



UNIVERSIDADE FEDERAL DO AMAZONAS - UFAM
INSTITUTO DE COMPUTAÇÃO- ICOMP
PROGRAMA PÓS-GRADUAÇÃO EM INFORMÁTICA - PPGI

Uso de códigos-solução para prever a dificuldade de exercícios de programação em sistemas juízes online

Francisco Ussosso Maurício

Manaus - AM

2026

Francisco Ussosso Maurício

Uso de códigos-solução para prever a dificuldade de exercícios de programação em sistemas juízes online

Dissertação submetida ao corpo docente do Programa de Pós-Graduação em Informática da Universidade Federal do Amazonas (PPGI-UFAM) como requisito para obtenção do título de Mestre em Informática.

Orientador(a)

Prof. Dr. David Braga Fernandes de Oliveira

Universidade Federal do Amazonas - UFAM

Instituto de Computação- IComp

Manaus - AM

2026

Ficha Catalográfica

Elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

M455u	Mauricio, Francisco Ussosso Uso de códigos-solução para prever a dificuldade de exercícios de programação em sistemas juízes online / Francisco Ussosso Mauricio. - 2026. 83 f. ; 31 cm. Orientador(a): David Braga Fernandes de Oliveira. Dissertação (mestrado) - Universidade Federal do Amazonas, Programa de Pós-Graduação em Informática, Manaus, 2026. 1. Código-solução. 2. Estimar dificuldade. 3. Juízes online. 4. Similaridade. I. Oliveira, David Braga Fernandes de. II. Universidade Federal do Amazonas. Programa de Pós-Graduação em Informática. III. Título
-------	---



Ministério da Educação
Universidade Federal do Amazonas
Coordenação do Programa de Pós-Graduação em Informática

FOLHA DE APROVAÇÃO

"USO DE CÓDIGOS-SOLUÇÃO PARA PREVER A DIFICULDADE DE EXERCÍCIOS DE PROGRAMAÇÃO EM SISTEMAS JUÍZES ONLINE"

FRANCISCO USSOSSO MAURÍCIO

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Dr. David Braga Fernandes de Oliveira - **Presidente**

Prof. Dr. Eduardo James Pereira Souto - **Membro Interno**

Prof.^a Dr.^a Marcela Savia Picanco Pessoa - **Membro Externo**

Manaus, 04 de setembro de 2026.



Documento assinado eletronicamente por **Eduardo James Pereira Souto, Professor do Magistério Superior**, em 08/09/2026, às 16:37, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **David Braga Fernandes de Oliveira, Professor do Magistério Superior**, em 16/09/2026, às 14:09, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Marcela Sávia Picanço Pessoa Bastos, Usuário Externo**, em 16/09/2026, às 14:49, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



A autenticidade deste documento pode ser conferida no site

[https://sei.ufam.edu.br/sei/controlador_externo.php?](https://sei.ufam.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0)

[acao=documento_conferir&id_orgao_acesso_externo=0](https://sei.ufam.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0), informando o código verificador **3351868** e o código CRC **335B45C2**.

Avenida General Rodrigo Octávio, 6200 - Bairro Coroados I Campus Universitário Senador Arthur Virgílio
Filho, Setor Norte - Telefone: (92) 3305-1181 / Ramal 1193
CEP 69080-900, Manaus/AM, coordenadorppgi@icomp.ufam.edu.br

Referência: Processo nº 23105.036799/2026-47

SEI nº 3351868

Titulação e nome do(a) orientador(a)

Orientador(a)

Departamento

Universidade

Titulação e nome do(a) membro da banca examinadora

Co-orientador(a), se houver

Departamento

Universidade

Titulação e nome do membro da banca examinadora

Departamento

Universidade

Titulação e nome do membro da banca examinadora

Departamento

Universidade

Manaus - AM, data de aprovação (por extenso).

Dedico esta dissertação aos meus pais Que sempre estiveram ao meu lado me ajudando. Amo vocês.

AGRADECIMENTOS

A Deus pelo dom da vida, pelos desafios que me permitiram viver oportunidades e pelas experiências gratificantes.

Aos meus pais Antônio Maurício e Avelina Cambamba pela motivação, força e por me ensinarem que através dos estudos eu posso conseguir o que é de mais valioso na vida.

Aos meus familiares, irmãos e primos, pela companhia, pela motivação e apoio.

Ao professor e a meu orientador, pelas dicas valiosas, pelos questionamentos que me motivaram a estar preparado e saber argumentar, pelas oportunidades, por me propor atividades e dar força quando eu não esperava conseguir. Agradeço ainda pela confiança para me conduzir durante este trabalho, pelos conselhos sobre a pesquisa, carreira e a vida.

Aos professores, Dra. Marcela e Dr. Eduardo Souto, por terem aceito o convite para participar da minha banca e pelas contribuições ao longo dos anos de mestrado.

Aos meus

Agradeço a todos os meus amigos que estiveram comigo durante o mestrado. Alguns eu já conhecia outros eu fui conhecendo ao longo do tempo. Durante este período fiz boas amizades e conheci pessoas maravilhosas tanto dentro quanto fora da universidade.

Agradeço ao programa de pós graduação do Instituto de computação (IComp) da Universidade Federal do Amazonas (UFAM) pela oportunidade e suporte que tem dado a todos os alunos. Me sinto grato por participar de um programa de pós-graduação de qualidade que a cada dia vem conquistando seu espaço perante as outras universidades.

Citação

Autor

Uso de códigos-solução para prever a dificuldade de exercícios de programação em sistemas juízes online

Autor: Francisco Ussosso Maurício

Orientador: Prof. Dr. David Braga Fernandes de Oliveira

Resumo

O ensino de programação representa um grande desafio no contexto educacional contemporâneo. A natureza técnica dos tópicos, aliada à exigência de um raciocínio lógico-estruturado, torna a aquisição de competências em programação especialmente complexa para estudantes em fase inicial. Face a essas dificuldades, inúmeras ferramentas tecnológicas foram desenvolvidas para apoiar o processo pedagógico, destacando-se os juízes online — plataformas que possibilitam submeter, corrigir e avaliar códigos automaticamente e em tempo real. Esses sistemas contam com repositórios que armazenam informações sobre diversos tópicos, constituindo bases de dados valiosas para docentes e discentes. Contudo, persiste uma lacuna relevante: a ausência de mecanismos eficazes para estimar automaticamente o nível de dificuldade de novos exercícios cadastrados. Essa limitação impacta o planejamento didático, podendo gerar desequilíbrios na progressão da aprendizagem e afetar a motivação dos estudantes. Neste cenário, este trabalho propõe uma metodologia para estimar automaticamente a dificuldade de novos exercícios em juízes online a partir de seus códigos-solução. A abordagem combina técnicas de comparação textual e semântica, utilizando a distância de Levenshtein e o modelo CodeBERT. A hipótese central assume que a dificuldade de um novo exercício pode ser inferida a partir do histórico de exercícios com soluções similares já existentes na base. Para validação experimental, utilizou-se um dataset com 958 exercícios e 42 mil submissões de uma disciplina introdutória de programação da Universidade Federal

do Amazonas (UFAM). Os resultados indicam coeficientes de correlação de Pearson entre 0,49 e 0,67 na presença de exercícios suficientemente similares, evidenciando a viabilidade da abordagem e destacando a importância da cobertura da base histórica para a eficiência das estimativas

Palavras-chave: Código-Solução, Estimar dificuldade, Similaridade, Juízes online.

Uso de códigos-solução para prever a dificuldade de exercícios de programação em sistemas juízes online

Autor: Francisco Ussosso Maurício

Orientador: Prof. Dr. David Braga Fernandes de Oliveira

Abstract

Teaching programming represents a major challenge in the contemporary educational context. The technical nature of the topics, combined with the requirement for structured logical reasoning, makes the acquisition of programming skills especially complex for novice students. In response to these difficulties, numerous technological tools have been developed to support the pedagogical process, notably online judges—platforms that enable the automated, real-time submission, grading, and evaluation of code. These systems rely on repositories that store information across various topics, constituting valuable databases for both instructors and students. However, a significant gap remains: the absence of effective mechanisms to automatically estimate the difficulty level of newly registered exercises. This limitation impacts instructional planning, potentially causing imbalances in learning progression and affecting student motivation. In this scenario, this work proposes a methodology to automatically estimate the difficulty of new exercises in online judges based on their reference solution codes. The approach combines textual and semantic comparison techniques, utilizing Levenshtein distance and the CodeBERT model. The central hypothesis posits that the difficulty of a new exercise can be inferred from the historical record of existing exercises with similar solutions in the database. For experimental validation, a dataset comprising 958 exercises and 42,000 submissions from an introductory programming course at the Federal University of Amazonas (UFAM) was used. The results indicate Pearson correlation coefficients

between 0.49 and 0.67 in the presence of sufficiently similar exercises, demonstrating the feasibility of the approach and highlighting the importance of historical database coverage for estimation efficiency.

Keywords: Solution Code, Difficulty Estimation, Online Judges.

LISTA DE ILUSTRAÇÕES

Figura 1 – Tela de bem vindo do Codebanch	28
Figura 2 – Integrated Development Environment (IDE)	29
Figura 3 – Representação esquemática do método	54
Figura 4 – Código Original	56
Figura 5 – Apresentação do código em Python após o processo da normalização simples	57
Figura 6 – Apresentação do código em Python após o processo da normalização avançada	59

LISTA DE TABELAS

Tabela 1 – Comparação de estudos sobre complexidade e dificuldade	46
Tabela 2 – Comparação de estudos sobre previsão de dificuldade	48
Tabela 3 – Classificação e percepção de dificuldade	50
Tabela 4 – Comparação de desempenho do modelo CodeBERT quando exercícios do mesmo bloco não são mantidos	70
Tabela 5 – Comparação de desempenho do modelo CodeBERT quando exercícios do mesmo bloco não mantidos	70
Tabela 6 – Comparação de desempenho do modelo Levenshtein quando exercícios do mesmo bloco não são mantidos	71
Tabela 7 – Comparação de desempenho do modelo Levenshtein quando exercícios do mesmo bloco não mantidos	72

LISTA DE ABREVIATURAS E SIGLAS

LISTA DE SÍMBOLOS

SUMÁRIO

1	INTRODUÇÃO	17
1.1	Justificativa	20
1.2	Objetivos	21
1.2.1	Objetivo Geral	21
1.2.2	Objetivos Específicos	21
1.3	Questões de Pesquisa	22
1.4	Metodologia	22
1.5	Hipótese	23
1.6	Organização do Trabalho	23
2	FUNDAMENTOS TEÓRICOS	25
2.1	Sistemas de Avaliação Automática de Códigos	25
2.2	Codebench	27
2.3	Dificuldade de Exercícios de Programação	31
2.4	Métricas de Similaridade	32
2.5	Métricas de Similaridade de Código-Fonte	33
2.6	Distância de Levenshtein	34
2.6.1	Codebert	36
2.7	Avaliação e Validação do Método	39
2.7.1	Coeficientes de Correlação (Validação da Relação)	39
2.7.1.1	Coeficiente de Pearson	39
2.7.1.2	Significância Estatística (p-value)	40
2.7.2	Avaliação do Desempenho	41
2.7.2.1	Erro Médio Absoluto (MAE)	41
2.7.2.2	Erro Quadrático Médio (RMSE)	41

2.8	Considerações Finais	42
3	TRABALHOS RELACIONADOS	44
3.1	Complexidade e Dificuldade	44
3.2	Previsão de Dificuldade	47
3.3	Classificação de dificuldade dos exercícios	48
3.4	Considerações Finais	51
4	MÉTODO	53
4.1	Cadastro do Exercício	55
4.2	Normalização dos códigos	55
4.2.1	Normalização simples	56
4.2.2	Normalização avançada	57
4.3	Cálculo de Similaridade	59
4.4	Estimativa da dificuldade de novos exercícios	60
4.4.1	Métricas de Dificuldade	60
4.4.2	Avaliação do Desempenho Preditivo	62
5	EXPERIMENTOS	63
5.1	Contexto Educacional	63
5.2	Dataset	64
5.3	Ambiente Computacional	64
5.4	Protocolo Experimental	65
5.4.1	Pré-processamento e Normalização	65
5.4.2	Conjuntos de Códigos Comparação	66
5.4.3	Uso de Blocos de Exercícios	67
5.4.4	Métricas de Dificuldade	68
5.5	Resultados	69
5.5.1	Experimentos com o CodeBERT	69
5.5.2	Experimentos com o Levenshtein	70
5.5.3	Discussão dos resultados	72
5.5.3.1	CodeBERT versus Levenshtein	72

5.5.3.2	Normalização avançada versus simples	73
5.5.3.3	Impacto do conjunto de comparação	73
5.5.3.4	Cobertura da base como fator determinante	74
6	CONSIDERAÇÕES FINAIS	75
6.1	Limitações	76
6.2	Trabalhos Futuros	77
6.3	Agradecimentos	78
	Referências	79

1

INTRODUÇÃO

O aprendizado em qualquer área do conhecimento exige empenho, dedicação e esforço contínuo. No campo da programação, esses fatores são igualmente essenciais, exigindo compromisso e prática constante para a aquisição das habilidades técnicas necessárias para criar soluções algorítmicas e implementar softwares que resolvam problemas reais ([FRANCISCO; JÚNIOR; AMBRÓSIO, 2016](#)).

De acordo ([SILVA; MENEZES; JUNIOR, 2023](#)), aprender a programar desempenha um papel crucial no desenvolvimento cognitivo dos alunos, pois exige esforço contínuo para compreender e analisar problemas, o que, por sua vez, resulta na tradução dessas soluções em códigos executáveis. Nesse processo, a prática constante da codificação e a resolução frequente de exercícios têm se mostrado essenciais para o sucesso no aprendizado de programação, pois proporcionam a aplicação prática dos conceitos e a melhoria gradual das habilidades técnicas ([PRIOR, 2003](#)). Também de acordo com ([FRANCISCO; AMBROSIO, 2015](#)), a resolução contínua de exercícios ao longo do processo de ensino é considerado fundamental, proporcionando feedback constante e contribuindo para o desenvolvimento de habilidades práticas pelos estudantes .

É notório que muitos alunos enfrentam diversas dificuldades no processo de aprendizado, o que se reflete em um baixo índice de aproveitamento em disciplinas cuja base é a programação ([MOREIRA; FAVERO, 2009](#)). Esse cenário pode ser explicado por diferentes fatores, como a falta de interesse por parte dos estudantes, o que se relaciona tanto à dificuldade em compreender a importância da disciplina, quanto à ausência de identificação com o conteúdo apresentado, e bem como a falta de maturidade por

parte dos alunos para perceber a devida importância do conteúdo para a sua formação (GALVÃO; FERNANDES; GADELHA, 2016).

As causas do insucesso por parte dos alunos têm sido um tema recorrente de estudo por pesquisadores ao longo do tempo (GOMES et al., 2008). O estudo de (SOUZA; BATISTA; BARBOSA, 2016) aponta algumas causas relevantes para o alto número de reprovações, tais como: (i) dificuldade dos alunos em compreender conceitos fundamentais da programação; (ii) dificuldade em aplicar esses conceitos na construção de programas; (iii) desinteresse ou falta de motivação dos estudantes em realizar atividades relacionadas à programação.

Na Universidade Federal do Amazonas (UFAM), a título de exemplo, a disciplina de Introdução à Programação de Computadores (IPC) utiliza sistemas de juízes online e é ofertada em mais de 14 cursos, incluindo engenharias e ciências exatas. Em oito desses cursos, IPC é ministrada no primeiro período. Contudo, os altos índices de abandono e reprovação representam um desafio considerável para os professores da disciplina (CARVALHO et al., 2016).

No cenário atual, a implementação de ferramentas digitais tem sido uma estratégia fundamental para melhorar o aprendizado e minimizar as dificuldades enfrentadas por alunos e professores (ALVES; JAQUES, 2014). Nesse contexto, diversas soluções tecnológicas têm sido desenvolvidas e propostas com o objetivo de promover melhorias no processo de ensino (FRANCISCO; JÚNIOR; AMBRÓSIO, 2016). Entre as soluções tecnológicas que vêm ganhando destaque, os juízes online têm se mostrado especialmente eficazes para a correção automática de exercícios, oferecendo feedback instantâneo aos estudantes e facilitando o gerenciamento de listas de atividades pelos professores (FERNANDES et al., 2023).

