Revista Brasileira de Informática na Educação – RBIE
Brazilian Journal of Computers in Education
(ISSN online: 2317-6121; print: 1414-5685)
https://sol.sbc.org.br/journals/index.php/rbie

Submission: 31/Jul/2023; 1st round notif.: 30/Oct/2023; New version: 28/Nov/2023; 2nd round notif.: 18/Dec/2023;
Camera ready: 02/Jan/2024; Edition review: 16/Jan/2024; Available online: 16/Jan/2024; Published: 16/Jan/2024;

Complexidade versus dificuldade: Uma análise da sua correlação em questões de programação em juízes on-line

Title: Complexity versus difficulty: An analysis of their correlation in programming questions in online judges

Título: Complejidad versus dificultad: Un análisis de su correlación en preguntas de programación en jueces en línea

Jackson Celestino Fernandes
Instituto de Computação
Universidade Federal do Amazonas
ORCID: 0009-0002-1222-9729
jackson.fernandes@icomp.ufam.edu.br

Leandro Silva Galvão de Carvalho
Instituto de Computação
Universidade Federal do Amazonas
ORCID: 0000-0003-2970-2084
galvao@icomp.ufam.edu.br

David Braga Fernandes de Oliveira
Instituto de Computação
Universidade Federal do Amazonas
ORCID: 0000-0002-2887-2324
david@icomp.ufam.edu.br

Elaine Harada Teixeira de Oliveira
Instituto de Computação
Universidade Federal do Amazonas
ORCID: 0000-0003-2884-9359
elaine@icomp.ufam.edu.br

Filipe Dwan Pereira
Departamento de Ciência da Computação
Universidade Federal de Roraima
ORCID: 0000-0003-4914-3347
filipe.dwan@ufrr.br

Tanara Lauschner
Instituto de Computação
Universidade Federal do Amazonas
ORCID: 0000-0001-7104-9432
tanara@icomp.ufam.edu.br

Resumo

Ambientes de correção automática de código são cada vez mais usados no processo de ensino-aprendizagem de disciplinas de programação. Porém, um problema frequentemente enfrentado pelos professores que usam tais ambientes é determinar a dificuldade das questões cadastradas. Este trabalho tem como objetivo realizar uma análise de correlação entre métricas de complexidade de código e a dificuldade enfrentada pelos alunos, de maneira que seja possível prever automaticamente o nível de dificuldade de uma questão apenas conhecendo seu modelo de solução. Este estudo foi dividido em três etapas: i) análise da correlação de Spearman *entre métricas de complexidade (extraídas da questão) e de dificuldade (extraídas da interação do aluno com a questão); ii) predição da classe de dificuldade de questões por meio de modelos de aprendizado de máquina para classificação; e iii) predição de métricas de dificuldade usando modelos de regressão. Quanto ao item i), observou-se que 96% das correlações foram fracas ou inexistentes entre métricas individuais de complexidade de código e de dificuldade, 4% de casos de correlação moderada e nenhum caso de correlação forte. Para o item ii), o maior* f1-score *obtido foi de 88%, considerando classificação com dois níveis de dificuldade ("fácil" e "difícil"), e* f1-score *máximo de 67%, considerando classificação com três níveis ("fácil", "médio" e "difícil"). Para o item iii), o melhor resultado obtido foi um coeficiente de determinação ajustado de 63%.*
Palavras-chave: *Juízes on-line; Dificuldade de questões; Métricas de dificuldade; Métricas de complexidade; Aprendizado de máquina.*

Abstract

Automatic code correction environments are increasingly used in the teaching-learning process of programming disciplines. However, a problem often faced by teachers who use these systems is to determine the difficulty of the

Cite as: Fernandes, J. C., Carvalho, L. S. G., Oliveira, D. B. F., Oliveira, E. H. T., Pereira, F. D. & Lauschner, T. (2024). Complexidade versus dificuldade: Uma análise da sua correlação em questões de programação em juízes on-line. Revista Brasileira de Informática na Educação, 32, 22-49. https://doi.org/10.5753/rbie.2024.3587.

questions registered in the environment. This work aims to carry out a correlation analysis between code complexity metrics and the difficulty faced by students, so that it is possible to automatically predict the difficulty level of a question just by knowing its solution model. This study was divided into three stages: i) analysis of Spearman's *correlation between complexity metrics (extracted from the question) and difficulty (extracted from the student's interaction with the question), ii) prediction of the difficulty class of questions through models machine learning for classification and iii) prediction of difficulty metrics using regression models. Regarding item i), it was observed that 96% of the correlations were weak or non-existent between individual metrics of code complexity and difficulty, 4% of cases of moderate correlation and no cases of strong correlation. For item ii), the highest f1-score obtained was 88%, considering classification with two levels of difficulty ("easy" and "hard"), and a maximum f1-score of 67%, considering classification with three levels ("easy", "medium" and "hard"). For item iii), the best result obtained was an adjusted correlation coefficient of 63%.*
Keywords: *Online Judges; Questions difficulty; Difficulty metrics; Complexity metrics; Machine Learning.*

