribuiu para a compreensão de particularidades de implementação que poderiam justificar sua distinção em relação às demais soluções. Por fim, a validação realizada pelo estudante foi posteriormente revisada e validada por uma professora com experiência na área de programação, sendo ambos autores deste trabalho.

A Tabela 2 apresenta os resultados quantitativos da análise manual realizada. A partir das anomalias identificadas pela aplicação do método, apresentadas na coluna "Anomalias", foi realizada a inspeção individual dos códigos com o objetivo de verificar quais, de fato, não atendiam às especificações descritas nos enunciados das questões. Os casos confirmados são apresentados na coluna "Não seguem o enunciado". É possível perceber que, em apenas duas das questões analisadas, mais da metade das anomalias identificadas correspondiam, de fato, a códigos onde os requisitos da questão não foram cumpridos.

A Tabela 3 apresenta um resumo das anomalias observadas nas oito questões de programação selecionadas, categorizando os desvios identificados em dois grupos: i) códigos que não atenderam aos requisitos especificados no enunciado da questão; e ii)

²<https://claude.ai>

Tabela 2. Número de anomalias identificadas por questão.

Questão	Quantidade de códigos	Anomalias (%)	Não seguem o enunciado (%)
1	30	40.00%	25.00%
2	23	21.74%	20.00%
3	21	23.81%	60.00%
4	15	33.33%	40.00%
5	15	33.33%	100.00%
6	19	15.79%	33.33%
7	32	25.00%	25.00%
8	17	29.41%	20.00%

casos especiais, nos quais as soluções apresentaram características significativamente diferentes das abordagens mais comumente utilizadas para resolver os problemas.

Tabela 3. Resumo das análises realizadas sobre as anomalias.

Enunciado resumido	Não seguem o enunciado	Casos especiais
Alguém deixou o quadro de medalhas das olimpíadas fora de ordem. Seu programa deve colocá-lo na ordem correta. Utilizem o <i>Selection Sort</i> para fazer a ordenação.	Uso de função própria da linguagem (<i>qsort</i>).	Abordagem com <i>loops</i> aninhados e várias condições, códigos extensos e verbosos, uso de listas encadeadas e alocação de memória.
Implemente uma função que receba uma matriz quadrada NxN de inteiros e um número inteiro K, retornando verdadeiro caso a matriz possua exatamente 4 posições com valor igual a K, e falso caso contrário.	Uso incorreto da variável de controle do laço de repetição.	Inversão dos valores retornados pela função.
Implemente uma função que receba como parâmetro um vetor de números inteiros de tamanho N e inverta a ordem dos elementos armazenados nesse vetor.	Apenas impressão em vez de inversão, criação de mais funções que o solicitado.	Lógica de ponteiros e uso de vetor auxiliar.
Faça um programa que leia um número inteiro, inverta seus dígitos e o imprima. Para inverter os dígitos do número crie uma função que receba o endereço de uma variável do tipo <i>int</i> e coloque o número com os dígitos invertidos no endereço passado como parâmetro.	Apenas impressão em vez de inversão.	Utilização de variável auxiliar, aumento da complexidade (uso de três estruturas de repetição) e conversão de inteiro para string para realizar a inversão dos dígitos.
Implemente uma função que receba um vetor ordenado de inteiros e uma chave, também inteira. A função deve retornar verdadeiro caso a chave ocorra exatamente 1 vez no vetor, retornando falso em caso contrário. Utilize busca binária.	Implementação incorreta da busca binária; Implementação de busca linear.	Sem casos especiais.
Implemente o algoritmo de ordenação por inserção em uma função que receba como parâmetros um vetor de inteiros e seu tamanho. A ordenação deve ser em ordem decrescente e sem a utilização do comando “for”.	Implementação do <i>Selection Sort</i> , uso do “for”.	Implementação do <i>Insertion Sort</i> com trocas adjacentes.
Escreva um função recursiva que retorne o valor do i-ésimo termo da série de Fibonacci. Faça uma função <i>main</i> para imprimir os primeiros 13 termos da série de Fibonacci.	Exibição dos valores em função diferente da solicitada e uso de iteração ao invés de recursão.	Uso de variáveis não inicializadas e condição ternária.
Faça uma função que receba como parâmetros um nome e uma letra. A função deve retornar a quantidade de vezes que a letra passada como parâmetro ocorre no nome. Não use qualquer função de manipulação de strings pré-existente em sua função.	Uso de funções de manipulação de string e criação de função própria para verificar o tamanho da string.	Uso de “for” ao invés de “while”.