A possibilidade oferecida pelos sistemas de juízes online de criar diferentes tipos de atividades os torna uma ferramenta valiosa para avaliações em disciplinas acadêmicas. Em uma avaliação que utilize um juiz online, as questões são selecionadas a partir de um banco previamente cadastrado, devendo ser resolvidas dentro de um período de tempo definido. Para prevenir práticas como plágio, colas ou conversas paralelas, as questões podem ser sorteadas de forma aleatória, garantindo maior segurança e

integridade no processo avaliativo.

Os juízes online dispõem de amplos repositórios de questões que abordam diversos tópicos fundamentais da programação, como estruturas de controle, vetores, matrizes, recursão, entre outros. Esses repositórios são ferramentas úteis para os professores no processo de composição de listas de exercícios. Para auxiliar nessa tarefa, é comum que esses sistemas compartilhem uma série de informações sobre cada exercício, de forma a ajudar os professores a montar suas avaliações e listas de exercícios. Dentre essas informações, destacamos: o nível de dificuldade do exercício, tempo médio de resolução, taxa de acerto, tópicos abordados, entre outras.

Essas informações são fundamentais para os professores, pois permitem a elaboração de listas de exercícios com níveis de dificuldade ajustados às necessidades específicas de suas turmas. Ademais, algumas abordagens propostas na literatura sugerem o uso de técnicas de recomendação automática de questões baseadas nessas informações, como descrito em (JÚNIOR et al., 2020) e (LIMA et al., 2020).

No entanto, ao criar um novo exercício, o professor não dispõe de informações imediatas sobre seu nível de dificuldade, já que o exercício ainda não foi solucionado por nenhum aluno. Essa ausência de dados dificulta a inclusão de exercícios novos em listas e avaliações, pois compromete a capacidade do professor de identificar se seus exercícios estão adequados ou não ao nível de suas turmas.

A dificuldade em avaliar o nível de dificuldade de novas questões em juízes online, devido à ausência de um histórico de submissões, representa uma limitação significativa no pleno aproveitamento dessas ferramentas pedagógicas. Essa incerteza pode levar à escolha inadequada de exercícios, comprometendo o desempenho e a motivação dos alunos. Diante dessa lacuna, torna-se crucial desenvolver mecanismos ou abordagens que permitam aos professores estimar de forma confiável a dificuldade de novas questões.

Neste cenário a presente pesquisa direciona tem como objetivo estimar a dificuldade de exercícios recém cadastrados em sistemas de juízes online. Para alcançar esse objetivo, pretende-se investigar e desenvolver um método inovador que utilize o código-solução fornecido como base para gerar informações preditivas sobre o nível de

dificuldade dessas novas questões.

1.1 Justificativa

Em (SANTOS et al., 2019) foi proposto um método de classificação da dificuldade dos exercícios com base em métricas de inteligibilidade de texto extraídas dos enunciados. Nesta abordagem, foram consideradas quatro classes de dificuldade (muito fácil, fácil, médio e difícil), alcançando uma acurácia de 75% utilizando o método Random Forest para classificar os exercícios em uma dessas classes predefinidas.

No entanto, os autores identificaram que a classificação da dificuldade com base apenas na inteligibilidade dos enunciados apresenta limitações, pois a interpretação do texto representa apenas uma etapa no processo de resolução da questão. Além disso, a utilização exclusiva de métricas sintáticas mostra-se restritiva, especialmente quando comparada ao potencial de métricas semânticas, que poderiam fornecer informações complementares se utilizadas em conjunto com as métricas sintáticas (SCARTON; ALUÍSIO, 2010).

Além dos enunciados, durante a criação do exercício os professores podem informar um código-solução, que representa um possível algoritmo para resolver o problema proposto no enunciado do exercício. Esse código desempenha um papel fundamental, auxiliando o professor na definição de casos de teste para avaliar a corretude dos códigos submetidos pelos alunos. Além disso, o código-solução pode ser utilizado como objeto de estudo para a análise de métricas de código e na avaliação das estratégias de solução adotadas pelos estudantes ao resolverem o exercício.

Neste contexto, o presente trabalho propõe uma abordagem alternativa para estimar exercícios ainda não resolvidos de acordo com seu nível de dificuldade, utilizando os códigos-solução como base para essa análise. Diferente da metodologia apresentada em (LIMA et al., 2020), que empregou métricas de código diretamente, a proposta deste estudo envolve a busca por exercícios cujos códigos desenvolvidos pelos alunos sejam semelhantes ao código-solução cadastrado pelo professor. Com base nessa comparação, será possível estimar o nível de dificuldade do novo exercício com referência nos níveis

de dificuldade dos exercícios encontrados com padrões similares.

A premissa dessa abordagem fundamenta-se na ideia de que exercícios cujas soluções apresentem alta similaridade tendem a possuir níveis de dificuldade semelhantes. Assim, esta técnica busca fornecer uma avaliação indireta da complexidade do problema, oferecendo uma alternativa para análise da dificuldade de exercícios no ambiente dos juízes online.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo principal deste trabalho é propor um método que utiliza os códigos-solução e técnicas de similaridade de códigos para identificar o nível de dificuldade de exercícios de programação que ainda não foram resolvidos por nenhum aluno em sistemas de juízes online.

1.2.2 Objetivos Específicos

Com base no objetivo geral, os seguintes objetivos específicos foram definidos:

- Criar um conjunto de dados (dataset) contendo exercícios de programação com os níveis de taxa de acerto pré computado.
- Propor estratégias de normalização de código adequado ao problema de calculo de estimativa de dificuldade.
- Comparar a eficácia de métricas de similaridade textuais e semânticas para a identificação de proximidade estrutural e lógica entre os códigos-fonte.
- Avaliar a correlação linear entre a similaridade de códigos e os níveis reais de dificuldade dos exercícios.
- Mensurar o desempenho e o erro absoluto da abordagem proposta por meio das métricas de erro contínuo Mean Absolute Error (MAE) e Root Mean Squared Error

(RMSE).

1.3 Questões de Pesquisa

Neste trabalho, consideramos as seguintes questões de pesquisa:

- QP1: É viável estimar a dificuldade de novos exercícios de programação em juízes online utilizando apenas a similaridade de seus códigos-solução com uma base histórica?
- QP2: Qual é o nível de precisão e confiabilidade das métricas de similaridade textual e semântica na predição da taxa de acerto desses exercícios?

1.4 Metodologia

A seguir são apresentadas as etapas metodológicas adotadas neste trabalho, as quais compreendem um conjunto de procedimentos característicos de projetos com esta natureza.

a) Revisão da literatura: levantamento e análise de trabalhos correlatos nas áreas de mineração de repositórios educacionais, estimativa de dificuldade de tarefas e processamento ou similaridade de código-fonte.

b) Seleção das métricas e modelos de similaridade: definição das abordagens de comparação textual (distância de Levenshtein) e representação semântica densa baseada em aprendizado profundo (CodeBERT) para mensurar a proximidade entre códigos-fonte.

c) Coleta e consolidação do dataset: extração de dados históricos de submissões e códigos-solução de turmas de Introdução à Programação de Computadores (IPC) da Universidade Federal do Amazonas (UFAM), armazenados no juiz online Codebench,

d) Pré-processamento e normalização dos códigos: aplicação de rotinas de limpeza sintática e padronização estrutural nos códigos-fonte para mitigar ruídos nas comparações.

e) Seleção e aplicação das métricas de similaridade: cálculo de proximidade textual via distância de Levenshtein e semântica via representações densas do CodeBERT.

e) Análise de correlação estatística: investigação da relação entre os escores de similaridade calculados e as taxas reais de acerto dos exercícios, aplicando o coeficiente de correlação de Pearson para verificar a consistência da hipótese.

1.5 Hipótese

A hipótese central deste trabalho sustenta que a similaridade estrutural e semântica entre códigos-solução atua como um preditor confiável do nível de dificuldade de novos exercícios em sistemas de juízes online.

Essa premissa fundamenta-se no princípio de que exercícios com padrões lógicos, estruturas algorítmicas e construções sintáticas semelhantes demandam níveis comparáveis de esforço cognitivo por parte dos estudantes.

Assim, ao comparar o código-solução de um exercício inédito com códigos de problemas previamente resolvidos cujos níveis de dificuldade já são conhecidos pelo histórico de desempenho dos alunos, torna-se possível inferir, de maneira automatizada, a taxa de acerto e a complexidade do novo problema.

1.6 Organização do Trabalho

O presente trabalho está estruturado em seis capítulos, organizados da seguinte forma:

*** **Capítulo 1 – Introdução:**** contextualiza a pesquisa, define o problema investigado e apresenta a justificativa, a hipótese central, os objetivos geral e específicos, bem como as questões de pesquisa que norteiam o estudo.

*** **Capítulo 2 – Fundamentação Teórica:**** aborda os conceitos essenciais e as teorias que sustentam a pesquisa, como juízes **online**, métricas de similaridade textual e representações semânticas de código-fonte.

*** **Capítulo 3 – Trabalhos Relacionados:**** apresenta uma revisão da literatura sobre abordagens de predição de dificuldade em ambientes educacionais de programação,

identificando lacunas e posicionando a contribuição deste estudo.

* **Capítulo 4 – Método:** descreve detalhadamente o método proposto para estimar a dificuldade de novos exercícios por meio da similaridade de códigos-solução, especificando as rotinas de normalização, os modelos de similaridade adotados e os algoritmos de inferência.

* **Capítulo 5 – Resultados e Discussão:** expõe a avaliação experimental realizada sobre o *dataset* da UFAM, analisando as correlações estatísticas, os erros de predição (MAE e RMSE) e a discussão crítica dos achados à luz da hipótese formulada.

* **Capítulo 6 – Conclusão:** sintetiza as principais contribuições da dissertação, discute as limitações identificadas e propõe direções para trabalhos futuros.

2

FUNDAMENTOS TEÓRICOS

Este capítulo apresenta a fundamentação teórica necessária para o entendimento deste trabalho. Serão abordados conceitos como sistemas de juízes online, seu funcionamento no contexto educacional, além de métricas e modelos usados para a construção da pesquisa.

2.1 Sistemas de Avaliação Automática de Códigos

Sistemas juízes online foram inicialmente criados como ferramentas para competições de programação, facilitando a avaliação automática de soluções dos exercícios ([ALVES; JAQUES, 2014](#)). No entanto, sua aplicação ultrapassou o contexto competitivo, despertando interesse na área pedagógica, especialmente no ensino de programação. Essa transição ampliou as possibilidades para educadores, permitindo o uso desses sistemas como recursos didáticos para ensinar lógica, algoritmos e boas práticas de desenvolvimento de software de maneira prática, interativa e contextualizada ([FRANCISCO; AMBROSIO, 2015](#)).

Os juízes online são ferramentas computacionais projetadas para facilitar a escrita, avaliação e correção automática de códigos de programação. Elas possibilitam tanto o desenvolvimento de soluções por programadores quanto a validação e análise desses códigos, oferecendo maior precisão e eficiência no processo de revisão e avaliação ([SILVA et al., 2022](#)). ; Os sistemas de juízes online apresentam um grande potencial pedagógico para transformar o ensino e a aprendizagem em programação ([VIER; GLUZ;](#)

[JAQUES, 2015](#)). Para garantir sua eficácia, é crucial selecionar uma ferramenta que esteja alinhada ao estilo de ensino do professor e às necessidades pedagógicas dos alunos, otimizando sua aplicabilidade e impacto no ambiente educacional .

Essas ferramentas disponibilizam uma variedade de recursos educacionais, como tutoriais, exercícios interativos e materiais de apoio, proporcionando aos alunos a oportunidade de aprender no próprio ritmo, de maneira personalizada e envolvente.

Assim que uma submissão para um exercício é realizada, o programa submetido é avaliado a partir de um conjunto de casos de teste. Para isso, o sistema juiz online verifica se a saída gerada pelo código do aluno é igual ou não à saída esperada para cada caso de teste.

De acordo ([FRANCISCO et al., 2018](#)), alguns requisitos funcionais essenciais para o funcionamento de sistemas juízes online educacionais incluem: o fornecimento de feedback imediato e detalhado aos alunos, a integração com os conteúdos e plataformas dos cursos, a análise do desempenho geral dos alunos por meio de métricas e relatórios, e a oferta de diferentes tipos de atividades, como desafios de programação e exercícios variados. O feedback é um requisito essencial em sistemas de correção automática e desempenha um papel central no processo de ensino e aprendizagem. Ele oferece orientação imediata, permitindo que os alunos identifiquem erros, compreendam suas falhas e aprimorem seus conhecimentos de maneira contínua e direcionada. Já entre os requisitos não funcionais, destacam-se a usabilidade, que garante uma interface intuitiva e amigável; a escalabilidade, que assegura o suporte a um grande número de usuários simultâneos; e a disponibilidade, que proporciona acesso contínuo ao sistema, minimizando interrupções.

A transparência desses sistemas é amplamente valorizada, pois elimina a interferência humana na correção dos exercícios, garantindo imparcialidade e confiabilidade no processo avaliativo.

Uma das características mais apreciadas dos sistemas juízes online é a possibilidade de o professor monitorar detalhadamente o desempenho dos alunos, permitindo o acompanhamento do progresso individual e a identificação precisa de áreas que necessitam de maior atenção e suporte.

Na literatura científica, há uma diversidade de sistemas de juízes online disponíveis. Como exemplo, podemos citar o BOCA (BOCA Online Contest Administrator) (CAMPOS; FERREIRA, 2004), que é um sistema de correção automática amplamente utilizado em competições de programação organizadas pela Sociedade Brasileira de Computação (SBC). O programa permite a criação de competições, além de possibilitar o envio de dúvidas aos juízes das provas, que podem respondê-las por meio de uma interface específica. O BOCA também oferece a funcionalidade de submissão de códigos em linguagens como C++ e Java. Por ser uma plataforma online, ele apenas informa se a solução está correta ou não, sem indicar detalhadamente onde o erro se encontra.

Outra ferramenta relevante é o The Huxley, um sistema web que permite aos alunos submeterem códigos em diversas linguagens de programação. Para cada exercício enviado, o sistema fornece feedback instantâneo, avaliando tanto a análise sintática quanto os testes de aceitação do código submetido (PAES et al., 2013).

Outra ferramenta amplamente conhecida no Brasil é o Beecrowd Online Judge¹ (antigo URI Online Judge), reconhecido como um dos melhores portais de programação do mundo. O sistema se destaca por sua interface amigável e por oferecer recursos avançados, que são raros de encontrar em outras plataformas similares. O portal conta com mais de 500 mil problemas cadastrados e mais de 15 mil usuários ativos, incluindo professores de diversas partes do mundo.

2.2 Codebench

Na Universidade Federal do Amazonas(UFAM), o CodeBench² é o sistema online de correção automática de código-fonte usado, desenvolvido pelo Instituto de Computação da UFAM, com o objetivo de oferecer suporte a professores e alunos em disciplinas de programação.

Por meio do CodeBench, os professores têm a possibilidade de disponibilizar exercícios de programação para seus estudantes, que podem desenvolver e submeter suas soluções diretamente por meio da interface intuitiva do sistema. Essa ferramenta

¹ <<https://beecrowd.com/>>

² <<https://codebench.icomp.ufam.edu.br/>>



Figura 1 – Tela de bem vindo do Codebench

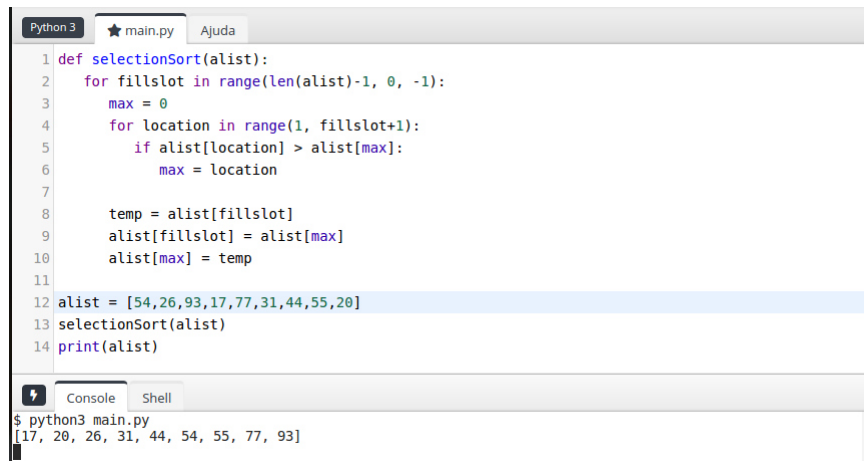
visa facilitar o processo de ensino, permitindo uma avaliação rápida e automatizada das respostas, promovendo, assim, uma experiência prática e eficiente no aprendizado da programação.