Resumen
Ambientes de corrección automática de código son cada vez más utilizados en el proceso de enseñanza-aprendizaje de disciplinas de programación. Sin embargo, un problema enfrentado frecuentemente por los profesores que utilizan tales sistemas es determinar la dificultad de las preguntas registradas en el entorno. Este trabajo tiene como objetivo realizar un análisis de correlación entre métricas de complejidad de código y la dificultad enfrentada por los alumnos, de manera que sea posible predecir automáticamente el nivel de dificultad de una pregunta solo conociendo su modelo de solución. Este estudio se dividió en tres etapas: i) análisis de correlación de Spearman entre métricas de complejidad (extraídas de la pregunta) y de dificultad (extraídas de la interacción del alumno con la pregunta), ii) predicción de la clase de dificultad de preguntas mediante modelos de aprendizaje automático para clasificación, y iii) predicción de métricas de dificultad usando modelos de regresión. En cuanto al ítem i), se observó que el 96% de las correlaciones fueron débiles o inexistentes entre métricas individuales de complejidad de código y de dificultad, 4% de casos de correlación moderada y ningún caso de correlación fuerte. Para el ítem ii), el mayor f1-score obtenido fue del 88%, considerando la clasificación con dos niveles de dificultad ("fácilz "difícil"), y el f1-score máximo de 67%, considerando la clasificación con tres niveles ("fácil", "medioz "difícil"). Para el ítem iii), el mejor resultado obtenido fue un coeficiente de determinación ajustado del 63%.
Palabras clave: *Juízes en línea; Dificultad de preguntas; Métricas de dificultad; Métricas de complejidad; Aprendizaje automático.*

1 Introdução

Ambientes de correção automática de códigos (ACAC), também conhecidos como Juízes *Online* (JO), são ferramentas muito usadas como apoio pedagógico em disciplinas de programação (Carvalho et al., 2016). A principal vantagem em utilizar esses sistemas no processo de ensino e aprendizagem é o rápido *feedback* fornecido por eles, permitindo assim que os alunos resolvam mais exercícios e corrijam seus erros rapidamente, já que o *feedback* dos JOs é instantâneo (Francisco et al., 2018; Paes et al., 2013; Pelz et al., 2012).

Devido à possibilidade de criação de diferentes tipos de atividades, é comum que ambientes com JO também sejam utilizados nas atividades avaliativas da disciplina. Em uma avaliação com JO, o professor seleciona do banco uma lista de questões que os alunos precisam resolver dentro do espaço de tempo definido para a prova. Geralmente, as questões são embaralhadas ou sorteadas, a fim de dificultar a cola entre alunos próximos. No caso do sorteio de questões, para garantir que nenhum aluno tenha uma prova (ou lista de exercícios) mais fácil ou mais difícil que outros, o professor deve garantir que o nível de dificuldade das listagens de questões sorteadas não seja

desbalanceado. Entretanto, a tendência é existir uma divergência entre a dificuldade estimada pelo professor e a enfrentada pelos alunos (Francisco et al., 2018; Meisalo et al., 2004; Prisco et al., 2017).

Para contornar o problema da subjetividade, diversos estudos têm sido conduzidos na tentativa de estimar a dificuldade de uma questão, fruto do esforço dos estudantes em resolver essa questão. A hipótese defendida é que atributos extraídos a partir das questões, chamados de *métricas de complexidade* (Effenberger et al., 2019; Liu & Li, 2012), influenciam no desempenho obtido pelos alunos ao tentarem resolver uma questão de escrita de código de programação e que, portanto, podem ser usados como indicadores de sua dificuldade.

Baseado na ideia descrita acima, este trabalho tem como objetivo verificar como métricas de complexidade de código se correlacionam com métricas de dificuldade sob diversas perspectivas. De maneira geral, tais perspectivas podem ser condensadas por meio das seguintes questões de pesquisa:

QP1: A dificuldade de uma questão pode ser estimada por meio de algum atributo de complexidade de código?

QP2: É possível estimar a classe de dificuldade de uma questão por meio de um conjunto de atributos de complexidade de código de seu modelo de solução?

QP3: É possível estimar, com boa precisão, métricas de desempenho dos estudantes em uma dada questão a partir de um conjunto de atributos referentes à complexidade de código de seu modelo de solução?