De modo geral, verificou-se que a maioria das anomalias correspondeu a substituições explícitas das técnicas solicitadas. Em questões que exigiam algoritmos específicos, como *Insertion Sort* ou busca binária, foram identificadas soluções que utilizavam métodos mais simples, como busca linear ou até mesmo funções prontas da linguagem, como *qsort*, uma função da biblioteca `<stdlib.h>` que organiza os elementos de um *array* usando o algoritmo *Quick Sort*. Essas substituições indicam que, mesmo quando o enunciado apresenta um requisito específico, muitos estudantes acabam optando por estratégias alternativas que consideram mais fáceis ou que já dominam, principalmente quando não há mecanismos

automáticos de correção capazes de identificar ou penalizar isso. O juiz on-line, ao avaliar apenas a saída, leva em conta apenas a funcionalidade e não outros aspectos do código.

Outro padrão observado está relacionado ao uso de recursividade. Na questão em que o professor solicitava a implementação de uma solução baseada em recursão, identificou-se códigos que usavam recursos iterativos em vez de recursivos. Essa escolha pode estar relacionada tanto a limitações conceituais, dada a dificuldade comum dos alunos em compreender o conceito de recursividade, quanto à percepção de que estruturas iterativas são mais intuitivas. No entanto, embora a solução iterativa passe nos casos de teste cadastrados no juiz on-line, ela não atende ao objetivo pedagógico da atividade, que é justamente aplicar o conceito de recursividade.

Em algumas questões, as anomalias não envolveram a troca direta da técnica exigida, mas sim simplificações da solução, que acabaram fugindo da intenção original do exercício. Em problemas que exigiam manipulação de vetores, por exemplo, apareceram códigos que apenas imprimiam o vetor invertido, sem realizar efetivamente a operação, ou seja, inverter os valores do vetor. Da mesma forma, soluções que exigiam manipulação de ponteiros ou uso de funções específicas acabaram sendo implementadas com estruturas mais complexas do que era necessário, o que pode sugerir tentativas de contornar dificuldades conceituais. Assim, estratégias excessivamente elaboradas, observadas em algumas das anomalias, não eram necessariamente eficientes e podiam indicar falta de domínio do conteúdo.

Nos casos identificados como especiais, percebeu-se que alguns códigos cumpriam com o que era solicitado pelo enunciado, porém possuíam características que podem ter levado o método a classificá-los como anomalias. Nesse cenário o LLM foi útil para esclarecer quais detalhes eram diferentes das abordagens geralmente utilizadas para resolver determinados problemas. Por exemplo, a implementação do *Insertion Sort* com trocas adjacentes (*swap-based insertion*), que é uma variação do algoritmo onde os elementos são movidos para suas posições ordenadas trocando repetidamente os itens adjacentes, em vez de deslocá-los. Na questão onde era solicitado para ler um número inteiro, inverter os dígitos e imprimir, um dos códigos invertia o inteiro para *string* para realizar a inversão dos dígitos, sendo uma abordagem não muito convencional para o problema.

O estudo permitiu observar que o método foi capaz de revelar padrões sobre o comportamento dos estudantes, como a preferência por soluções mais simples, a dificuldade com determinadas técnicas e a tendência de contornar requisitos quando não há mecanismos de verificação, como é o caso dos juízes on-line. Além disso, nos casos especiais percebe-se que existem diversas estratégias para solucionar um mesmo problema e é interessante, pedagogicamente, entender essa diversidade. Esses achados reforçam a importância de ferramentas que consigam identificar não apenas se o código funciona, mas como ele foi construído, permitindo ao professor compreender lacunas conceituais e até mesmo o nível de aprendizagem da turma.