O CodeBench tem por objetivos: i) prover ao discente de disciplinas de programação um conjunto de ferramentas pedagógicas capazes de estimular e facilitar seu aprendizado; ii) prover o docente com informações úteis sobre a caminhada do aluno nas disciplinas de programação; iii) dispor um conjunto de ferramentas capazes de simplificar o trabalho docente; e iv) fomentar e apoiar professores no desenvolvimento e/ou implementação de práticas de ensino mais modernas e criativas.

Uma vez que um aluno submete uma solução para um dado exercício, o sistema informa instantaneamente ao aluno se sua solução está correta ou errada. Além disso, o CodeBench permite a troca de mensagens entre alunos e professores de uma dada turma, bem como o compartilhamento de recursos didáticos por parte dos professores.

O CodeBench possui um Ambiente de Desenvolvimento Integrado, ou Integrated Development Environment (IDE), que é usado pelos alunos para desenvolver as soluções dos exercícios propostos. Atualmente, esse ambiente suporta as principais funcionalidades de um IDE típico, tais como: Autocompletion, Autosave, Syntax Highlighting, busca e substituição de strings.

O CodeBench disponibiliza recursos de:



```
Python 3  ★ main.py  Ajuda
1 def selectionSort(alist):
2     for fillslot in range(len(alist)-1, 0, -1):
3         max = 0
4         for location in range(1, fillslot+1):
5             if alist[location] > alist[max]:
6                 max = location
7
8         temp = alist[fillslot]
9         alist[fillslot] = alist[max]
10        alist[max] = temp
11
12 alist = [54,26,93,17,77,31,44,55,20]
13 selectionSort(alist)
14 print(alist)

Console  Shell
$ python3 main.py
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Figura 2 – Integrated Development Environment (IDE)

- Banco de exercícios: o CodeBench possui mais de 5000 exercícios já cadastrados, que podem ser usados pelos professores para avaliar os trabalhos de suas próprias turmas.
- Materiais Didáticos: os professores compartilham os materiais para que possam servir de auxílio para os alunos no seu processo de aprendizagem.
- Troca de mensagens: cada usuário do CodeBench possui uma caixa de e-mails que pode ser usada para troca de mensagens entre os demais usuários do sistema.
- Detecção de plágio: o CodeBench possui um detector de plágio que identifica códigos semelhantes submetidos pelos alunos para um mesmo exercício.
- Tecnologia Docker: os códigos dos alunos são executados dentro de ambientes seguros e virtualizados chamados de docker containers.

Atualmente, o CodeBench suporta várias linguagens de programação como: C, C++, Java, Python, Haskell, Lua, Prolog, Script Bash e Assembly (ARM), Dart. Além dessas linguagens, o ambiente também suporta a linguagem SQL, para exercícios envolvendo consultas a bancos de dados.

O CodeBench utiliza dois passos principais para avaliar a corretude do código submetido por um aluno, conforme descrito por ([CARVALHO et al., 2016](#)):

- Análise sintática do código: onde o sistema verifica se o código submetido possui algum tipo de erro sintático de acordo com a gramática da linguagem definida pelo professor. Se porventura for encontrado qualquer erro no código submetido, o aluno recebe uma informação sobre o tipo de erro e onde ele se encontra.
- Análise lógica do código: Verificação do sistema se o código submetido pelo estudante atende aos requisitos do enunciado do exercício.

De acordo com (CARVALHO et al., 2016), a análise lógica do código é realizada por meio de casos de testes, que são cadastrados pelos professores no momento da criação da questão. Cada caso de teste possui dois valores: um valor de entrada, correspondente ao dado fornecido pelo aluno no programa, e um valor de saída, que representa a resposta correta esperada para essa entrada.

O registro no CodeBench pode ser realizado de forma individual pelo aluno, sem a necessidade de autorização prévia por parte de um administrador. Após o cadastro, o aluno tem a opção de se matricular em uma turma, possibilitando a interação com os exercícios e o acesso aos recursos disponibilizados pelo professor da turma.

Dentro do CodeBench assim como de outros Juizes Online (JO), é possível encontramos uma gama de exercícios pré-cadastrados, os quais podem ser resolvidos independentemente da linguagem de programação utilizada. Para cada exercício, existe um conjunto de casos de teste que o sistema utiliza para determinar se as soluções submetidas pelos alunos são corretas ou não.

Essas ferramentas ajudam nas correções automáticas e respostas rápidas e podem desempenhar um papel essencial no estímulo ao interesse dos alunos. Além disso, tais ferramentas contribuem para a redução do tempo dedicado pelos professores a tarefas repetitivas, como a correção de exercícios, permitindo que possam direcionar seus esforços para atividades mais estratégicas, como a orientação personalizada e o aprofundamento no conteúdo (ALVES et al., 2019).

2.3 Dificuldade de Exercícios de Programação

Saber resolver problemas e ter um domínio básico de matemática, por exemplo, certamente podem ajudar no aprendizado de programação. Programar não é uma tarefa fácil, como já foi mencionado anteriormente, e é amplamente reconhecido que muitos estudantes enfrentam dificuldades no aprendizado, resultando em altas taxas de evasão.

Parte dessa dificuldade surge, precisamente, do fato de que a maioria dos alunos enfrenta desafios em compreender e aplicar os conceitos abordados, especialmente em problemas que exigem a integração de múltiplas estruturas lógicas, como loops aninhados, condicionais complexas ou recursão.

No entanto, é importante considerar outros fatores que também influenciam esse processo e nos ajudam a entender melhor as dificuldades enfrentadas pelos alunos. Segundo [Kirschner et al. \(2018\)](#) os exercícios que integram múltiplos conceitos simultaneamente podem exceder a capacidade cognitiva do aluno, gerando frustração e erros frequentes. Isso explica por que problemas aparentemente simples, quando combinados com sintaxe complexa, são percebidos como desproporcionalmente difíceis por iniciantes.

O ambiente de aprendizagem também influencia significativamente a percepção da dificuldade. Uma pesquisa feita [Ihantola, Sorva e Vihavainen \(2014\)](#) demonstram que ambientes com recursos de depuração visual e exemplos interativos reduzem em até 30% o tempo de resolução de exercícios, indicando que a qualidade do suporte técnico chegam sendo um fator determinante.

Nesse contexto, classificar a dificuldade de exercícios ainda não resolvidos torna-se um desafio. Sistemas de Julgamento Online (JOs) dependem de métricas históricas como taxa de acerto, número médio de submissões, tamanho médio dos códigos corretos e tempo médio de codificação que, além de só estarem disponíveis após a resolução dos exercícios pelos alunos, serão discutidas com maior detalhes nas seções seguintes. Para contornar essa limitação, métodos baseados em similaridade entre códigos-solução têm sido propostos, oferecendo suporte na estimativa da dificuldade e na resolução dessas questões.

2.4 Métricas de Similaridade

Uma métrica pode ser definida como uma medida quantificável utilizada para avaliar ou descrever uma característica específica de uma entidade. Nesse sentido, as métricas podem estar relacionadas à avaliação da similaridade, representando os métodos empregados para obter uma medida quantitativa ou qualitativa no que diz respeito à semelhança entre objetos ou conceitos.

A similaridade é uma medida que quantifica o grau em que dois ou mais objetos compartilham características comuns, ou seja, o quanto esses objetos são semelhantes entre si (JULIO, 2005). Assim, pode-se definir a similaridade como o grau de semelhança existente entre dois ou mais objetos, conforme destacado por (ALI; AYDAM; RASHED, 2020).

É possível utilizar diferentes métricas para calcular a similaridade entre objetos, o que pode resultar em variações nos resultados, uma vez que cada métrica possui fórmulas e abordagens distintas. Ao selecionar uma métrica de similaridade específica para uma aplicação determinada, os pesquisadores têm a oportunidade de obter resultados mais precisos e significativos, adequando a análise às particularidades do contexto em questão.

A título de exemplo, a métrica de similaridade da distância euclidiana, que se destaca por sua simplicidade e eficiência na determinação de similaridades entre objetos. Devido a essas características, muitos aplicativos a utilizam para tarefas como reconhecimento de padrões, agrupamentos e classificação de objetos em diversas áreas, oferecendo resultados rápidos e confiáveis.

Além da distância euclidiana, devemos destacar a similaridade cosseno, que é outra métrica amplamente utilizada para medir a similaridade entre dois vetores em um espaço vetorial. Ela é calculada com base no cosseno do ângulo entre os vetores e é especialmente eficiente em análises de texto. A similaridade cosseno se destaca por ser uma abordagem mais robusta e sofisticada para avaliar a proximidade e semelhança entre elementos, tornando-a uma ferramenta valiosa em aplicações como mineração de texto, processamento de linguagem natural e recomendação de sistemas.

Embora reconheçamos a diversidade de métricas existentes, optamos por utilizar

métricas de similaridade de código-fonte, que analisam padrões sintáticos, lógicos e semânticos em trechos de código, além da Distância de Levenshtein para detetar equivalências textuais. Nas próximas seções, apresentamos em detalhes cada uma das métricas escolhidas.

2.5 Métricas de Similaridade de Código-Fonte

O cálculo de similaridade entre códigos-fonte é uma abordagem utilizada para mensurar o grau de semelhança entre trechos de código, considerando características textuais, sintáticas e semânticas. Essa técnica oferece subsídios relevantes para análises de similaridade, detecção de padrões e compreensão de práticas no desenvolvimento de software ([ZAKERI-NASRABADI et al., 2023](#)).

A similaridade de código-fonte consiste em comparar dois trechos de código para determinar o grau de semelhança entre eles com base em características e propriedades compartilhadas.

Essa análise é realizada por meio de técnicas especializadas, que atribuem uma pontuação numérica real a cada par de códigos comparados. Quanto maior a pontuação, maior é a semelhança entre os dois trechos analisados.

Geralmente, a similaridade de código-fonte é utilizada em aplicações para detecção de plágio, identificação de softwares maliciosos ou até mesmo na análise de casos de clonagem de códigos.

De acordo com ([ZAKERI-NASRABADI et al., 2023](#)), em uma revisão sistemática da literatura (RSL), observou-se que os métodos híbridos, que combinam duas ou mais abordagens, são os mais utilizados para medir a similaridade de código-fonte. Os estudos sugerem que a combinação de métodos híbridos oferece resultados mais eficazes na avaliação da similaridade entre trechos de código.

Para comparar os códigos-solução e capturar tanto aspectos sintáticos quanto semânticos, este trabalho adota duas abordagens complementares: a Distância de Levenshtein (baseada em edição de texto) e o CodeBERT associado à Similaridade do Cosseno (baseado em representações vetoriais densas).

2.6 Distância de Levenshtein

A Distância de Levenshtein, criada pelo russo e matemático Vladimir Levenshtein em 1965, é uma métrica de distância de edição utilizada para comparar duas strings (BAC-KURS; INDYK, 2015). A distância entre duas sequências de texto (a,b) é definida como o número mínimo de operações necessárias para transformar uma sequência na outra. Essas operações incluem inserções, exclusões ou substituições de símbolos. Quando realizadas no menor número possível elas determinam a chamada distância de edição, fornecendo uma medida quantitativa da similaridade entre as duas sequências.

- **Inserção:** Trata-se da operação que consiste em adicionar um caractere a uma string, podendo ocorrer em qualquer posição ou seja no início, no meio ou no final da sequência.
- **Exclusão:** Refere-se à operação de remover um caractere de uma string, reduzindo seu comprimento em uma unidade. A exclusão pode ocorrer em qualquer posição dentro da sequência.
- **Substituição:** É a operação que consiste em substituir um caractere de uma string por outro distinto. Essa operação mantém o comprimento da string inalterado, mas modifica seu conteúdo em uma determinada posição.

A fórmula usada para calcular a Distância de Levenshtein é dada por:

$$D(i, j) = \min \begin{cases} D(i-1, j) + 1 \\ D(i, j-1) + 1 \\ D(i-1, j-1) + \delta(a_i, b_j) \end{cases}$$

$D(i, j)$ representa a distância de edição entre os i primeiros caracteres da string a e os j primeiros caracteres da string b . Com isto a função de substituição é definida por:

A substituição idêntica de um símbolo por ele mesmo, tem custo igual a zero:

$$a_i = b_j \Rightarrow \delta(a_i, b_j) = 0$$

Quando os símbolos são diferentes, a substituição tem custo igual a 1:

$$a_i \neq b_j \Rightarrow \delta(a_i, b_j) = 1$$

Além disso, tanto a operação de deleção quanto a de inserção possuem custo unitário:

$$\text{Del}(a) = \text{Ins}(a) = 1$$

Abaixo é apresentado um exemplo do cálculo da distância de edição entre as palavras "corre" e "correr", por meio do preenchimento de uma matriz.

	ε	c	o	r	r	e	r
ε	0	1	2	3	4	5	6
c	1	0	1	2	3	4	5
o	2	1	0	1	2	3	4
r	3	2	1	0	1	2	3
r	4	3	2	1	0	1	2
e	5	4	3	2	1	0	1

Primeiramente, é construída uma matriz de dimensão $(m+1)(n+1)$, onde m corresponde ao comprimento da palavra "corre" e n ao comprimento da palavra "correr".

A primeira linha e a primeira coluna da matriz são preenchidas com números crescentes, representando o número de inserções ou deleções necessárias para transformar uma string vazia na outra.

Em seguida, cada célula interna da matriz é preenchida com base no menor custo entre três possíveis operações: inserção, deleção ou substituição de caracteres, conforme os valores calculados nas células vizinhas. O valor final, localizado na última célula (canto inferior direito), representa a distância mínima de edição entre as duas palavras. No caso da transformação da palavra "corre" em "correr", a distância de Levenshtein é igual a 1.

Pela capacidade de detectar até mesmo pequenas modificações entre textos tais como alterações de caracteres ou estrutura, a Distância de Levenshtein é empregada neste projeto para análise comparativa de códigos-fonte.

No entanto, um dos principais desafios e desvantagens da DL é que ela envolve uma tarefa computacionalmente exigente. Essa métrica normalmente utiliza um algoritmo bem estabelecido, baseado em programação dinâmica, o que a torna uma operação computacional clássica com alto custo em cenários que demandam grandes quantidades de códigos ou sequências extensas.

Essa abordagem, embora eficiente em contextos menores, pode apresentar limitações em situações onde é necessário realizar comparações em larga escala ou com um número elevado de operações simultâneas.

2.6.1 Codebert

Com o objetivo de ir além da métrica mencionada anteriormente e superar as limitações, esta pesquisa também adota o modelo CodeBERT como abordagem complementar para a análise de similaridade de código.

O CodeBERT é o primeiro grande modelo pré-treinado bimodal, desenvolvido para atender demandas das áreas de programação e desenvolvimento de software (FENG et al., 2020). É um modelo capaz de capturar relações semânticas entre a linguagem natural (NL) e a linguagem de programação (PL), gerando representações vetoriais contextuais que auxiliam na avaliação de similaridade e compreensão semântica entre códigos, contribuindo para o aprimoramento de tarefas de análise automática e comparação de soluções em ambientes de programação (FENG et al., 2020).

Conforme descrito por (FENG et al., 2020), o CodeBERT adota uma arquitetura baseada em Transformer bidirecional de múltiplas camadas, semelhante à dos modelos RoBERTa. Portanto supera os modelos de baseline do desafio e também melhora o desempenho de geração de documentação de códigos (ISHTIAQ et al., 2021).

Apesar de seu desempenho superior em muitas tarefas, o modelo pode apresentar limitações em cenários específicos, exigindo ajuste do conjunto de dados de

pré-treinamento.

Por ter sido pré-treinado em um grande corpus de código-fonte e documentação, o CodeBERT demonstra eficácia em tarefas de compreensão e análise de código, como geração de documentação, busca de código e avaliação de similaridade

O modelo em si possui dois objetivos principais: i) Modelagem de Linguagem Mascarada (MLM), e ii) Detecção de Tokens Substituídos (RTD).