A partir da definição das questões de pesquisa, foram estabelecidos três estudos:

1. Análise da correlação entre métricas individuais de complexidade de código e métricas de dificuldade;
2. Predição da classe de dificuldade de uma questão, a partir da extração de atributos de código, utilizando técnicas de aprendizado de máquina para classificação em dois e três níveis de dificuldade; e
3. Estimativas de desempenho dos alunos para uma questão a partir de métricas de complexidade de código, usando modelos de regressão.

Este artigo está organizado da seguinte forma: a Seção 2 discorre sobre juízes on-line e suas aplicações, apresenta as definições de complexidade e dificuldade, além de uma breve revisão exploratória da literatura sobre os esforços já feitos para tentar resolver esse problema ou similares; a Seção 3 descreve o ambiente utilizado, a base de questões coletada, bem como a descrição das variáveis utilizadas e a metodologia aplicada para treinamento dos preditores; a Seção 4 apresenta os resultados obtidos para o estudo realizado sobre correlação entre variáveis dependentes, entre variáveis dependentes e independentes, além dos experimentos de predição com classificadores e regressores; na Seção 5, é feita uma análise em cima dos resultados obtidos; e por fim, a Seção 6 conclui o trabalho com algumas ponderações.

2 Referencial teórico

Nesta seção, é aprofundado o uso de juízes on-line no contexto educacional, os conceitos de complexidade e dificuldade, além de ser realizada uma revisão exploratória da literatura, através da técnica *Snowballing*, sobre as estratégias de resolução já empregadas para o tema abordado.

2.1 Juízes on-line

Um juiz on-line é um sistema que realiza correção automática de códigos-fontes (Carvalho et al., 2016; Francisco et al., 2018). Geralmente, um JO conta com um banco de questões cadastradas no ambiente, as quais podem ser resolvidas em determinadas linguagens de programação. Cada questão é composta por um enunciado, que descreve o problema a ser resolvido, e um conjunto de casos de teste (entrada e saída), que valida se a solução está correta ou não. Quando uma solução é submetida ao JO, o programa gerado pelo código-fonte enviado é executado para todos os casos de teste cadastrados para a questão e comparado com as suas respectivas saídas esperadas. Caso a saída gerada pelo programa seja diferente da saída cadastrada para algum caso de teste em questão, ou o programa gerou algum erro de execução, o JO informa o erro para o usuário; caso contrário, é informado que o programa está correto. Os casos de teste usados para avaliação são ocultos aos estudantes. Alguns casos de teste podem ser mostrados publicamente, para servir de exemplo durante a resolução da questão.

Um dos casos de uso mais comuns de JO é em competições de programação. Nesse contexto, é proposto um conjunto pequeno de problemas que devem ser resolvidos em um espaço de tempo limitado. Cada solução para um problema passa por um exaustivo processo de validação, por meio da execução de diversos casos de testes. Além disso, o programa gerado possui um tempo limite de execução, de modo que programas que ultrapassem esse tempo são invalidados. O principal objetivo é validar se a complexidade assintótica do algoritmo implementado está dentro do esperado para uma solução válida. Todo esse processo de validação é feito por um JO, o qual é capaz de retornar a resposta em poucos segundos. Em geral, as respostas dos JOs contêm uma simples mensagem descrevendo o resultado da submissão, sendo as mais comuns: *accepted*, *wrong answer*, *time limit exceeded*, *runtime error* e *compilation error*.

Apesar dos benefícios que um JO traz com respeito à eficiência de correção de códigos-fonte, Francisco et al. (2018) notaram que não era adequado utilizar JO em um ambiente de ensino da mesma maneira que se usa em competições de programação. Um dos principais motivos é que o *feedback* sucinto dado pelo JO, em geral, é insuficiente para o aluno conseguir identificar e corrigir o erro. Além disso, também foram identificados problemas de usabilidade em alguns JOs, o que poderia prejudicar a experiência dos estudantes durante suas primeiras interações com o ambiente. Por fim, foi notado também que um sistema de *ranking* semelhante ao de uma competição de programação pode não ser adequado, pois pode desestimular alunos com mais dificuldades.