Além disso, os resultados mostraram que a abordagem baseada em TF-IDF e similaridade de código, embora limitada na captura de aspectos semânticos, foi suficiente para apontar códigos que divergiam das soluções predominantes. Um possível trabalho futuro é verificar se, ao utilizar outras formas de representação de código, que sejam capazes de capturar aspectos semânticos, os resultados são semelhantes ou melhores.

A análise qualitativa indicou que parte das anomalias identificadas correspondia a códigos que divergiam, de alguma forma, dos requisitos definidos nas questões, sugerindo a importância de ferramentas capazes de avaliar a corretude das soluções e as estratégias utilizadas pelos estudantes durante a resolução. Além disso, em dois dos códigos analisados, observou-se que a abordagem empregada não parecia adequada para resolver o problema proposto, embora as soluções tenham sido aprovadas nos casos de teste do juiz on-line. Esse resultado reforça uma limitação comum da avaliação baseada exclusivamente em testes, já que os conjuntos de entrada e saída nem sempre contemplam todas as possibilidades relevantes. Nesse contexto, os resultados sugerem que alguns estudantes podem direcionar seus esforços prioritariamente para satisfazer os casos de teste do sistema, sem necessariamente desenvolver soluções alinhadas aos objetivos conceituais da atividade, como já discutido em outros trabalhos na literatura [Figueras et al. 2025].

5. Limitações

Apesar de haver evidências da viabilidade da abordagem proposta, algumas limitações importantes precisam ser consideradas na interpretação dos resultados. Uma primeira limitação está associada ao uso de representações baseadas em TF-IDF, que capturam apenas aspectos estruturais e superficiais do código. Embora essa representação tenha sido útil para identificar diferenças sintáticas, ela não consegue expressar, de maneira satisfatória, aspectos semânticos associados à lógica de programação. Assim, códigos estruturalmente semelhantes podem empregar estratégias conceituais distintas, enquanto soluções estruturalmente diferentes podem apresentar a mesma estratégia. Essa restrição impacta diretamente o desempenho do método de detecção baseado em similaridade.

Portanto, uma possibilidade para mitigar essa limitação, é o uso de representações mais expressivas, como por exemplo modelos do tipo CodeBERT³, que são capazes de capturar relações sintáticas e semânticas mais profundas entre programas. Isso poderá permitir análises mais precisas e maior sensibilidade para identificar estratégias conceituais que não dependem apenas da estrutura sintática.

Consequentemente, a abordagem com KNN e amplitude interquartil (AIQ), embora simples e alinhada ao caráter exploratório do estudo, também apresenta limitações. A técnica se mostra eficaz para destacar valores atípicos do ponto de vista estrutural, mas não necessariamente para identificar desvios conceituais mais sutis.

Outra limitação importante refere-se à análise qualitativa. Por razões práticas, apenas os códigos previamente rotulados como anomalias foram avaliados, o que significa que as demais soluções presentes nos *rankings* de similaridade não passaram por inspeção manual nem automatizada. Assim, não é possível afirmar que esses códigos atendem totalmente às especificações do enunciado. Essa restrição limita a generalização dos achados.

Além disso, as questões selecionadas para compor o estudo foram escolhidas com base na clareza dos enunciados, priorizando exercícios com exigências explícitas (por exemplo, implementação de algoritmos específicos, uso obrigatório de recursão ou funções). Embora essa escolha tenha sido adequada para fins exploratórios, ela pode ter

³Modelo pré-treinado para linguagens de programação. Disponível em: <https://github.com/microsoft/codebert>.

introduzido um viés de seleção, concentrando a análise em problemas com instruções objetivas e deixando de fora questões mais abertas ou com múltiplas estratégias de solução válidas.