A Modelagem de Linguagem Mascarada (MLM) consiste em prever tokens que são mascarados tanto no código ou na linguagem natural (ZHOU; HAN; LO, 2021). Trata-se de ser uma técnica de treinamento utilizada principalmente em redes neurais baseadas em transformadores, como o BERT.

A Detecção de Tokens Substituídos é uma técnica de pré-treinamento utilizada em modelos como o ELECTRA. Essa abordagem foi projetada para treinar o modelo de maneira eficiente ao identificar quais tokens foram substituídos artificialmente de forma sequencial durante o processo de treinamento (ANTOUN; BALY; HAJJ, 2021).

Para transformar textos em vetores, é necessário primeiramente segmentá-los em unidades atômicas, processo denominado tokenização (SANTANA; RIBEIRO, 2022). A partir desses tokens, são então atribuídos valores numéricos que compõem o vetor representativo do texto.

No CodeBERT o pré-processamento de entrada funciona de maneira que cada sequência de código ou texto é tokenizada em subpalavras e estruturada usando tokens especiais: [CLS] no início como token de classificação, [SEP] para separar segmentos e [EOS] para indicar o fim da sequência.

Após esse pré-processamento, cada token é convertido em um embedding que combina três componentes: Token embedding que representa a semântica do token; segment embedding que indica a que segmento o token pertence; e position embedding que informa a posição do token na sequência, podendo ser sequencial, em árvore ou relativa, dependendo da tarefa (XIAO et al., 2025).

(GUO et al., 2020).

Nesta pesquisa o CodeBERT é empregado para transformar cada trecho de código em um vetor numérico (embedding). Esses vetores são então comparados em

um espaço vetorial, permitindo medir a similaridade entre diferentes soluções de programação.

Cada código-solução normalizado é inserido no modelo, que transforma a estrutura do algoritmo em um vetor numérico. Esse vetor representa as características do código em um espaço latente.

Uma vez geradas essas representações vetoriais, utiliza-se a Similaridade do Cosseno para mensurar o grau de proximidade semântica entre diferentes soluções.

A similaridade do cosseno mede o grau de proximidade entre dois vetores não nulos, $\mathbf{u}$ e $\mathbf{v}$, considerando a direção para a qual eles aponta, o que é diferente de medidas que consideram o tamanho dos vetores, essa métrica concentra-se apenas na orientação entre eles.

Dessa forma podemos inferir que quando os valores da similaridade do cosseno estão próximos de 1, eles indicam uma grande semelhança entre os vetores, enquanto valores próximos de -1 indicam forte dissimilaridade.

$$\text{sim}(\mathbf{u}, \mathbf{v}) = \cos(\theta) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|} = \frac{\sum_{i=1}^d u_i v_i}{\sqrt{\sum_{i=1}^d u_i^2} \cdot \sqrt{\sum_{i=1}^d v_i^2}} \quad (2.1)$$

onde:

- $\mathbf{u} \cdot \mathbf{v}$: representa o produto escalar entre os dois vetores de *embeddings*;
- $\|\mathbf{u}\|$ e $\|\mathbf{v}\|$: correspondem às normas euclidianas dos vetores;
- d : representa a dimensionalidade do espaço latente ($d = 768$ no *CodeBERT base*).

A similaridade do cosseno é empregada para quantificar a proximidade angular entre os vetores de embeddings gerados pelo CodeBERT, permitindo mensurar a equivalência funcional e contextual entre o código-solução do novo exercício e os códigos armazenados na base histórica. Desse modo os coeficientes quantitativos obtidos estabelecem a base de entrada para a estimativa automatizada da taxa de acerto do novo problema, cuja confiabilidade preditiva é validada a partir da confrontação com o desempenho real observado dos estudantes.

2.7 Avaliação e Validação do Método

A validação da metodologia proposta é realizada por meio de uma análise de correlação estatística. Este procedimento complementar é fundamental para comprovar a hipótese de que a similaridade entre os códigos é um indicador confiável para prever a dificuldade. O processo de avaliação é dividido em duas partes: a análise da força da relação e a mensuração do erro preditivo.

2.7.1 Coeficientes de Correlação (Validação da Relação)

A análise de correlação constitui um passo fundamental para validar se a dificuldade estimada pelo nosso modelo se comporta de maneira consistente em relação à dificuldade real observada. Por meio dessa análise é possível avaliar o grau de associação entre os valores previstos e os valores reais, verificando se variações na dificuldade estimada acompanham, de forma proporcional as variações na dificuldade efetiva. Para o nosso estudo é adotado o coeficiente de Pearson para a validação estatística da relação linear.

2.7.1.1 Coeficiente de Pearson

A Correlação de Pearson é uma medida estatística que quantifica a força e a direção da relação linear entre duas variáveis numéricas (PARANHOS et al., 2014). A sua fórmula é a seguinte:

$$\rho_{X,Y} = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^n (X_i - \bar{X})^2} \sqrt{\sum_{i=1}^n (Y_i - \bar{Y})^2}}$$

onde X_i e Y_i representam os valores observados para o i -ésimo exercício (dificuldade estimada e dificuldade real, respectivamente), $\bar{X}$ e $\bar{Y}$ correspondem às médias amostrais, e n denota o número total de exercícios analisados.

O coeficiente de correlação de Pearson varia entre -1 e 1, onde o sinal indica a direção da relação (positiva ou negativa), e o valor absoluto reflete a força da correlação. Uma correlação perfeita (-1 ou 1) indica que o valor de uma variável pode ser previsto exatamente quando se conhece o valor da outra.

$\rho = 0$: Um coeficiente de correlação igual a zero indica que as variáveis são ortogonais, ou seja, não apresentam qualquer relação linear entre si (FILHO; JÚNIOR, 2009).

$\rho > 0$ A correlação positiva indica que, quando o valor de uma variável (X) aumenta, a outra variável (Y) também tende a aumentar. $\rho < 0$: Em uma correlação negativa, por outro lado, se a variável (X) aumenta, a variável (Y) tende a diminuir.

O correto é que quanto mais o coeficiente se aproxima de 1, independentemente do sinal, maior é a dependência estatística linear entre as variáveis. Por outro lado, quanto mais o valor se aproxima de 0, mais fraca é a correlação.

2.7.1.2 Significância Estatística (p-value)

Além do valor do coeficiente de correlação de Pearson, é calculado o *p-value* associado, com o objetivo de avaliar a significância estatística da correlação observada. O *p-value* indica a probabilidade de se obter um coeficiente de correlação igual ou mais extremo do que o observado, assumindo a hipótese nula de ausência de correlação linear entre as variáveis. Valores de *p-value* inferiores ao nível de significância adotado indicam que a correlação observada é estatisticamente significativa, fornecendo evidências de que a relação entre a dificuldade estimada e a dificuldade real não ocorre ao acaso (FERREIRA; PATINO, 2015). Como podemos ver representado abaixo

Onde temos:

$$\begin{cases} H_0 : \rho = 0 & \text{(Hipótese nula: não há correlação linear)} \\ H_1 : \rho \neq 0 & \text{(Hipótese alternativa: há correlação linear significativa)} \end{cases}$$

Adota-se o nível de significância de $\alpha = 0,05$. Dessa forma, valores de *p* inferiores a 0,05 são considerados evidências estatisticamente significativas contra a hipótese nula, indicando que a associação observada entre as variáveis é pouco compatível com a hipótese de ausência de relação. Assim, o resultado fornece evidências de que a correlação observada não pode ser atribuída apenas à variabilidade amostral, considerando as premissas do teste estatístico adotado

2.7.2 Avaliação do Desempenho

Para quantificar a magnitude dos desvios entre os valores contínuos estimados e as taxas de acerto reais, o estudo adota duas métricas consolidadas. Essas métricas permitem avaliar a precisão das estimativas produzidas pelo modelo em relação aos valores observados, mensurando o erro entre as previsões e os resultados efetivos. Dessa forma, torna-se possível analisar o grau de acurácia do modelo e identificar eventuais tendências de superestimação ou subestimação da dificuldade dos exercícios.

2.7.2.1 Erro Médio Absoluto (MAE)

Para avaliar o desempenho do modelo proposto, utilizaremos um dos índices mais usados em técnicas de previsão que é o Erro Médio Absoluto (MAE). O MAE é uma métrica que avalia a diferença entre os valores reais e os valores previstos, sem considerar a direção do erro.([NASCIMENTO; JUNIOR; FAGUNDES, 2018](#)).

A sua fórmula é dada pelo seguinte:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

Nessa fórmula, n representa a quantidade total de observações utilizadas no cálculo; y_i corresponde aos valores reais observados; e $\hat{y}_i$ representa os valores estimados ou preditos pelo modelo.

2.7.2.2 Erro Quadrático Médio (RMSE)

O erro quadrado médio (RMSE) é a raiz quadrada da média dos quadrados das diferenças ([SILVA et al.,](#)). E a sua fórmula é representada da seguinte forma:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{j=1}^n (y_j - \hat{y}_j)^2}$$

Onde: O n é o número total de observações; y_j é o valor real da observação; $\hat{y}_j$ é o valor previsto pelo modelo para a observação.

A utilização conjunta do MAE e do RMSE é estratégica devido à diferença de sensibilidade destas métricas em relação aos erros de predição. Enquanto o MAE oferece uma visão linear do erro médio, o RMSE é significativamente mais sensível a outliers. Como o RMSE utiliza o quadrado das diferenças, erros de grande magnitude são penalizados com maior intensidade.

2.8 Considerações Finais

Neste capítulo, foram articulados os conceitos teóricos e metodológicos fundamentais para a compreensão das relações entre complexidade, dificuldade e métricas de similaridade no cenário do ensino de programação. Esse arcabouço conceitual estabelece a sustentação necessária para justificar as escolhas arquiteturais e algorítmicas adotadas no decorrer deste trabalho.

As abordagens de similaridade destacam-se como estratégias promissoras para estimar, de forma antecipada, a dificuldade de problemas de codificação com base em seus respectivos códigos-solução. Sob a perspectiva sintática e estrutural, a Distância de Levenshtein (DL) consolidou-se como uma métrica relevante para aferir discrepâncias textuais diretas em nível de caracteres. Embora exija um custo computacional considerável em decorrência de sua implementação baseada em programação dinâmica, a DL manifesta-se vantajosa em contextos em que o alinhamento de texto explícito é suficiente para discriminar variações de código.

Em contrapartida, os modelos vetoriais densos expandem essa análise ao mapear trechos de código em espaços numéricos contínuos, viabilizando uma avaliação flexível das propriedades semânticas e da equivalência funcional das soluções. Modelos baseados na arquitetura Transformer, como o CodeBERT, tornam-se capazes de reconhecer similaridades conceituais profundas em strings de código de linguagens de programação distintas, reflexo de seu pré-treinamento massivo em repositórios de software.

Para além do cálculo de proximidade algorítmica, o Coeficiente de Correlação de Pearson foi introduzido como o instrumento estatístico responsável por mensurar a

força e a direção da associação linear entre as variáveis investigadas. Apesar de sua conhecida sensibilidade à influência de valores discrepantes (outliers), a métrica atende aos requisitos deste estudo para validar a consistência quantitativa entre os valores inferidos e o comportamento real observado. Complementarmente, a introdução das métricas de erro contínuo (MAE e RMSE) fornece o subsídio matemático indispensável para aferir a acurácia do modelo preditivo, penalizando desvios com base em suas respectivas magnitudes.

Com os alicerces teóricos e as ferramentas de validação devidamente delimitados, torna-se imperativo contextualizar a proposta desta dissertação diante do estado da arte da literatura científica. No capítulo seguinte, serão discutidos os principais trabalhos relacionados, traçando paralelos analíticos e evidenciando as lacunas que motivaram o desenvolvimento desta metodologia.

3

TRABALHOS RELACIONADOS

Diversos fatores que influenciam o processo de aprendizagem dos alunos têm sido objetos de uma grande quantidade de estudos acadêmicos, que buscam identificar métodos eficazes para avaliar e adaptar os exercícios de acordo com as necessidades de cada aluno/turma.

Nas seções seguintes, serão discutidas pesquisas relacionadas a métodos automáticos para estimar a dificuldade de exercícios em sistemas de correção automática de código. Serão abordados métodos que conduzem análises de submissões, métricas de desempenho, técnicas de aprendizagem de máquina, entre outros métodos que têm como objetivo estimar o grau de dificuldade de exercícios de programação.

Para organizar essa discussão, os estudos foram agrupados em três categorias: a primeira trata da análise da complexidade e da dificuldade dos exercícios em si; a segunda reúne trabalhos que propõem métodos para estimar a dificuldade; e a terceira concentra-se em abordagens voltadas à classificação da dificuldade dos exercícios.

3.1 Complexidade e Dificuldade

Compreender a diferença entre complexidade e dificuldade de um exercício de programação é essencial, pois esses dois aspectos afetam diretamente o desempenho dos alunos. A investigação sobre a relação entre complexidade e dificuldade em exercícios de programação tem sido explorada sob diferentes perspectivas. De acordo com [Beckmann, Birney e Goode \(2017\)](#), a complexidade é entendida como uma percepção mental,

relacionada às demandas de processamento de informações impostas ao sistema cognitivo pelas características da tarefa e do contexto situacional. Por sua vez, a dificuldade refere-se ao nível de desafio enfrentado por um indivíduo ao lidar com essas exigências, influenciando diretamente seu desempenho na execução da tarefa.

Segundo [Effenberger, Cechák e Pelánek \(2019\)](#), a complexidade e a dificuldade podem ser estimadas por diferentes métodos e perspectivas, a depender do enfoque adotado no estudo. Nesse contexto, [Fernandes et al. \(2023\)](#) investigaram a relação entre métricas de complexidade de código e as dificuldades enfrentadas por alunos em atividades de programação. O estudo foi conduzido a partir de três abordagens principais: (i) análise de correlação entre variáveis utilizando o coeficiente de Spearman, (ii) predição da dificuldade por meio de modelos de classificação, e (iii) predição de métricas de dificuldade utilizando técnicas de regressão.

A análise de correlação de Spearman entre complexidade de código e dificuldade estudantil revelou uma correlação fraca, indicando que não é possível estimar a dificuldade de um exercício com base em métricas de complexidade. Diante disso, os autores [Fernandes et al. \(2023\)](#) realizaram experimentos adicionais utilizando técnicas de classificação e regressão, que permitem a combinação de múltiplos atributos para uma análise mais robusta da dificuldade.

Já para os de modelos de classificação, observou-se que a escassez de exemplos pertencentes à classe "difícil" comprometeu significativamente o desempenho dos modelos, resultando em baixas taxas de acerto ([FERNANDES et al., 2023](#)). Como consequência, os classificadores não conseguiram distinguir adequadamente entre exercícios fáceis e difíceis (no caso binário), nem entre níveis fácil, médio e difícil (no caso de três classes), tampouco generalizar as classificações para outras métricas de dificuldade. A ausência de dados suficientes da classe "difícil" levou a erros recorrentes de classificação em várias amostras utilizadas nos testes.

Para a predição das métricas de dificuldade por meio de regressão, o melhor desempenho foi alcançado com a métrica de tempo de implementação, utilizando o modelo XGBoost. Os resultados obtidos com os regressores foram semelhantes aos dos classificadores, mas com a vantagem de fornecerem estimativas contínuas, e não apenas

classificações discretas. Assim, o método de regressão mostrou-se mais promissor, pois além de prever a classe de dificuldade, possibilita estimar valores numéricos específicos para cada métrica. No entanto, é importante destacar que a eficácia da predição variou entre as diferentes métricas, sendo mais precisa para algumas do que para outras [Fernandes et al. \(2023\)](#).