Pensando em uma adaptação dos JOs para uso em um contexto educacional, Francisco et al. (2018), através de estudos dos pontos positivos e negativos de diversos JOs, sintetizaram os principais requisitos funcionais e não-funcionais que podem ser atendidos para a construção de um JO, sendo a escolha deles dependente do contexto de ensino. Os principais requisitos funcionais destacados foram:

- ***Feedback*:** requisito indispensável, um JO com um bom *feedback* permite que o aluno prossiga no seu ritmo, e promove a autoaprendizagem ao trazer aspectos da sintaxe, estrutura, semântica e resultados de testes no *feedback* dado (Llana et al., 2012; Wang et al., 2011).
- **Integração do sistema com os cursos:** alguns JO permitem gerenciar cursos, adicionar exercícios, interagir com os alunos, criar trabalhos, criar exames, etc. (Petit et al., 2012).
- **Análise do desempenho geral dos estudantes:** permite que o professor consiga acompanhar o progresso de seus alunos, de maneira que seja possível identificar possíveis dificuldades de entendimento do assunto, observando os dados de resolução das questões.
- **Possibilidade de diferentes tipos de atividades:** além de lista de exercícios, o JO também pode ter a opção de criar outros tipos de atividade, como avaliações e tarefas de casa.

Já os principais requisitos não-funcionais identificados foram:

- **Usabilidade:** é importante que os alunos não tenham dificuldades em usar o sistema, pois isso facilita o processo de memorização por parte do usuário (Llana et al., 2012).
- **Escalabilidade:** mesmo se o sistema tiver uma crescente de usuários, ele ainda deve ser capaz de responder rapidamente, pois caso contrário, perde-se o conceito de *feedback* imediato (Petit et al., 2012).
- **Disponibilidade:** permite que os alunos possam utilizar o ambiente em qualquer lugar e momento. Um adicional seria disponibilizar o sistema para uso em *Massive Open Online Courses* (MOOCs) (Vihavainen et al., 2013).

Além disso, os autores também identificaram a necessidade de classificar as questões quanto aos tópicos abordados e à dificuldade, a fim de permitir que o sistema fosse capaz de gerar listas de exercícios com níveis de dificuldade semelhantes, com base em parâmetros definidos pelo professor.

2.2 Diferença entre complexidade e dificuldade

Definir de maneira clara e objetiva a diferença entre a complexidade e a dificuldade de uma tarefa é objeto de vários estudos. Apesar de haver similaridade entre os dois conceitos, eles não são nem independentes e nem equivalentes (Liu & Li, 2012). Existem diversos pontos de vista sobre a diferença entre os dois conceitos. Na visão de Rouse e SH (1979), a complexidade é um sub-conceito de dificuldade, pois uma tarefa difícil não necessariamente é complexa, mas uma tarefa complexa, em geral, tende a ser difícil. Já na visão de Bonner (1994), a dificuldade é um sub-elemento da complexidade, pois a complexidade de uma tarefa consiste em duas dimensões: quantidade de informação e estrutura da tarefa. Por outro lado, Robinson (2001) define como conceitos distintos, já que a dificuldade diz respeito ao esforço cognitivo demandado em uma tarefa, enquanto a complexidade se refere à quantidade de recursos trazidos para resolvê-la.

Na definição adotada por Effenberger et al. (2019), a *complexidade* diz respeito a características intrínsecas da tarefa que influenciam o desempenho, mas são independentes do contexto

de aplicação, tais como pessoas resolvendo um problema. Já a dificuldade diz respeito ao desempenho observado, tais como taxa de acerto, tempo médio de resolução ou número de tentativas, o qual é totalmente dependente do contexto de aplicação.

No contexto de programação, a complexidade de um problema pode ser vista em termos de métricas de código, tais como *complexidade ciclomática* (McCabe, 1976), *número de operadores*, e *quantidade de linhas de código*. Já a dificuldade pode ser vista como o nível de esforço e habilidade necessários para que o estudante construa um código logicamente correto, ou seja, que resolva o problema proposto.

Problemas que possuem como solução códigos mais complexos possivelmente demandam mais habilidade ou esforço cognitivo dos alunos para serem resolvidos e, portanto, tendem a ser mais difíceis. No entanto, a recíproca não necessariamente é verdadeira, pois, para alguns problemas de programação, o esforço ou habilidade exigido para se chegar a uma solução pode ser altíssimo, mas o código construído pode acabar sendo simples, como, por exemplo, em problemas com formulações recursivas, ou quando é necessário derivar uma fórmula matemática (Elnaffar, 2016). Por isso, este trabalho limitou-se a analisar a correlação entre complexidade e dificuldade apenas para questões de programação introdutória (CS1), uma vez que os tópicos abordados em geral limitam-se somente aos conceitos básicos, como fluxos de controle e listas.

2.3 Trabalhos relacionados

O problema de estabelecer uma relação entre complexidade e dificuldade em questões de programação é objeto de vários estudos. Effenberger et al. (2019) exploram essa relação por meio de uma análise da correlação de *Spearman*, conduzida através da proposição de 4 conjuntos de problemas de programação introdutória, sendo um desses conjuntos programação em *Python*, com 73 exercícios propostos para 2.000 estudantes. Apesar de ser inconclusivo para casos mais gerais, foi possível notar que o *número de linhas de código* e o *número de conceitos de programação abordados*, quando usados individualmente, têm correlação fraca com a dificuldade das questões. Portanto, para a construção de bons preditores de dificuldade, é necessário o uso combinado de diversas métricas de complexidade de código. Além disso, os autores observaram uma correlação forte entre complexidade e dificuldade para as questões propostas.