Apesar das limitações identificadas, os resultados mostraram que a avaliação baseada em casos de teste, como é o caso dos juízes on-line, possui suas limitações. A análise qualitativa demonstrou que existem códigos que não seguem os requisitos da questão mas que são dados como corretos pelo sistema.

Além disso, percebeu-se que alguns dos desvios detectados correspondiam a padrões pedagógicos recorrentes, como substituição de algoritmos, simplificações excessivas ou o fato dos estudantes evitarem o uso de recursividade, sugerindo que abordagens automatizadas podem apoiar professores na identificação de lacunas conceituais.

6. Considerações Finais

Este trabalho apresentou um método baseado em técnicas de aprendizagem de máquina para identificar códigos que não cumpriram os requisitos do enunciado de uma questão de programação. Em juízes on-line, a correção do código ocorre com base em casos de teste pré-definidos, logo, não são considerados aspectos que podem ser relevantes para a aprendizagem de programação.

Questões que pedem a implementação de abordagens específicas no enunciado podem ser resolvidas utilizando outras estratégias, e é importante para o professor identificar quando isso acontece. Por isso, este trabalho explorou um conjunto limitado de exercícios e códigos que resolvem os exercícios, visando investigar casos em que os códigos aprovados pelo juiz on-line na verdade não cumpriram o que foi solicitado no enunciado.

Para tanto, foi aplicado o algoritmo *K-Nearest Neighbors* (KNN), para gerar *rankings* de similaridade entre os códigos e a amplitude interquartil (AIQ) para identificar anomalias dentro desses *rankings*. Nesse contexto, foram consideradas anomalias os códigos que possuíam menos similaridade com os demais. Com o método aplicado, foi realizada uma análise qualitativa sobre os códigos classificados como anômalos, para identificar se existiam, ou não, casos em que o código infringiu as regras do enunciado.

Os resultados indicaram que existem códigos classificados como corretos pelo juiz on-line, mas que na verdade não foram implementados conforme o enunciado solicitava. Por exemplo, em uma questão onde era necessário usar o *Selection Sort* para solucionar um problema de ordenação, foram identificados casos onde o estudante utilizou uma função pronta da linguagem.

Isso aponta para uma limitação da avaliação baseada em testes, pois, do ponto de vista pedagógico, é interessante para o professor identificar esses casos e fazer intervenções, caso achar necessário. Durante a análise também foram identificados casos em que os códigos não eram apropriados para resolver o problema, mas passaram nos casos de teste do sistema. O que aponta para outra limitação em juízes on-line, onde os casos de teste não alcançam todas as possibilidades de entrada e saída.

Como trabalhos futuros pretende-se utilizar representações de código que tenham maior capacidade de capturar aspectos semânticos dos códigos, além de explorar outras formas de identificar casos em que o código não cumpre os requisitos da questão.

7. Declaração sobre uso de Inteligência Artificial

Declara-se que foram utilizados recursos de Inteligência Artificial para realizar a revisão de alguns trechos do artigo. A concepção do estudo, o desenvolvimento da metodologia, a análise e interpretação dos dados, bem como a redação do conteúdo científico e das conclusões, foram realizados integralmente pelos autores. As ferramentas utilizadas não contribuíram para a geração de resultados, para as decisões metodológicas ou para as conclusões apresentadas.

Agradecimentos

O presente trabalho foi realizado com o apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (AUXPE-CAPES-PROEX) - Código de Financiamento 001. Adicionalmente, este trabalho foi parcialmente financiado pela Fundação de Amparo à Pesquisa do Estado do Amazonas - FAPEAM - por meio dos projetos PDPG-CAPES, POSGRAD 2025-2026 e POSGRAD 2026-2027.

Os autores expressam sua gratidão à Universidade do Estado do Amazonas (UEA) pelo apoio institucional. Agradecem também aos seus colegas do ThinkTED Lab pelas contribuições e discussões que enriqueceram este trabalho. Os autores reconhecem ainda o apoio do centro de pesquisa Nexus, que forneceu recursos materiais essenciais, incluindo espaço físico, computadores e infraestrutura relacionada.