[Effenberger, Cechák e Pelánek \(2019\)](#) exploraram diversas medidas de complexidade e dificuldade em quatro conjuntos de problemas de introdução à programação. Um desses conjuntos, composto por 73 exercícios em Python e resolvido por cerca de 2.000 estudantes, recebeu atenção especial. De forma geral, os autores observaram que métricas individuais, como o número de linhas de código ou a quantidade de conceitos de programação abordados, apresentaram correlação fraca com a dificuldade das questões. Esse resultado sugere que tais métricas, quando consideradas isoladamente, não são suficientes para estimar com precisão o nível de dificuldade. No entanto, os autores destacam que a combinação de múltiplas métricas de complexidade pode resultar em modelos preditivos mais robustos. Apesar das correlações fracas entre métricas isoladas e dificuldade, o estudo identificou uma relação consistente e significativa entre os conceitos de complexidade e dificuldade ao considerar os exercícios como um todo.

Tabela 1 – Comparação de estudos sobre complexidade e dificuldade

Autor	Objetivo Principal	Métricas e Métodos	Limitações
Beckmann et al. (2017)	Diferenciar complexidade e dificuldade.	Análise teórica de demandas cognitivas	Não aplica métricas quantitativas.
Effenberger et al. (2019)	Mensurar complexidade e dificuldade	Métricas de código (linhas, conceitos).	Correlação fraca com métricas isoladas
Fernandes et al. (2023)	Analisar a correlação entre as métricas de complexidade e a dificuldade enfrentada pelos alunos.	Spearman, modelos de classificação/regressão.	Desbalanceamento de classes ("difícil").

Para além dos estudos que tratam da relação entre complexidade e dificuldade, no entanto, propuseram-se modelos para prever automaticamente a dificuldade de

exercícios.

3.2 Previsão de Dificuldade

[Silva et al. \(2022\)](#) realizaram um estudo utilizando técnicas de aprendizado de máquina para prever a dificuldade de questões. Para isso, foram analisadas métricas de complexidade do código-solução fornecido pelo professor em comparação com métricas de dificuldade, como as taxas de acerto e a taxa de consultas com erro. O estudo identificou que as taxas de acerto, a taxa de consultas com erro, o tamanho do *log* de resolução e o tempo de implementação apresentaram melhor desempenho preditivo quando as métricas foram extraídas a partir do código-solução do professor.

[Elnaffar \(2016\)](#) Conduziu um estudo com uma turma em que foi ministrada uma disciplina introdutória de programação. Neste trabalho, foi mensurada a dificuldade de 10 exercícios de programação, utilizando o Índice Previsto de Dificuldade (PDI), que é obtido por meio de uma combinação linear de diversas métricas extraídas do código. Os resultados mostraram que o PDI é uma ferramenta eficaz, apresentando um bom desempenho na previsão antecipada da dificuldade de um conjunto de questões. No entanto, o número reduzido de questões analisadas levanta questionamentos quanto à validade dos resultados obtidos. Além disso, não foi avaliado o desempenho em tópicos mais avançados, o que pode limitar a generalização das conclusões

Este trabalho também busca prever a dificuldade de novos exercícios, mas adota uma abordagem distinta daquelas descritas nesta seção. Nessa estratégia, o primeiro passo é procurar por códigos da base que sejam similares ao código-solução fornecido pelo(a) professor(a). Para isso, será empregada uma métrica de similaridade que permitirá identificar a proximidade entre o código-solução do exercício e os códigos da base desenvolvidos por alunos para outros exercícios. A intuição é que quando os alunos desenvolvem códigos para um determinado exercício da base, e esses códigos são semelhantes ao código-solução fornecido pelo professor para o novo exercício, então o exercício da base e o novo exercício possuem dificuldades semelhantes. Desta forma, uma vez identificados os exercícios com códigos similares ao código-solução, os níveis

de dificuldade desses exercícios serão usados para estimar o nível de dificuldade do novo exercício.

Tabela 2 – Comparação de estudos sobre previsão de dificuldade

Autor	Objetivo Principal	Métricas e Métodos	Limitações
Silva et al. (2022)	prever dificuldade via ML.	Taxas de acerto, logs de resolução.	Dependência de código do professor.
Elnaffar (2016)	prever dificuldade.	Índice Previsto de Dificuldade (PDI)	Amostra pequena e falta de avaliação em tópicos avançados

Portanto, enquanto os modelos anteriores focam na previsão, outros estudos se concentram na classificação da dificuldade a partir de diferentes abordagens, como percepção dos alunos ou métricas automatizadas.

3.3 Classificação de dificuldade dos exercícios

Alguns estudos na literatura científica abordam a temática da classificação de dificuldade de exercícios de programação com base na percepção dos usuários, utilizando questionários aplicados antes e após a resolução das questões [Alvarez e Scott \(2010\)](#), [Crosby, Valentine e Williams \(2014\)](#), [Ihantola, Sorva e Vihavainen \(2014\)](#), [Bastian e Mühling \(2022\)](#), [Francisco e Ambrosio \(2015\)](#). O principal problema identificado nessa abordagem é o caráter subjetivo envolvido na classificação da dificuldade da questão pelos usuários, uma vez que eles se baseiam em suas experiências anteriores. Esse aspecto resulta em um alto grau de divergência nos resultados, o que dificulta a classificação precisa da dificuldade. Além disso, o tempo necessário para responder ao questionário pode ser elevado, tornando essa solução pouco escalável.

[Zaffalon et al. \(2019\)](#) apresentaram dois modelos para estimar as habilidades dos alunos: Sistema de Rating e a TRI (Teoria de Resposta ao Item). A TRI estima as habilidades dos alunos com base na probabilidade de um indivíduo acertar um determinado item. Já o Sistema de Rating, originalmente proposto para avaliar jogadores de xadrez,

atribui uma classificação inicial a cada jogador, que vai aumentando conforme ele participa das partidas e obtém resultados. No entanto, os resultados obtidos mostraram diferenças significativas entre os resultados obtidos a partir dos dois modelos.

[Kasto e Whalley \(2013\)](#) realizaram um estudo empírico sobre o uso de métricas de software como método para classificar as questões por dificuldade. No estudo, os autores analisaram dois tipos de questões frequentemente utilizadas em exames de programação: i) rastreamento de código e ii) EiPE (Examine-in-Place Editing). Os resultados mostraram que algumas métricas de software apresentaram correlações com a dificuldade das questões, indicando que elas podem ser ferramentas úteis para a previsão inicial da dificuldade de questões em exames introdutórios de programação do primeiro ano. No entanto, também foi constatado que algumas questões não foram bem capturadas por essas métricas, o que afeta sua validade. Em particular, questões relacionadas a operadores ou conceitos matemáticos podem não ser adequadamente avaliadas pelas métricas utilizadas.

Segundo [Lima et al. \(2020\)](#), foram criados diversos modelos classificadores capazes de prever a dificuldade das questões, utilizando métricas extraídas diretamente dos códigos de solução disponíveis. Nesse processo, um total de 91 métricas foram utilizadas, e a taxa de acerto foi adotada como indicador da dificuldade das questões. Entre os modelos testados, o meta-classificador XGBoost se destacou, apresentando o melhor desempenho em todas as abordagens, com uma acurácia de 95% e um F1-score de 81%.

No final, observou-se que os exercícios que possuem laços de repetição ou intervalos numéricos apresentaram a maior quantidade de respostas erradas. Isso sugere que métricas extraídas do código solução podem ser utilizadas para classificar a dificuldade de um exercício de programação. No entanto, essas métricas não devem ser consideradas como parâmetro único, indicando a necessidade de explorar combinações com outros fatores para uma análise mais robusta.

No outro estudo apresentado em [Lima et al. \(2021\)](#), as questões foram classificadas tanto por facilidade quanto por dificuldade. Para a classificação por facilidade, o modelo obteve um f1-score de 0,84 e uma acurácia de 0,86. Já na classificação por dificul-

dade, os resultados foram ainda melhores, com um f1-score de 0,84 e uma acurácia de 0,96. Todos os atributos utilizados foram extraídos automaticamente a partir do código das soluções desenvolvidas por alunos.

[Meisalo, Sutinen e Torvinen \(2004\)](#) afirmam que os professores frequentemente têm uma percepção equivocada sobre o nível de dificuldade de um exercício em comparação com a experiência dos alunos. Isso ocorre devido à maneira como o nível de dificuldade é avaliado. Além disso, diversos outros fatores podem influenciar essa percepção, como, por exemplo, questões que exigem muito tempo para serem resolvidas. Esses exercícios longos podem levar ao desânimo dos alunos, o que, por sua vez, aumenta a probabilidade de respostas erradas. [Derus e Ali \(2012\)](#) realizaram uma pesquisa com 105 alunos, na qual foi identificado que os estudantes enfrentaram dificuldades em compreender os conceitos básicos de estrutura de programação e em projetar programas para resolver tarefas simples. Além disso, observou-se que a falta de exemplos fornecidos pelos professores impactou negativamente a compreensão dos alunos. Como sugestão para melhorar o aprendizado, foi recomendado o uso de ferramentas de visualização para facilitar a aprendizagem da programação.

Tabela 3 – Classificação e percepção de dificuldade

Autor	Objetivo Principal	Métricas e Métodos	Limitações
Zaffalon et al. (2019)	estimar a habilidade dos estudantes	Sistema de Rating e Teoria TRI.	Discrepâncias entre modelos.
Kasto et al. (2013)	Classificar dificuldade	restreamento do código e EiPE.	Falhas em capturar questões envolvendo operadores matemáticos ou conceitos abstratos.
Lima et al. (2020)	Classificação automática com XGBoost	91 métricas de código	Necessidade de combinar métricas com outros fatores externos
Meisalo et al. (2004)	Avaliar percepção de dificuldade	Questionários subjetivos	Alta variabilidade e baixa escalabilidade.

3.4 Considerações Finais

Em síntese, a revisão dos trabalhos relacionados revela avanços significativos, mas também desafios persistentes, na análise da complexidade, precisão e classificação de exercícios de programação. Importante diferenciar a complexidade e a dificuldade, pois nem sempre há uma relação direta entre os dois conceitos. Os estudos feitos reforçam a compreensão de que, embora a complexidade de um exercício possa ser mensurada, a tradução dessa complexidade em dificuldade depende de múltiplas variáveis, incluindo aspectos cognitivos e contextuais [Elnaffar \(2016\)](#).

No que tange a modelagem preditiva as abordagens apresentados por [Silva et al. \(2022\)](#) e [Elnaffar \(2016\)](#) demonstram maior eficácia na previsão de dificuldades. O primeiro utiliza métricas combinadas para realizar as previsões, embora a dependência excessiva do código possa comprometer a precisão dos resultados. O segundo emprega o índice PDI em conjunto com métricas combinadas, mostrando-se promissor, apesar da limitação quanto à quantidade de dados disponíveis, o que pode afetar a robustez do modelo.

Na classificação da dificuldade dos exercícios, modelos como o ELO e o TRI se destacam pela objetividade, porém enfrentam limitações para capturar nuances conceituais e contextuais do processo de aprendizagem. Já o meta classificador destacou-se tendo mostrado um melhor desempenho em todas as abordagens feitas.

Para [Meisalo, Sutinen e Torvinen \(2004\)](#) as limitações dos questionários tradicionais, somadas às diferenças entre a percepção dos professores e a experiência vivida pelos alunos, evidenciam a necessidade de métodos híbridos que integrem dados quantitativos e qualitativos. Portanto é necessário combinar análises automáticas do código com dados reais de desempenho dos alunos, permitindo a criação de um modelo mais preciso e eficaz, tanto para professores quanto para sistemas de apoio ao ensino.

Diante desse panorama, a metodologia proposta nesta dissertação preenche tal lacuna ao estruturar um mecanismo baseado em similaridade semântica e textual que conecta a solução de referência do docente à base de dados de submissões históricas. Com os fundamentos teóricos consolidados (Capítulo 2) e o estado da arte delimitado diante da literatura (Capítulo 3), o capítulo seguinte descreve detalhadamente a

especificação técnica do método desenvolvido.

4

MÉTODO

Neste capítulo é detalhado o método utilizado para estimar o nível de dificuldade de exercícios de programação inéditos ou seja, que ainda não foram resolvidos por alunos em sistemas de juízes online.

Quando um novo exercício é cadastrado, não há métricas empíricas disponíveis que permitam avaliar sua dificuldade. No entanto, é importante oferecer ao(à) professor(a) uma estimativa inicial para avaliar se o exercício é apropriado ao perfil da turma.

Métricas como taxa de acerto e número médio de submissões são amplamente utilizadas para estimar a dificuldade de exercícios resolvidos ([EFFENBERGER; CECÁK; PELÁNEK, 2019](#)), mas não são aplicáveis a exercícios inéditos. Neste contexto, propomos um método que utiliza o código-solução elaborado pelo professor como base para estimar a dificuldade, por meio da comparação com soluções de exercícios já resolvidos.

Atualmente, não existe um método claro e específico para estimar exercícios de programação que ainda não foram resolvidos. Diante desse desafio, esta pesquisa propõe o uso dos códigos solução como alternativa para estimar a dificuldade desses exercícios. Um código-solução é um exemplo de resposta para um exercício, criado pelo professor no momento da elaboração do exercício. Ele auxilia os juízes online na definição dos casos de teste que serão utilizados para avaliar os exercícios.

Considerando que novos exercícios e suas respectivas soluções são inseridos no sistema online, onde não há até o momento indicadores disponíveis quanto ao seu nível

de dificuldade uma vez que se trata de um exercício recém-cadastrado e que ainda não foi resolvido por nenhum aluno, a estratégia a ser adotada neste trabalho consiste em buscar na base de dados exercícios onde os alunos produziram soluções semelhantes ao código-solução do novo exercício e então usar métricas como taxa de acerto desses exercícios já resolvidos para estimar a dificuldade de novo exercício.

Após o cadastro do exercício e seu código-solução, é feita uma busca na base por exercícios onde os alunos produziram códigos semelhantes ao código-solução cadastrado pelo professor, permitindo assim uma estimativa mais precisa da dificuldade do exercício.

Neste trabalho, a estratégia utilizada para estimar a dificuldade de um novo exercício de avaliação baseia-se no cálculo da similaridade entre o código solução de um novo exercício cadastrado pelo professor e os códigos criados em avaliações anteriores, abrangendo tanto os códigos desenvolvidos pelos alunos quanto aos códigos solução criados pelos professores. Abaixo, a Figura 3 apresenta o diagrama de fluxo que esquematiza as etapas metodológicas, desde o cadastro do novo exercício até a estimativa de sua dificuldade.

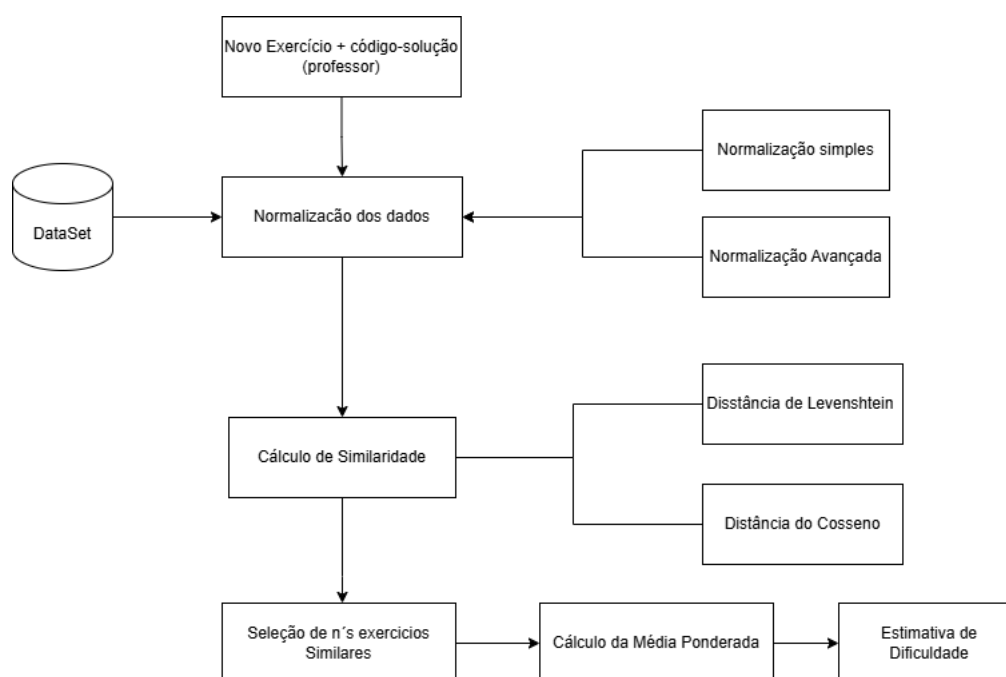


Figura 3 – Representação esquemática do método

4.1 Cadastro do Exercício

O processo tem início com o cadastro do novo exercício no sistema pelo docente responsável. Nessa fase além do enunciado descritivo do problema, o professor insere também o respectivo código-solução, que representa uma implementação considerada correta e pedagogicamente adequada para a resolução da tarefa proposta.

Após o cadastro o sistema valida o código inserido, verificando sua consistência sintática e garantindo que esteja em conformidade com a linguagem de programação definida para o exercício.

Essa etapa inicial é fundamental, pois a qualidade, clareza e representatividade do código-solução têm um impacto direto na precisão das estimativas de dificuldade, uma vez que todo o processo de similaridade é construído a partir dessa referência.

No entanto, o código cadastrados e os códigos contidos na base passam antes por uma etapa de pré-processamento, visando a normalização desses códigos. Neste trabalho, são propostos dois tipos de normalização, a normalização simples e a avançada, conforme descrito na seção a seguir.

4.2 Normalização dos códigos

A normalização de código visa principalmente identificar e remover partes irrelevantes, facilitando a padronização e comparação. Além disso, ela desconsidera modificações superficiais inseridas para ocultar plágio, pois essas alterações não afetam a funcionalidade essencial do programa.

O código de programação original ilustrado na figura acima será submetido à normalização simples e avançada, resultando em uma estrutura otimizada que possibilitará o cálculo de similaridade