Já Elnaffar (2016) propõe uma métrica única chamada *Predicted Difficulty Index* (PDI), calculada a partir da combinação linear de cinco métricas de complexidade de código igualmente ponderadas. O objetivo do PDI é estimar a dificuldade que os estudantes terão ao resolver a questão, baseado em uma solução de exemplo provida pelo instrutor. A avaliação da métrica proposta foi feita a partir de 10 questões de programação introdutória em linguagem Java, extraídas de uma única turma. Como resultado, viu-se que essa métrica, apesar de não ser adequada para estimar a dificuldade da questão em si, pode ser útil para ordenar as questões baseado nos valores obtidos de PDI. No entanto, apesar de ter gerado bons resultados para questões envolvendo fluxos de controle de programação, a métrica não conseguiu estimar corretamente a dificuldade de questões envolvendo tópicos mais avançados, como recursão, os quais demandam mais esforço cognitivo. Portanto, essa solução limita-se apenas a questões de programação introdutória.

Por outro lado, Santos et al. (2019) propõem um método para classificar a dificuldade de questões de programação com base em métricas de inteligibilidade textual. Uma das métricas utilizadas é o *Flesch Reading Ease*, uma fórmula que avalia superficialmente características do

texto, como número de palavras em uma sentença (Scarton & Aluísio, 2010). Além do índice *Flesch*, também foram extraídas outras 5 métricas de um total de 41 fornecidas pela biblioteca *Coh-Metrix-Port*, uma adaptação do *Coh-Metrix* para o português que utiliza técnicas de processamento de linguagem natural e aprendizagem de máquina para extração de atributos de texto. A análise abrangeu 450 questões, respondidas por 800 alunos em disciplinas de introdução à programação, utilizando as seguintes variáveis como indicador de dificuldade: *taxa de acerto*, *número médio de tentativas*, *número médio de submissões* e *tempo de solução*. Como resultado, obteve-se uma acurácia de 75%. A grande limitação desse método foi na classificação de questões com enunciados simples, uma vez que não foi encontrada correlação com a classe de dificuldade de uma questão.

No trabalho de Lima et al. (2021), foi desenvolvido um preditor capaz de classificar de forma dicotômica tanto a facilidade quanto a dificuldade de uma questão a partir de métricas de código extraídas de seu modelo de solução. Nessa pesquisa, a base usada para treino e teste foi composta por 404 questões aplicadas em avaliações presenciais de uma disciplina de introdução à programação, cadastradas no JO CodeBench entre 2017 e 2019. A taxa de acerto foi usada como indicador de dificuldade, enquanto que para as métricas de código, foram usados 92 atributos, extraídos por meio das bibliotecas *Radon* e *Tokenize* da linguagem Python. O treinamento foi feito em duas etapas, utilizando uma técnica conhecida como *stacking*. Como resultado, obteve-se um *f1-score* de 0,84 usando classificação por facilidade (fácil e não-fácil), e 0,82 em classificação por dificuldade (difícil e não-difícil). Além disso, viu-se que atributos como a existência *palavras-chave*, *existência de operadores lógicos* e *existência de laços de repetição* tiveram bastante importância em ambas as classificações. A grande limitação deste trabalho foi a exploração de apenas uma métrica como indicador de dificuldade.

Em Silva et al. (2022), foi desenvolvido um preditor capaz de classificar a dificuldade de uma questão em dois níveis de dificuldade (fácil e difícil). Neste trabalho, foram utilizadas 394 questões de programação em Python, resolvidas por 1.474 estudantes de turmas de uma disciplina de introdução à programação, também usando os dados fornecidos pelo JO CodeBench, por meio de *logs* capturados durante a interação dos alunos com a questão. Como indicador de dificuldade, além da taxa de acerto, foram utilizadas outras 8 métricas que, de certa forma, também medem o esforço necessário do aluno para resolver a questão. Como resultado, obteve-se um *f1-score* de 0,92 e acurácia de 0,85 para a taxa de acerto. Além disso, verificou-se através dos logs, que os alunos tendem a testar mais seus códigos do que a submeterem, pois a correlação entre o número de consultas (número de testes + número de submissões) e número de testes foi quase perfeita, algo que não aconteceu quando se mediu a correlação entre número de consultas e número de submissões. Um dos problemas identificados neste trabalho foi a falta de uma análise mais aprofundada para a taxa de acerto, uma vez que, considerando o método de discretização e a distribuição de frequência para esta métrica, possivelmente gerou classes muito desbalanceadas, o que pode ter comprometido a qualidade do modelo para a classe com menos amostras.