Referências

- Aha, D. W., Kibler, D., e Albert, M. K. (1991). Instance-based learning algorithms. *Machine learning*, 6(1):37–66.
- Alves, F. P. e Jaques, P. (2014). Um ambiente virtual com feedback personalizado para apoio a disciplinas de programação. In *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*, volume 3, page 51.
- Barbosa, A. d. A., de Barros Costa, E., e Brito, P. H. (2023). Juízes online são suficientes ou precisamos de um var? In *Simpósio Brasileiro de Educação em Computação (EDUCOMP)*, pages 386–394. SBC.
- Burrows, S. e Tahaghoghi, S. M. (2007). Source code authorship attribution using n-grams. In *Proceedings of the twelfth Australasian document computing symposium, Melbourne, Australia, RMIT University*, pages 32–39. Citeseer.
- Carvalho, L. S., Oliveira, D. B., e Gadelha, B. F. (2016). Juiz online como ferramenta de apoio a uma metodologia de ensino híbrido em programação. In *Simpósio Brasileiro de Informática na Educação (SBIE)*, pages 140–149. SBC.
- de Oliveira, D. B., Lavareda Filho, R. M., Oliveira, E. H., Carvalho, L. S., Pereira, F. D., Colonna, J. G., e Menezes, A. (2021). Um método de detecção de plágio para sistemas juiz on-line baseado no comportamento dos alunos. In *Simpósio Brasileiro de Informática na Educação (SBIE)*, pages 836–848. SBC.
- Figueras, C., Farazouli, A., Cerratto Pargman, T., McGrath, C., e Rossitto, C. (2025). Promises and breakages of automated grading systems: a qualitative study in computer science education. *Education Inquiry*, pages 1–22.

- Glassman, E. L., Scott, J., Singh, R., Guo, P. J., e Miller, R. C. (2015). Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2):1–35.
- Ihantola, P., Ahoniemi, T., Karavirta, V., e Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research*, pages 86–93.
- Keuning, H., Jeuring, J., e Heeren, B. (2018). A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)*, 19(1):1–43.
- Liu, K., Han, Y., Zhang, J. M., Chen, Z., Sarro, F., Harman, M., Huang, G., e Ma, Y. (2023). Who judges the judge: An empirical study on online judge tests. In *Proceedings of the 32nd acm sigsoft international symposium on software testing and analysis*, pages 334–346.
- Melo, R., Souza, T., Pires, F., Oliveira, E., Carvalho, L., Pessoa, M., e Fernandes, D. (2026). Exploring the use of clustering algorithms and llms to identify programming strategies. *Revista Brasileira de Informática na Educação*, 34:59–82.
- Nijkamp, E., Hayashi, H., Xiong, C., Savarese, S., e Zhou, Y. (2023). Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*.
- Oliveira, A. M. d. et al. (2016). Um método de detecção de plágio em códigos-fonte para disciplinas iniciais de programação.
- Pimentel, E. P., da Silva, L. D. L., Takeuti, R. H., e Braga, J. C. (2025). Abordagem com llm para geração automatizada de feedback no ensino de programação de computadores. In *Simpósio Brasileiro de Informática na Educação (SBIE)*, pages 590–602. SBC.
- Raihan, N., Siddiq, M. L., Santos, J. C., e Zampieri, M. (2025). Large language models in computer science education: A systematic literature review. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*, pages 938–944.
- Sparck Jones, K. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21.
- Vinutha, H., Poornima, B., e Sagar, B. (2018). Detection of outliers using interquartile range technique from intrusion dataset. In *Information and decision sciences: Proceedings of the 6th international conference on ficta*, pages 511–518. Springer.
- Wasik, S., Antczak, M., Badura, J., Laskowski, A., e Sternal, T. (2018). A survey on online judge systems and their applications. *ACM Computing Surveys (CSUR)*, 51(1):1–34.
- Zhang, J., Li, D., Kolesar, J. C., Shi, H., e Piskac, R. (2022). Automated feedback generation for competition-level code. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*, pages 1–13.