```
# Introducao a Programacao de Computadores
# Trabalho Pratico 01 - Variaveis e Estrutura Sequencial
# Versao B
# Conserva o do momentum: peixes
# Criado em 08 / 03 / 2024
# @author: IComp / UFAM

mG = float(input("massa peixe Grande: "))
vG = float(input("velocidade peixe Grande: "))
mP = float(input("massa peixe Pequeno: "))
vP = float(input("velocidade peixe Pequeno: "))

vf = (mG * vG - mP * vP) / (mG + mP)

print(vf)
```

Figura 4 – Código Original

4.2.1 Normalização simples

A normalização simples visa padronizar os códigos submetidos, eliminando elementos que não impactam a lógica do programa. São removidos comentários, linhas em branco, nomes originais de variáveis e parâmetros de entrada. Strings são substituídas por representações vazias e a indentação é uniformizada. Esse processo facilita a comparação entre códigos que implementam a mesma lógica com diferentes estilos. O processo de normalização simples é feito em cada código da base e envolve as seguintes etapas:

Normalização do nome das variáveis: O processo consiste em renomear as variáveis do programa de forma sequencial, atribuindo à primeira variável o identificador 'a', à segunda o identificador 'b', à terceira o identificador 'c', e assim sucessivamente. É importante ressaltar que, caso uma variável apareça em múltiplas partes do código, todas as suas ocorrências devem ser substituídas pelo mesmo identificador. Por exemplo, a primeira variável encontrada no programa será renomeada para 'a', e todas as ocorrências subsequentes dessa mesma variável também serão substituídas por 'a', garantindo a consistência do mapeamento;

Remoção de comentários: Este procedimento consiste em eliminar todos os comentários presentes nos códigos, abrangendo tanto comentários de linha única quanto aqueles de múltiplas linhas. Essa remoção visa reduzir elementos que não impactam diretamente na lógica ou estrutura do programa, contribuindo para uma representação

mais essencial do código;

Normalização de inputs: Esse procedimento consiste em remover os parâmetros fornecidos às funções input. O objetivo é padronizar a forma como as entradas do usuário são representadas no código, eliminando possíveis variações que não impactem a lógica do programa;

Substituição de strings: Todas as strings presentes no código são substituídas por uma string vazia representada por duas aspas duplas (). Essa transformação busca uniformizar os valores literais textuais, concentrando a análise na estrutura lógica do programa;

Remoção de linhas em branco: Todas as linhas em branco são eliminadas dos códigos, com o objetivo de reduzir variações estéticas ou estruturais que não influenciam diretamente a lógica do programa.

Padronização de indentações: Todas as indentações no código são substituídas por duas unidades de espaço em branco. Essa padronização elimina discrepâncias na formatação que poderiam interferir na análise comparativa entre diferentes códigos.

```
a = float(input())
b = float(input())
c = float(input())
d = float(input())
e = (a * b - c * d) / (a + c)
print(e)
```

Figura 5 – Apresentação do código em Python após o processo da normalização simples

4.2.2 Normalização avançada

A normalização avançada é um processo de transformação mais radical dos códigos-fonte, cujo objetivo é reduzir ainda mais as variações estilísticas e semânticas entre as diferentes implementações. Nesse processo, apenas símbolos e tokens essenciais da linguagem são preservados, incluindo operadores, palavras-chave e identificadores padrão. Além disso, strings literais são substituídas por uma string vazia padrão (), e elementos irrelevantes à lógica do programa, como comentários de linha e de múltiplas

linhas, são completamente removidos. Essa abordagem busca abstrair os códigos para uma representação mínima e uniforme, concentrando-se exclusivamente na essência lógica e estrutural das soluções apresentadas. Note que, ao contrário do processo de normalização simples, no processo de normalização avançada o código deixa de seguir as regras de sintaxe da linguagem original, e portanto não pode mais ser interpretado ou compilado.

De uma forma mais detalhada, o processo de normalização avançada é feito em cada código da base e envolve as seguintes etapas:

Substituição de strings por : Todas as strings literais no código são substituídas por uma string vazia (), eliminando informações textuais específicas.

Remoção de comentários: Comentários de linha e de múltiplas linhas são completamente eliminados para evitar interferências de anotações ou explicações no código.

Eliminação de parâmetros de funções input: Todos os parâmetros das chamadas da função input são removidos, padronizando sua aparência e comportamento.

Remoção de símbolos e tokens não essenciais: Apenas os símbolos e tokens mais relevantes, como operadores, palavras-chave da linguagem e identificadores padrão, são mantidos no código. No caso do Python, os símbolos e tokens mantidos são: `':', ',', '(', ')', '[', ']', '"', '*', '-', '+', '%', '!', ':', '/', ' ', '=', '>', '<', 'and', 'or', 'numpy', 'os', 'sys', 'random', 'math', 'False', 'None', 'True', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars' e 'zip'.`

Redução a uma forma uniforme: O código é abstraído a uma representação mínima, destacando apenas os elementos essenciais à sua lógica estrutural.