Apesar dos resultados obtidos pelos estudos citados terem sido significativos, algumas questões ficaram em aberto. Em Effenberger et al. (2019), não foi feito um estudo aprofundado de como a combinação de métricas de dificuldade poderia ser feita. Já no estudo de Elnaffar (2016), não foi possível obter um modelo que classificasse a dificuldade das questões, ficando limitado a apenas a ordenação por ordem de dificuldade. Nos estudos de Lima et al. (2021), não foi feita uma análise de desempenho com outras métricas de dificuldade, tais como o número de submissões fei-

tas para uma dada questão, ou o tempo necessário para codificar a solução. Além disso, tanto nos estudos de Lima et al. (2021) quanto nos trabalho de Santos et al. (2019) e Silva et al. (2022), seria importante também verificar o desempenho dos modelos de regressão utilizando outras métricas de dificuldade, pois apesar de Lima et al. (2021) terem obtido resultados ruins, a métrica usada foi apenas a taxa de acerto, havendo a possibilidade de outras métricas de dificuldade poderem gerar bons resultados.

Os trabalhos apresentados acima foram encontrados a partir da técnica *Snowballing* (Wohlin, 2014), tendo como base trabalhos previamente conhecidos. Como trabalho futuro, pensa-se em utilizar técnicas mais sofisticadas, como Revisão Sistemática da Literatura (RSL), para encontrar outros trabalhos relacionados.

3 Métodos

Nesta seção, serão abordados os processos de geração da base de dados, definição e obtenção das variáveis dependentes e independentes, e o processo de treinamento dos modelos preditores[1].

3.1 O sistema CodeBench

O CodeBench[2] é um juiz on-line desenvolvido no Instituto de Computação (IComp) da Universidade Federal do Amazonas (UFAM) por um dos autores deste artigo. É principalmente utilizado como ferramenta de apoio pedagógico nas disciplinas de Introdução à Programação de Computadores (IPC), ministrada para vários cursos de graduação nas áreas de engenharia e ciências exatas da UFAM.

No CodeBench, o professor pode criar uma nova turma, onde os alunos devem se inscrever para terem acesso às atividades da disciplina. As atividades da disciplina tipicamente são listas de exercícios e avaliações, com data de início e término pré-determinadas. Todas as atividades só podem ser visualizadas e respondidas durante o prazo estipulado pelo professor. Para elaborar as atividades, o professor pode selecionar algumas questões já cadastradas no banco ou criar uma nova questão. No caso do professor criar uma nova questão, ela também fica disponível para outros professores, criando um ambiente colaborativo e otimizado. Atualmente, o banco de questões do CodeBench contém mais de 6600 questões cadastradas.

Além disso, o CodeBench também conta com um ambiente de desenvolvimento integrado (IDE), que possibilita os alunos escrever e testar seus códigos dentro do próprio ambiente, sem a necessidade de usar editores instalados localmente em uma máquina. Atualmente, o CodeBench suporta as seguintes linguagens de programação: Assembly (ARM), C, C++, Dart, Java, Python, Haskell, Lua e Prolog. A Figura 1 mostra a interface atual do sistema.

Para fins de pesquisa, o CodeBench conta com um sistema de logs[3], o qual contém registros de todas as ações realizadas pelos estudantes na IDE durante a tentativa de resolução dos exercí-

[1] Os códigos utilizados nos experimentos estão disponíveis em: https://bit.ly/3QKAWg0

[2] Acessível em https://codebench.icomp.ufam.edu.br

[3] O dataset de logs anonimizados está disponibilizado publicamente em: https://codebench.icomp.ufam.edu.br/dataset/

Figura 1: Interface da IDE do CodeBench.

cios. Os logs contêm os dados relacionados ao histórico de execução dos estudantes (submissões e testes), aos códigos executados e a qualquer interação realizada diretamente na IDE, tais como movimentações do cursor e digitações no teclado. Além disso, os logs também contêm dados anonimizados dos alunos cadastrados no ambiente, obtidas por meio de um questionário.

3.2 Geração da base de questões

A base de questões foi extraída a partir dos dados gerados pelo sistema de *logs* do CodeBench para as turmas de IPC da UFAM. Atualmente, o conteúdo da disciplina é dividido em 7 módulos: (1) variáveis e programação sequencial; (2) estruturas condicionais; (3) estruturas condicionais aninhadas; (4) repetição por condição; (5) vetores e strings; (6) repetição por contagem; e (7) matrizes.