```
=float(input())=float(input())=float(input()) =float(input()) =(*-*)  
/(+) print()
```

Figura 6 – Apresentação do código em Python após o processo da normalização avançada

Em seguida a realização dessas etapas citadas anteriormente, os códigos são comparados e calcula-se a similaridade entre o código solução cadastrado pelo professor e todos os códigos desenvolvidos para todos os exercícios da mesma disciplina para qual o novo exercício foi criado. Para os códigos similares encontrados, são selecionados para próxima etapa do método, onde é feita a estimativa de dificuldade dos exercícios dos códigos encontrados.

4.3 Cálculo de Similaridade

O cálculo de similaridade é realizado com base no código-solução conforme mencionado anteriormente, comparando-o com os códigos já armazenados na base de dados de um juiz online. Assim que o novo código é registrado, o sistema realiza a seleção de n exercícios que apresentam semelhança com o código acabado de cadastrar, a partir dessa seleção, a similaridade é calculada por meio de comparações diretas entre os códigos, permitindo determinar com precisão o grau de semelhança entre o novo exercício e os existentes na base. Para o cálculo de similaridade avaliamos duas estratégias a ser adotada entre códigos: A Distância de Levenshtein, e o Cálculo do Cosseno.

A utilização da Distância de Levenshtein (DL) e o Cálculo do cosseno fundamenta-se na premissa de que ambos são métodos eficientes para o cálculo do grau de similaridade entre exercícios, considerando aspectos textuais e semânticos.

A Distância de Levenshtein destaca-se por sua capacidade de medir a similaridade entre sequências de caracteres de forma intuitiva e eficaz, mesmo quando existem pequenas variações textuais entre as strings. Por outro lado, o Cálculo do Cosseno quando aplicado sobre representações vetorizadas do código, permite medir similaridade semântica entre soluções distintas.

4.4 Estimativa da dificuldade de novos exercícios

A determinação da dificuldade dos novos exercícios pressupõe o reconhecimento de sua natureza subjetiva e a análise criteriosa das métricas de dificuldade disponibilizadas pelos sistemas de juízes online.

Nesta etapa do trabalho, é importante frisar que não é possível chegar a um valor exato e preciso para a dificuldade de um exercício. A dificuldade precisa ser estimada, porque ela é uma característica subjetiva que varia de acordo com o aluno e com o contexto. O que pode ser considerado difícil para um iniciante, sem experiência em conceitos como interações ou condicionais, pode ser trivial para alguém experiente na área.

Além disso, fatores como motivação, autoconfiança e até mesmo o ambiente em que o problema é resolvido influenciam a percepção de dificuldade. Resolver um exercício sob pressão de tempo, como em uma avaliação, pode ser muito mais desafiador do que em uma lista de exercícios, onde o aluno tem acesso a recursos como materiais didáticos e a ajuda de colegas. A forma como o problema é apresentado, incluindo a clareza das instruções, também afeta significativamente a percepção de dificuldade.

Desta forma, mesmo as métricas usadas nesse trabalho para estimar a dificuldade, como a taxa de acerto ou o tempo médio para resolver o exercício, são insuficientes para capturar toda a variabilidade de percepções e contextos. Essas estimativas, portanto, estão sempre sujeitas a variações que dependem de fatores intrínsecos ao exercício e de características dos alunos. Por essas razões, nunca é possível determinar a dificuldade de um exercício com absoluta precisão, apenas aproximá-la para contextos específicos.

4.4.1 Métricas de Dificuldade

Em juízes online cada exercício resolvido pelos alunos pode ser associado a métricas quantitativas que indicam sua dificuldade de forma objetiva, permitindo a calibração de modelos preditivos. Entre essas métricas, destacam-se

- **Taxa de acerto:** proporção de alunos que resolveram corretamente o exercício em relação ao total de tentativas.

- **Número médio de submissões:** média de submissões por aluno, independentemente do sucesso.
- **Tamanho médio dos códigos corretos:** média de linhas dos códigos que obtiveram sucesso.
- **Tempo médio de codificação:** tempo efetivamente gasto na implementação, desconsiderando pausas superiores a três minutos.

Embora todas essas métricas possam contribuir para a análise, este trabalho adota exclusivamente a taxa de acerto como métrica auxiliar na estimativa da dificuldade de exercícios inéditos.

Desta forma, neste trabalho, a dificuldade de um novo exercício é estimada com base na taxa de acerto dos exercícios cujos códigos (submetidos por alunos ou códigos-solução) são mais semelhantes ao código-solução do novo exercício. Para isso, utiliza-se uma média ponderada, em que o peso de cada exercício é determinado pelo grau de similaridade entre o código-solução recém-cadastrado e os códigos previamente submetidos pelos alunos.

O processo ocorre da seguinte forma: após a normalização e cálculo de similaridade entre o código-solução do novo exercício e os códigos existentes na base, identificam-se os n códigos mais semelhantes. Cada um desses códigos está vinculado a um exercício previamente resolvido, cuja taxa de acerto é conhecida. A similaridade entre o código-solução e cada código submetido é utilizada como peso na média ponderada que define a dificuldade estimada.

A fórmula da média ponderada é dada por:

$$S = \frac{\sum_{i=1}^n (\text{sim}_i \times T_i)}{\sum_{i=1}^n \text{sim}_i}$$

onde:

- S : estimativa de dificuldade do novo exercício;
- n : número de códigos similares considerados;
- sim_i : grau de similaridade entre o código-solução;

- T_i : taxa de acerto do exercício associado ao código i .

Ao final do processo, o novo exercício recebe uma estimativa de dificuldade derivada da taxa de acerto dos exercícios cujos códigos submetidos apresentam maior semelhança com seu código-solução. Essa abordagem permite antecipar a estimativa de dificuldade mesmo antes dos alunos tentarem resolver o exercício.

4.4.2 Avaliação do Desempenho Preditivo

Para validar a confiabilidade do método proposto e testar a hipótese de pesquisa, o procedimento de avaliação adota o seguinte protocolo:

1. **Análise de Associação Linear:** cálculo do Coeficiente de Correlação de Pearson e do respectivo *p-value* entre as estimativas inferidas e as taxas reais de acerto ;
2. **Quantificação do Erro Contínuo:** mensuração do Erro Médio Absoluto (MAE) e da Raiz do Erro Quadrático Médio (RMSE) para aferir a precisão da predição sob diferentes limiares de similaridade e combinações de normalização.

5

EXPERIMENTOS

Este capítulo apresenta o planejamento e a execução dos experimentos conduzidos para avaliar o método proposto no capítulo anterior. Tal método baseia-se no uso de códigos-solução elaborados pelo professor no momento da criação dos exercícios, permitindo ao docente antecipar a dificuldade das tarefas antes de sua aplicação em sistemas de juízes online. Para sustentar essa análise, são descritos, nas seções seguintes, o contexto educacional, o conjunto de dados utilizado, o protocolo experimental e os procedimentos adotados para a obtenção e validação dos resultados.

5.1 Contexto Educacional

Os experimentos foram realizados em turmas da disciplina de Introdução à Programação de Computadores (IPC) da Universidade Federal do Amazonas (UFAM), ofertada para cursos fora da área de computação, como Engenharias, Física e Matemática. Nessas turmas, o processo de ensino-aprendizagem é apoiado pelo uso do juiz online Code-Bench, por meio do qual são disponibilizadas listas de exercícios voltadas à prática dos conteúdos abordados. Essas listas possuem caráter formativo, sendo utilizadas como instrumento de treinamento e consolidação do conhecimento.

Além dessas atividades, a disciplina também contempla avaliações formais realizadas em ambiente controlado, com o objetivo de mitigar a ocorrência de plágio. A organização do conteúdo segue uma progressão estruturada em módulos que incluem: programação sequencial; estruturas condicionais; condicionais encadeadas; estruturas

de repetição por condição; vetores e strings; estruturas de repetição por contagem; e matrizes. Essa progressão favorece o desenvolvimento gradual das habilidades fundamentais de programação e serve de base para a construção do dataset utilizado neste trabalho.

5.2 Dataset

O dataset utilizado foi extraído do juiz online *Codebench*, com dados coletados em turmas de IPC da UFAM entre os anos de 2016 e 2023.

Embora a base contenha códigos provenientes tanto de listas quanto de avaliações, foram considerados apenas aqueles submetidos em avaliações. Essa escolha se deve ao fato de que, em listas de exercícios, os alunos frequentemente recorrem a materiais externos ou a soluções de terceiros, o que pode distorcer a dificuldade percebida das tarefas. Assim, restringir a análise aos dados de avaliações contribui para uma estimativa mais fidedigna da dificuldade dos exercícios.

Ao todo, a base contempla 958 exercícios aplicados em avaliações nesse período, totalizando 42.245 códigos submetidos pelos alunos.

Para avaliar a precisão do método proposto, foi selecionado aleatoriamente um subconjunto de 200 exercícios ($N=200$). Para cada um deles, foi estimada a dificuldade, sendo a qualidade das estimativas avaliada por meio das métricas *Root Mean Squared Error* (RMSE), *Mean Absolute Error* (MAE) e coeficiente de correlação de Pearson.

Cabe destacar que, embora a avaliação experimental seja realizada sobre esse subconjunto de 200 exercícios, o processo de estimativa de dificuldade considera a totalidade dos 958 exercícios disponíveis na base.

5.3 Ambiente Computacional

Todas as etapas do protocolo experimental, incluindo pré-processamento, normalização, vetorização e cálculo das métricas de similaridade, foram implementadas na linguagem Python. As bibliotecas *Pandas* e *NumPy* foram utilizadas para a manipulação,

organização e processamento dos dados. As bibliotecas *SciPy* e *Scikit-learn* foram empregadas para o cálculo das métricas de avaliação, incluindo o coeficiente de correlação de Pearson, por meio da função `pearsonr`, e métricas de erro como MAE e MSE.

O módulo *re* foi utilizado no processo de normalização do código-fonte, permitindo a aplicação de regras de limpeza textual, como remoção de comentários, substituição de literais de string, padronização de indentação e filtragem seletiva de símbolos e palavras-chave.

A biblioteca *transformers* foi utilizada para carregar o modelo de linguagem pré-treinado CodeBERT, enquanto o framework *PyTorch* foi empregado para a execução eficiente do modelo. A similaridade entre os vetores gerados foi calculada por meio da função `cosine_similarity`, disponível na biblioteca *Scikit-learn*.

5.4 Protocolo Experimental

O protocolo experimental deste estudo foi desenhado para avaliar a eficácia dos modelos implementados na predição de dificuldade de exercícios de programação. O fluxo consiste na normalização sistemática dos códigos, extração de representações vetoriais (embeddings) e cálculo de similaridade, culminando na análise de correlação estatística entre os resultados do modelo e os dados reais de desempenho dos estudantes na plataforma Codebench.

5.4.1 Pré-processamento e Normalização

Conforme apresentado na Seção 4.2, a metodologia empregada nesta pesquisa adotada duas estratégias de normalização de código-fonte: a normalização simples e a normalização avançada.

Enquanto a normalização simples foca em testar cenários mais próximos do código real escrito pelo aluno, a normalização avançada foca estritamente na estrutura lógica do algoritmo. Essa distinção permite avaliar se o modelo é capaz de identificar a essência funcional do código, independentemente da variabilidade estilística adotada

pelo estudante.

5.4.2 Conjuntos de Códigos Comparação

Conforme apresentado no Capítulo 4, a estimativa da dificuldade de um exercício inédito A é realizada por meio da identificação, na base de dados, de exercícios previamente resolvidos cujos códigos apresentem alta similaridade com o código-solução de A . A hipótese central é que exercícios cujas soluções são estrutural ou semanticamente semelhantes tendem a compartilhar níveis de dificuldade semelhantes. Dessa forma, uma vez identificados esses exercícios relacionados, suas métricas de dificuldade são utilizadas como base para inferir a dificuldade do exercício alvo.

Nos experimentos conduzidos neste capítulo, são consideradas duas estratégias distintas para a definição do conjunto de comparação, isto é, do conjunto de códigos utilizados no processo de busca por similaridade.

Na primeira estratégia, o conjunto de comparação é composto exclusivamente pelos códigos-solução elaborados pelos professores para os 958 exercícios da base. Essa abordagem permite avaliar o desempenho do método em um cenário mais controlado, no qual os códigos apresentam menor variabilidade e seguem padrões mais consistentes de implementação.

Na segunda estratégia, o conjunto de comparação é ampliado para incluir, além dos códigos-solução, todos os códigos desenvolvidos pelos alunos ao longo das avaliações, totalizando 43.203 códigos (958 códigos-solução e 42.245 códigos de alunos). Nesse caso, o processo de comparação passa a considerar um espaço de busca significativamente mais heterogêneo, incorporando diferentes estilos de programação, abordagens parciais e possíveis inconsistências lógicas.

A adoção dessas duas estratégias permite analisar o impacto da variabilidade dos dados no desempenho do método proposto. Em particular, a segunda estratégia possibilita avaliar a robustez da abordagem frente a ruídos inerentes ao processo de aprendizagem, investigando se o modelo é capaz de identificar corretamente relações de similaridade relevantes mesmo na presença de implementações incompletas ou

incorretas.

5.4.3 Uso de Blocos de Exercícios

No juiz online CodeBench, as avaliações são estruturadas em um conjunto de blocos de exercícios, sendo que cada bloco contém um ou mais exercícios. Por exemplo, uma avaliação pode ser composta por quatro blocos, e cada bloco pode conter dez exercícios distintos.

Quando um aluno acessa a avaliação pela primeira vez, o sistema realiza o sorteio de um exercício de cada bloco, compondo assim a prova individual do estudante. No exemplo citado, o aluno receberia uma avaliação com quatro questões, sendo cada uma delas selecionada a partir de um dos blocos disponíveis.

Esse mecanismo é amplamente utilizado como estratégia para mitigar a ocorrência de plágio, especialmente em ambientes controlados, como laboratórios presenciais. Além disso, na disciplina de Introdução à Programação de Computadores da UFAM, os docentes organizam os blocos de modo que os exercícios de um mesmo grupo apresentem enunciados e níveis de dificuldade semelhantes, garantindo maior equidade entre as provas geradas.

No contexto deste trabalho, essa característica introduz uma limitação importante. Como os exercícios de um mesmo bloco são resolvidos simultaneamente durante a avaliação, não há dados prévios de desempenho que permitam estimar a dificuldade de um exercício com base nos demais exercícios do mesmo bloco. Isso decorre do fato de que o método proposto depende da comparação com exercícios previamente resolvidos, cujas soluções apresentem similaridade com o código-solução do exercício alvo.

Diante disso, foram definidos dois cenários experimentais para analisar o impacto dessa restrição na estimativa de dificuldade de um exercício A : (i) um cenário em que os exercícios pertencentes ao mesmo bloco de A são removidos do conjunto de comparação, representando uma situação mais realista; e (ii) um cenário em que esses exercícios são mantidos, configurando um ambiente controlado no qual há disponibilidade de exercícios altamente similares ao exercício avaliado.

A análise comparativa entre esses dois cenários permite investigar tanto a robustez do método em condições realistas quanto seu desempenho em um limite superior, no qual exemplos fortemente relacionados estão disponíveis na base de dados.

5.4.4 Métricas de Dificuldade

Em juízes online, cada exercício resolvido pelos alunos pode ser associado a diferentes métricas quantitativas que permitem caracterizar sua dificuldade de forma objetiva. Entre essas métricas, destacam-se a taxa de acerto, o número médio de submissões, o tamanho médio dos códigos corretos e o tempo médio de codificação.

Embora essas métricas forneçam perspectivas complementares sobre o esforço necessário para a resolução dos exercícios, neste trabalho adotamos exclusivamente a **taxa de acerto** como base para a estimativa de dificuldade. Essa escolha se deve ao fato de que essa métrica expressa diretamente a proporção de alunos que conseguiram resolver corretamente um exercício, sendo, portanto, uma medida simples e interpretável de sua dificuldade.

A partir dessa definição, a dificuldade de um novo exercício A é estimada com base na taxa de acerto de exercícios previamente resolvidos cujos códigos são mais semelhantes ao seu código-solução. Para isso, utiliza-se um processo de comparação entre o código-solução de A e um conjunto de códigos existentes na base, identificando aqueles que apresentam maior similaridade.

Neste capítulo, adotamos $n = 10$, ou seja, a estimativa de dificuldade de cada exercício é baseada nos 10 exercícios mais similares encontrados. A dificuldade de A é então calculada por meio de uma média ponderada das taxas de acerto desses exercícios, na qual os pesos são determinados pelos respectivos graus de similaridade.

Dessa forma, exercícios mais semelhantes ao código-solução de A exercem maior influência na estimativa final, permitindo obter uma medida de dificuldade consistente mesmo na ausência de interações prévias de alunos com o novo exercício.

5.5 Resultados

Esta seção apresenta os resultados obtidos a partir das diferentes configurações experimentais, com o objetivo de comparar o desempenho de abordagens semânticas (CodeBERT) e textuais (Levenshtein) na estimativa da dificuldade de exercícios de programação, considerando variações de normalização, escopo do dataset e restrições temáticas.

5.5.1 Experimentos com o CodeBERT

A Tabela 4 mostra os resultados obtidos para os experimentos com o CodeBERT, sem considerar códigos de um mesmo bloco. A primeira coluna indica o tipo de normalização. A segunda indica o conjunto de códigos comparação usado - Professores indica que o conjunto de comparação é composto exclusivamente pelos códigos-solução elaborados pelos professores; Todos indica que o conjunto de comparação é ampliado para incluir, além dos códigos-solução, todos os códigos desenvolvidos pelos alunos ao longo das avaliações.

Com base nos resultados apresentados na Tabela 4, observa-se que o desempenho do CodeBERT é bastante limitado quando os exercícios do mesmo bloco são removidos do conjunto de comparação. De modo geral, os coeficientes de correlação de Pearson são baixos em todas as configurações, variando entre 0,0267 e 0,2380, indicando uma fraca relação linear entre a similaridade semântica dos códigos e a dificuldade real dos exercícios. Além disso, a maioria dos resultados não apresenta significância estatística (p-value elevado), com exceção da configuração com normalização avançada e uso de todos os códigos, que, embora significativa ($p = 0,0170$), ainda apresenta uma correlação de baixa magnitude. As métricas de erro (RMSE e MAE) também não evidenciam melhorias consistentes entre as configurações, sugerindo que nem o tipo de normalização nem a ampliação do conjunto de comparação são suficientes para compensar a ausência de exercícios semanticamente próximos. Esses achados reforçam que, sem a presença de exercícios do mesmo bloco, o modelo tem dificuldade em capturar relações relevantes para a estimativa de dificuldade, comprometendo sua eficácia em cenários mais realistas.

Em contraste com o cenário anterior, a Tabela 5 evidencia uma melhora significativa no desempenho do CodeBERT quando exercícios do mesmo bloco são considerados no conjunto de comparação. Observa-se uma redução consistente nas métricas de erro (RMSE e MAE) em todas as configurações, indicando maior precisão nas estimativas de dificuldade. Além disso, os coeficientes de correlação de Pearson apresentam valores moderados a altos, variando entre 0,4931 e 0,5876, todos estatisticamente significativos ($p\text{-value} < 0,001$), o que demonstra uma relação linear consistente entre a similaridade semântica e a dificuldade real dos exercícios. Destaca-se a configuração com normalização avançada e uso de todos os códigos, que alcança o melhor desempenho geral, com menores erros e maior correlação. Esses resultados indicam que a presença de exercícios semanticamente próximos, como ocorre dentro de um mesmo bloco temático, é um fator determinante para o sucesso do método, permitindo que o modelo capture padrões relevantes e produza estimativas de dificuldade mais confiáveis.

Normalização	Códigos	RMSE	MAE	Pearson	p-value
Simples	Professores	0.1653	0.1179	0.0430	0.6709
Simples	Todos	0.2129	0.1387	0.1004	0.3198
Avançada	Professores	0.1656	0.1144	0.0267	0.7916
Avançada	Todos	0.1869	0.1266	0.2380	0.0170

Tabela 4 – Comparação de desempenho do modelo CodeBERT quando exercícios do mesmo bloco não são mantidos

Normalização	Códigos	RMSE	MAE	Pearson	p-value
Simples	Professores	0.1338	0.0886	0.4931	1.856e-07
Simples	Todos	0.1509	0.0929	0.5604	1.330e-09
Avançada	Professores	0.1292	0.0782	0.5547	2.110e-09
Avançada	Todos	0.1287	0.0841	0.5876	1.294e-10

Tabela 5 – Comparação de desempenho do modelo CodeBERT quando exercícios do mesmo bloco não são mantidos

5.5.2 Experimentos com o Levenshtein

No que diz respeito à aplicação da métrica de similaridade baseada na distância de Levenshtein, os resultados apresentados na Tabela 6 indicam um desempenho bastante limitado quando exercícios do mesmo bloco não são considerados no conjunto de