As questões selecionadas foram utilizadas em avaliações presenciais, entre os anos letivos de 2017 a 2019, e 2021 (realizado em 2022, devido a um atraso no calendário acadêmico causado pela pandemia de COVID-19). Em uma avaliação presencial, além do tempo reduzido, os alunos são supervisionados pelo professor e pelo tutor da disciplina, na tentativa de reduzir a prática de cola. Como as avaliações das turmas do ano letivo de 2020 foram feitas de forma remota (devido à pandemia de COVID-19), resolveu-se remover da base os dados dessas turmas por haver possibilidade de cola, além da diferença de condições de aplicação das avaliações.

Ao final dessa extração, 709 questões foram selecionadas. No entanto, como visto por Lima et al. (2021), questões com poucas interações de alunos não são adequadas para amostragem, pois levam em consideração o desempenho de poucos estudantes. Portanto, utilizou-se o mesmo critério de Lima et al. (2021) e foram selecionadas apenas as questões com submissões de 16 ou mais alunos. Como resultado, obteve-se um *dataset* com 395 questões, sendo essa a base utilizada neste trabalho.

3.3 Variáveis independentes

As variáveis independentes consistem em atributos de código, tais como *complexidade ciclomática*, *número de operadores* e *número de linhas lógicas*. A extração de atributos foi feita utilizando os *scripts*[4] disponibilizados por Lima et al. (2021). Ao final, a complexidade de cada código da base seria descrita por meio de 130 atributos[5]. No entanto, vários deles não foram levados em consideração, pois eram atributos referentes a tópicos mais avançados, como classes ou expressões *lambda*, os quais não abordados em IPC. Portanto, dos 130 atributos originais, 67 foram removidos, restando 63.

Para selecionar os códigos que serviriam de modelo para a extração de atributos, foi utilizada a solução do instrutor para as questões que tinham essa informação. A solução do instrutor é um código feito pelo próprio professor ao cadastrar a questão no sistema, o qual serve como modelo de solução. Das 395 questões, 316 tinham códigos-fonte elaborados pelo próprio instrutor, sendo eles utilizados como referência para extração das métricas de complexidade. Para as 79 questões restantes, não havia solução cadastrada. Para não descartá-las, foi adotada a estratégia de selecionar as soluções dos próprios alunos, idealmente a que mais se aproximaria de uma solução do instrutor. Como a seleção manual seria muito custosa, a estratégia adotada para seleção foi baseada nos mesmos critérios adotados por Lima et al. (2021): 1. solução correta; 2. menor número de submissões; 3. menor número de testes; e 4. menor número de erros de sintaxe.

3.4 Variáveis dependentes

Neste estudo, foram empregadas como variáveis dependentes as métricas de dificuldade de uma questão, obtidas através da interação dos estudantes com as questões, registradas nos arquivos de *logs* do CodeBench. Ao total, foram definidas 13 variáveis dependentes. Algumas delas foram tomadas de trabalhos anteriores, enquanto outras são propostas neste trabalho. A Tabela 1 descreve sucintamente cada variável. As que foram definidas nesta pesquisa foram: *taxa de corretude*, *número de erros de lógica*, *número de erros* e *número de estudantes sem submissão*. A motivação para a criação dessas variáveis são dadas a seguir:

- **Taxa de corretude**: foi criada pensando em uma maneira de capturar dados sobre questões que tiveram um alto número de submissões, ao invés de somente medir a proporção de alunos que acertaram a questão (*taxa de acerto*). A hipótese é de que, questões com alto número de submissões, mesmo com taxa de acerto alta, pode indicar que a questão pode não ser tão simples quanto parece.

- **Número de estudantes sem submissão**: procura capturar dados sobre a quantidade de alunos que, por não conseguirem codificar uma solução funcional (mesmo que incorreta) para uma questão, acabaram desistindo de resolvê-la. Para garantir que o aluno tenha de fato interagido com a questão, adotou-se a estratégia de selecionar apenas os casos de alunos que interagiram com a questão por mais de 3 minutos.

- **Número de erros de lógica**: uma submissão pode estar incorreta por dois motivos: o código enviado está sintaticamente incorreto ou a solução não funciona para todos os casos de teste.