comparação. Observa-se que os coeficientes de correlação de Pearson são baixos e, em alguns casos, negativos (como nas configurações com códigos de professores), o que evidencia a ausência de uma relação consistente entre a similaridade textual dos códigos e a dificuldade real dos exercícios. Além disso, nenhum dos resultados apresenta significância estatística ($p\text{-value} > 0,05$), reforçando que a abordagem não é capaz de capturar padrões relevantes nesse cenário. As métricas de erro (RMSE e MAE) também se mantêm relativamente elevadas e sem variações expressivas entre as configurações, sugerindo que nem o tipo de normalização nem a ampliação do conjunto de códigos são suficientes para melhorar o desempenho do método em um ambiente heterogêneo.

Por outro lado, conforme evidenciado na Tabela 7, o desempenho da abordagem baseada em Levenshtein melhora substancialmente quando a comparação é restrita a exercícios pertencentes ao mesmo bloco temático. Nesse cenário, observa-se uma redução consistente nas métricas de erro em todas as configurações, acompanhada de coeficientes de correlação de Pearson moderados a fortes, variando entre 0,5219 e 0,6710, todos estatisticamente significativos ($p\text{-value} < 0,001$). Destaca-se a configuração com normalização avançada e códigos de professores, que apresenta o melhor desempenho geral, com menor RMSE e MAE e maior correlação. Esses resultados indicam que, embora a distância de Levenshtein não capture aspectos semânticos do código, a forte similaridade estrutural entre soluções de exercícios de um mesmo bloco atua como um fator determinante para o sucesso da estimativa de dificuldade, permitindo que a abordagem produza resultados consistentes quando o contexto pedagógico é preservado.

Normalização	Códigos	RMSE	MAE	Pearson	p-value
Simples	Professores	0.1856	0.1337	-0.1006	0.3192
Simples	Todos	0.2106	0.1339	0.1590	0.1139
Avançada	Professores	0.1942	0.1365	-0.1689	0.0928
Avançada	Todos	0.2002	0.1315	0.0844	0.4036

Tabela 6 – Comparação de desempenho do modelo Levenshtein quando exercícios do mesmo bloco não são mantidos

Normalização	Códigos	RMSE	MAE	Pearson	p-value
Simple	Professores	0.1225	0.0835	0.6463	3.771e-13
Simple	Todos	0.1523	0.0876	0.5219	2.563e-08
Avançada	Professores	0.1185	0.0787	0.6710	2.18e-14
Avançada	Todos	0.1375	0.0832	0.6567	1.181e-13

Tabela 7 – Comparação de desempenho do modelo Levenshtein quando exercícios do mesmo bloco não mantidos

5.5.3 Discussão dos resultados

A discussão a seguir está organizada em quatro pontos principais: (i) a comparação entre o CodeBERT e o método de Levenshtein; (ii) o efeito da normalização simples e avançada nos códigos; (iii) a influência dos códigos usados na comparação sobre a precisão das estimativas; e (iv) a importância de ter uma base histórica com quantidade e variedade suficientes de dados para que o método consiga funcionar bem em novos exercícios.

5.5.3.1 CodeBERT versus Levenshtein

A análise comparativa entre as abordagens permite identificar em quais situações o método proposto apresenta melhor desempenho. De forma geral quando os exercícios do mesmo bloco não são considerados, tanto o CodeBERT quanto a distância de Levenshtein apresentam baixo desempenho, com correlações fracas e, na maioria dos casos, sem significância estatística. Isso indica que, para os 200 exercícios analisados, o método não conseguiu encontrar, na base de dados, exercícios previamente resolvidos que fossem suficientemente similares para permitir uma boa estimativa de dificuldade.

Por outro lado, quando a comparação é feita entre exercícios de um mesmo bloco, os resultados melhoram de forma significativa. Nesse cenário, as correlações passam a ser moderadas ou fortes, e os erros diminuem consideravelmente. O Levenshtein não apenas iguala o CodeBERT como o supera em todas as configurações, alcançando correlação de 0,671 contra 0,588 na melhor configuração de cada abordagem. Esse resultado aparentemente contraintuitivo tem uma explicação plausível: dentro de um

mesmo bloco, os exercícios são deliberadamente construídos para serem estruturalmente muito próximos entre si, o que faz com que a similaridade textual capturada pelo Levenshtein seja um sinal especialmente forte e preciso.

A comparação entre as duas abordagens também traz alguns insights. O CodeBERT tende a apresentar um comportamento mais estável em cenários heterogêneos, enquanto a distância de Levenshtein depende mais fortemente de similaridades estruturais explícitas. Ainda assim, ambas as abordagens seguem o mesmo padrão geral: funcionam bem quando há exercícios similares e falham quando essa similaridade não está presente.

5.5.3.2 Normalização avançada versus simples

Em quase todas as configurações testadas, a normalização avançada produziu resultados iguais ou superiores aos da normalização simples. Em outras palavras, o que torna dois exercícios similares em termos de dificuldade não é a semelhança superficial entre os códigos que os resolvem, mas sim a semelhança entre as estruturas lógicas e algorítmicas que esses códigos implementam. A normalização avançada captura essa dimensão de forma mais eficaz, ao reduzir o código à sua essência estrutural.

5.5.3.3 Impacto do conjunto de comparação

. A ampliação do conjunto de comparação para incluir os códigos submetidos pelos alunos e os códigos-solução produziu efeitos distintos para cada abordagem. Para o CodeBERT, a ampliação tendeu a melhorar os resultados no cenário com blocos, sugerindo que a diversidade de estilos de implementação dos alunos enriquece o espaço de busca semântica. Para o Levenshtein, o efeito foi inverso: a melhor configuração utilizou apenas os códigos-solução dos professores, mais padronizados e consistentes. Isso reforça a interpretação de que o Levenshtein é mais sensível ao ruído introduzido pela variabilidade estilística dos códigos de alunos, ao passo que o CodeBERT é mais robusto a essa heterogeneidade.

5.5.3.4 Cobertura da base como fator determinante

Dessa forma, a principal conclusão deste trabalho é que o método depende diretamente da existência de exercícios suficientemente similares na base de dados. Quando essa condição é atendida, o método é capaz de produzir boas estimativas de dificuldade. Caso contrário, seu desempenho é comprometido. Isso mostra que, para aplicação prática, a qualidade e a cobertura da base de exercícios são fatores essenciais para o sucesso da abordagem.

6

CONSIDERAÇÕES FINAIS

Este trabalho teve como objetivo investigar o uso de códigos-solução para estimar a dificuldade de exercícios de programação em sistemas juízes online. A proposta baseia-se na ideia de que exercícios com soluções semelhantes tendem a apresentar níveis de dificuldade semelhantes, permitindo inferir a dificuldade de um novo exercício a partir de exercícios já resolvidos. Essa hipótese foi avaliada sobre um *dataset* com 958 exercícios e mais de 42 mil submissões, utilizando duas métricas de similaridade, a distância de Levenshtein e o modelo CodeBERT, combinadas com duas estratégias de normalização de código.

Os resultados mostraram que o método proposto funciona bem quando há exercícios suficientemente similares disponíveis na base de dados. Exercícios com soluções parecidas com as dos novos exercícios. Nesse cenário, os dois métodos apresentaram resultados bons e estatisticamente significativos. O destaque foi o método de Levenshtein com normalização avançada e usando os códigos dos professores, que apresentou a melhor correlação, com $\rho = 0,671$. Além disso, apresentou um erro médio de 0,079 (MAE) e um erro de 0,119 (RMSE), indicando que suas estimativas ficaram próximas dos valores reais.

Por outro lado, quando não existem exercícios semelhantes na base de dados, o desempenho do método diminui bastante. Isso aconteceu nos testes em que os exercícios do mesmo bloco foram retirados, resultando em correlações fracas e, em muitos casos, sem resultados significativos. Isso mostra que apenas comparar os códigos não é suficiente para estimar a dificuldade de um exercício quando não existem exemplos

realmente parecidos. Portanto, ter uma base de dados com exercícios semelhantes e em quantidade suficiente é mais importante para o sucesso do método do que escolher uma determinada métrica de similaridade ou forma de normalização.

Além disso, a comparação entre as abordagens indicou que o CodeBERT apresenta maior capacidade de generalização em relação à distância de Levenshtein, por considerar aspectos semânticos do código. Ainda assim, ambas as abordagens seguem a mesma limitação fundamental: a dependência da existência de exercícios similares na base.

Do ponto de vista prático, o método proposto apresenta uma vantagem relevante em relação às abordagens anteriores: não requer a extração explícita de métricas de código nem o treinamento de classificadores supervisionados, dependendo apenas do código-solução fornecido pelo professor no momento do cadastro do exercício. Isso facilita sua integração a sistemas existentes, onde poderia auxiliar professores na composição de listas de exercícios e avaliações com níveis de dificuldade adequados ao perfil de suas turmas.

6.1 Limitações

Uma das principais limitações deste trabalho está relacionada à dependência do método em relação à base de dados disponível. Como observado nos experimentos, a ausência de exercícios suficientemente similares compromete diretamente o desempenho do modelo, limitando sua aplicação em cenários onde a base não possui boa cobertura de problemas.

Outra limitação diz respeito ao tamanho do subconjunto utilizado na avaliação experimental (200 exercícios). Embora esse número seja adequado para experimentação, ele pode não representar toda a diversidade de exercícios existentes na base completa, o que pode impactar a generalização dos resultados.

Além disso, o trabalho considera apenas a taxa de acerto como métrica de dificuldade. Embora essa seja uma medida simples e amplamente utilizada, ela não captura outros aspectos importantes do esforço necessário para resolver um exercício,

como tempo de resolução ou número de tentativas.

Por fim, as abordagens utilizadas não incorporam diretamente informações do enunciado dos exercícios, considerando apenas os códigos-solução. Isso pode limitar a capacidade do método em diferenciar exercícios que possuem soluções semelhantes, mas exigem níveis diferentes de compreensão do problema.

6.2 Trabalhos Futuros

Como continuidade deste trabalho, uma linha importante de pesquisa é explorar estratégias para lidar com a ausência de exercícios similares na base. Isso pode incluir o uso de técnicas de agrupamento (clustering) para organizar os exercícios por similaridade ou o desenvolvimento de mecanismos híbridos que combinem diferentes fontes de informação, como código e enunciado.

Outra possibilidade é investigar o uso de modelos mais avançados de representação de código, incluindo versões mais recentes de modelos baseados em transformers ou abordagens que considerem múltiplas linguagens de programação.

Também é interessante ampliar o conjunto de métricas de dificuldade utilizadas, incorporando informações como número médio de submissões, tempo de resolução e tamanho dos códigos, de forma a obter uma estimativa mais completa da dificuldade dos exercícios.

Além disso, recomenda-se avaliar o método em bases de dados maiores e mais diversas, bem como em diferentes contextos educacionais, para verificar sua robustez e aplicabilidade em cenários reais.

Por fim, uma direção promissora é a integração do método ao próprio sistema CodeBench, permitindo sua utilização prática no apoio à criação de listas de exercícios e avaliações, auxiliando professores na escolha de questões com níveis de dificuldade adequados.

6.3 Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001. Este trabalho foi parcialmente financiado pela Fundação de Amparo à Pesquisa do Estado do Amazonas – FAPEAM – por meio do projeto POSGRAD.

REFERÊNCIAS

- ALI, S. K.; AYDAM, Z. M.; RASHED, B. M. Similarity metrics for classification: A review. *IOP Conference Series: Materials Science and Engineering*, IOP Publishing, v. 928, n. 3, p. 032052, nov 2020. Disponível em: <<https://dx.doi.org/10.1088/1757-899X/928/3/032052>>. 32
- ALVAREZ, A.; SCOTT, T. A. Using student surveys in determining the difficulty of programming assignments. *Journal of Computing Sciences in Colleges*, Consortium for Computing Sciences in Colleges, v. 26, n. 2, p. 157–163, 2010. 48
- ALVES, A. et al. Análise comportamental em juízes online para predição do desempenho final de alunos em disciplinas de computação. In: *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*. [S.l.: s.n.], 2019. v. 30, n. 1, p. 1906. 30
- ALVES, F. P.; JAQUES, P. Um ambiente virtual com feedback personalizado para apoio a disciplinas de programação. In: *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*. [S.l.: s.n.], 2014. v. 3, n. 1, p. 51. 18, 25
- BACKURS, A.; INDYK, P. Edit distance cannot be computed in strongly subquadratic time (unless seth is false). In: *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. [S.l.: s.n.], 2015. p. 51–58. 34
- BASTIAN, M.; MÜHLING, A. Comparing estimates of difficulty of programming constructs. In: *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research*. [S.l.: s.n.], 2022. p. 1–12. 48
- BECKMANN, J. F.; BIRNEY, D. P.; GOODE, N. Beyond psychometrics: the difference between difficult problem solving and complex problem solving. *Frontiers in psychology*, Frontiers Media SA, v. 8, p. 1739, 2017. 44
- CAMPOS, C. P. de; FERREIRA, C. E. Boca: um sistema de apoio a competições de programação. In: SOCIEDADE BRASILEIRA DE COMPUTAÇÃO. *Workshop de Educação em Computação*. [S.l.], 2004. 27
- CARVALHO, L. S. et al. Ensino de programação para futuros não-programadores: contextualizando os exercícios com as demais disciplinas de mesmo período letivo. In: SBC. *Anais do XXIV Workshop sobre Educação em Computação*. [S.l.], 2016. p. 2116–2125. 18, 29, 30

- CROSBY, R. W.; VALENTINE, S.; WILLIAMS, T. L. Leveraging programming difficulty to improve understanding and perceptions of non-majors. *Journal of Computing Sciences in Colleges*, Consortium for Computing Sciences in Colleges, v. 29, n. 4, p. 27–35, 2014. [48](#)
- DERUS, S.; ALI, A. M. Difficulties in learning programming: Views of students. In: *1st International Conference on Current Issues in Education (ICCIE 2012)*. [S.l.: s.n.], 2012. p. 74–79. [50](#)
- EFFENBERGER, T.; CECHÁK, J.; PELÁNEK, R. Difficulty and complexity of introductory programming problems. *Educational Data Mining in Computer Science Education (CSEDM)*, 2019. [45](#), [46](#), [53](#)
- ELNAFFAR, S. Using software metrics to predict the difficulty of code writing questions. In: IEEE. *2016 ieee global engineering education conference (educon)*. [S.l.], 2016. p. 513–518. [47](#), [51](#)
- FENG, Z. et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020. [36](#)
- FERNANDES, J. et al. Correlação entre complexidade e dificuldade de questões de programação em juízes online. In: SBC. *Anais do III Simpósio Brasileiro de Educação em Computação*. [S.l.], 2023. p. 97–107. [18](#), [45](#), [46](#)
- FERREIRA, J. C.; PATINO, C. M. What does the p value really mean? *Jornal Brasileiro de Pneumologia*, v. 41, n. 5, p. 485, 2015. [40](#)
- FILHO, D. B. F.; JÚNIOR, J. A. S. Desvendando os mistérios do coeficiente de correlação de pearson (r). *Revista política hoje*, Recife, v. 18, n. 1, p. 115–146, 2009. [40](#)
- FRANCISCO, R.; JÚNIOR, C. P.; AMBRÓSIO, A. P. Juiz online no ensino de programação introdutória-uma revisão sistemática da literatura. In: *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*. [S.l.: s.n.], 2016. v. 27, n. 1, p. 11. [17](#), [18](#)
- FRANCISCO, R. E.; AMBROSIO, A. P. Mining an online judge system to support introductory computer programming teaching. In: CITESEER. *EDM (Workshops)*. [S.l.], 2015. [17](#), [25](#), [48](#)
- FRANCISCO, R. E. et al. Juiz online no ensino de cs1-lições aprendidas e proposta de uma ferramenta. *Revista Brasileira de Informática na Educação*, v. 26, n. 03, p. 163, 2018. [26](#)
- GALVÃO, L.; FERNANDES, D.; GADELHA, B. 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)*. [S.l.: s.n.], 2016. v. 27, n. 1, p. 140. [18](#)
- GOMES, A. et al. Aprendizagem de programação de computadores: dificuldades e ferramentas de suporte. *Revista portuguesa de pedagogia*, p. 161–179, 2008. [18](#)
- GUO, D. et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020. [37](#)

- IHANTOLA, P.; SORVA, J.; VIHAVAINEN, A. Automatically detectable indicators of programming assignment difficulty. In: *Proceedings of the 15th Annual Conference on Information technology education*. [S.l.: s.n.], 2014. p. 33–38. [31](#), [48](#)
- ISHTIAQ, A. A. et al. Bert2code: Can pretrained language models be leveraged for code search? *arXiv preprint arXiv:2104.08017*, 2021. [36](#)
- JULIO, M. R. F. M. *Um estudo de metricas de similaridade em sistemas baseados em casos aplicados a area da saude*. Tese (Doutorado) — [sn], 2005. [32](#)
- JÚNIOR, H. B. de F. et al. Recomendação automática de problemas em juízes online usando processamento de linguagem natural e análise dirigida aos dados. In: SBC. *Anais do XXXI simpósio brasileiro de informática na educação*. [S.l.], 2020. p. 1152–1161. [19](#)
- KASTO, N.; WHALLEY, J. Measuring the difficulty of code comprehension tasks using software metrics. In: *Proceedings of the Fifteenth Australasian Computing Education Conference-Volume 136*. [S.l.: s.n.], 2013. p. 59–65. [49](#)
- KIRSCHNER, P. A. et al. From cognitive load theory to collaborative cognitive load theory. *International journal of computer-supported collaborative learning*, Springer, v. 13, p. 213–233, 2018. [31](#)
- LIMA, M. et al. Classificação de dificuldade de questões de programação com base em métricas de código. In: SBC. *Anais do XXXI Simpósio Brasileiro de Informática na Educação*. [S.l.], 2020. p. 1323–1332. [19](#), [20](#), [49](#)
- LIMA, M. A. et al. Uso de atributos de código para classificar a dificuldade de questões de programação em juízes online. *Revista Brasileira de Informática na Educação*, v. 29, p. 1137–1157, 2021. [49](#)
- MEISALO, V.; SUTINEN, E.; TORVINEN, S. Classification of exercises in a virtual programming course. In: IEEE. *34th Annual Frontiers in Education, 2004. FIE 2004*. [S.l.], 2004. p. S3D–1. [50](#), [51](#)
- MOREIRA, M. P.; FAVERO, E. L. Um ambiente para ensino de programação com feedback automático de exercícios. In: *Workshop sobre Educação em Computação (WEI 2009)*. [S.l.: s.n.], 2009. v. 17. [17](#)
- NASCIMENTO, R. L. S. do; JUNIOR, G. G. da C.; FAGUNDES, R. A. de A. Mineração de dados educacionais: Um estudo sobre indicadores da educação em bases de dados do inep. *RENTE*, v. 16, n. 1, 2018. [41](#)
- PAES, R. de B. et al. Ferramenta para a avaliação de aprendizado de alunos em programação de computadores. In: *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*. [S.l.: s.n.], 2013. v. 2, n. 1. [27](#)
- PARANHOS, R. et al. Desvendando os mistérios do coeficiente de correlação de pearson: o retorno. *Leviathan (São Paulo)*, n. 8, p. 66–95, 2014. [39](#)
- PRIOR, J. Online assessment of sql query formulation skills. In: AUSTRALIAN COMPUTER SOCIETY INC. *Australasian Computing Education Conference*. [S.l.], 2003. [17](#)

- SANTANA, D. R. de; RIBEIRO, L. A. Junções por similaridade em espaços vetoriais semânticos. In: SBC. *Simpósio Brasileiro de Banco de Dados (SBBDD)*. [S.l.], 2022. p. 147–153. [37](#)
- SANTOS, P. dos et al. Classificação de dificuldade de questões de programação com base na inteligibilidade do enunciado. In: *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*. [S.l.: s.n.], 2019. v. 30, n. 1, p. 1886. [20](#)
- SCARTON, C. E.; ALUÍSIO, S. M. Análise da inteligibilidade de textos via ferramentas de processamento de língua natural: adaptando as métricas do coh-metrix para o português. *Linguamática*, v. 2, n. 1, p. 45–61, 2010. [20](#)
- SILVA, E. S. et al. Previsão de indicadores de dificuldade de questões de programação a partir de métricas do código de solução. In: SBC. *Anais do XXXIII Simpósio Brasileiro de Informática na Educação*. [S.l.], 2022. p. 859–870. [25](#), [47](#), [51](#)
- SILVA, F. X. da; MENEZES, C. S. de; JUNIOR, A. N. de C. Ensino introdutório de programação: Um estudo rumo ao uso das arquiteturas pedagógicas. In: SBC. *Anais do XXXIV Simpósio Brasileiro de Informática na Educação*. [S.l.], 2023. p. 428–438. [17](#)
- SILVA, M. R. D. da et al. Xxv simpósio brasileiro de recursos hídricos avaliação da técnica de regionalização estatística bccaq aplicado ao modelo can-esm5 do cmip6 para o nordeste. [41](#)
- SOUZA, D. M.; BATISTA, M. H. da S.; BARBOSA, E. F. Problemas e dificuldades no ensino de programação: Um mapeamento sistemático. *Revista Brasileira de Informática na Educação*, v. 24, n. 1, p. 39, 2016. [18](#)
- VIER, J.; GLUZ, J.; JAQUES, P. A. Empregando redes bayesianas para modelar automaticamente o conhecimento dos alunos em lógica de programação. *Revista Brasileira de Informática na Educação*, v. 23, n. 02, p. 45, 2015. [26](#)
- XIAO, Y. et al. Promises and perils of using transformer-based models for se research. *Neural Networks*, Elsevier, v. 184, p. 107067, 2025. [37](#)
- ZAFFALON, F. et al. Um estudo comparativo entre dois modelos que estimam a habilidade dos estudantes: Elo e teoria de resposta ao item. In: *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*. [S.l.: s.n.], 2019. v. 30, n. 1, p. 459. [48](#)
- ZAKERI-NASRABADI, M. et al. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *Journal of Systems and Software*, Elsevier, p. 111796, 2023. [33](#)
- ZHOU, X.; HAN, D.; LO, D. Assessing generalizability of codebert. In: IEEE. *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2021. p. 425–436. [37](#)