[4] https://github.com/marcosmapl/codebench-miner-tool

[5] A descrição das variáveis independentes pode ser encontrada em: http://bit.ly/3G9LIY0

Tabela 1: Descrição das variáveis dependentes

Métrica de dificuldade	Rótulo	Descrição
Taxa de acerto	taxa_acerto	Razão entre a quantidade de alunos que acertaram a questão e a quantidade de alunos que submeteram a questão pelo menos uma vez (Lima et al., 2021).
Número de submissões	num_submissoes	Quantidade média de submissões feitas para uma questão. Essa métrica é definida como *attempts* por Pereira, Fonseca et al. (2021).
Taxa de corretude	taxa_corretude	Razão entre o número de submissões corretas e o total de submissões feitas para uma questão.
Número de testes	num_testes	Quantidade média de execuções de uma questão usando o ambiente do JO. Adaptação da métrica proposta por Silva et al. (2022).
Número de consultas	num_consultas	Somatório entre o número de submissões e o número de testes da questão. Métrica proposta por Silva et al. (2022).
Número de erros de lógica	num_erros_lgcs	Quantidade média de submissões que não passaram em todos os casos de teste por não terem gerado a saída correta.
Número de erros de sintaxe	num_erros_stx	Quantidade média de submissões que geraram erro de execução durante os casos de teste. Adaptação da métrica *SyntaxError* de Pereira, Fonseca et al. (2021).
Número de erros	num_erros	Somatório entre número de erros lógicos e o número de erros de sintaxe.
Número de eventos	num_eventos	Média de linhas do arquivo de *log* de cada aluno que interagiu com a questão, similar à métrica *events* de Pereira, Fonseca et al. (2021).
Número de eventos de deleção	num_eventos_del	Média de vezes que a tecla *backspace* ou *del* foi pressionada. Métrica proposta por Pereira, Fonseca et al. (2021).
Tempo de implementação	tempo_implementacao	Tempo médio decorrido entre a primeira interação do aluno com a questão até a primeira submissão correta. Eventos consecutivos com mais de 5 minutos de diferença são considerados e não entram no cálculo desta métrica, por suspeita de inatividade.
Número de estudantes sem submissão	num_std_sem_submissao	Número de estudantes que interagiram por pelo menos 3 minutos com a questão, mas não fizeram nenhuma submissão.
Quantidade de alterações no código	qtd_alteracoes_codigo	Número de modificações no código entre duas submissões consecutivas, semelhante à métrica *amountOfChange* usada por Pereira, Fonseca et al. (2021).

No entanto, códigos com erros de sintaxe tendem a indicar mais a dificuldade do aluno com a linguagem de programação do que com o problema em si. Dessa forma, pensou-se em criar uma métrica cujo objetivo seja selecionar apenas as submissões com erros de lógica.

- **Número de erros**: A finalidade para a utilização dessa métrica é compará-la com a métrica de *número de erros de lógica*. A hipótese é de que essa métrica deva ter resultados inferiores à métrica que captura apenas erros de submissão, visto que ela também engloba a quantidade de submissões com erros de sintaxe.

3.5 Categorização das variáveis dependentes

Para responder à QP2, um dos estudos necessários é treinar modelos de aprendizado de máquina que possam prever a dificuldade de uma nova questão, por meio das métricas de código de sua solução modelo. Como não existe uma classificação prévia da dificuldade das questões, utilizaram-se heurísticas para defini-las para cada questão da base. A classificação foi feita de duas formas: classificação binária (*fácil* ou *difícil*) e classificação ternária (*fácil*, *médio* ou *difícil*). A classificação em três classes de dificuldade foi adotada por ser mais expressiva para o professor do que simplesmente uma rotulação “fácil” ou “difícil”. Entretanto, por não haver garantia de que o desempenho se mantenha, foi necessário experimentar e comparar com a classificação em dois níveis. Rotulações com mais níveis de dificuldade não foram experimentadas devido à baixa quantidade de amostras que cada classe teria.

Para a *taxa de acerto*, utilizou-se a convenção de “índice de facilidade”, do Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira (INEP) (INEP, 2017). Como originalmente existem cinco classes, foi necessário adaptá-la para a rotulação ser feita com base em duas ou três classes. A Tabela 2 mostra como a adaptação foi feita para ambos os casos de classificação binária e ternária.

Tabela 2: Classificação da dificuldade para a taxa de acerto, baseada na classificação do INEP

<table>
<tr><th>Valor</th><th>Classificação INEP</th><th>Binária</th><th>Ternária</th></tr>
<tr><td>$\geq 0,86$</td><td>Muito fácil</td><td rowspan="2">Fácil</td><td>Fácil</td></tr>
<tr><td>$0,61$ a $0,85$</td><td>Fácil</td><td>Médio</td></tr>
<tr><td>$0,41$ a $0,60$</td><td>Médio</td><td rowspan="3">Difícil</td><td rowspan="3">Difícil</td></tr>
<tr><td>$0,16$ a $0,40$</td><td>Difícil</td></tr>
<tr><td>$\leq 0,15$</td><td>Muito difícil</td></tr>
</table>

Já para as outras métricas de dificuldade, a heurística adotada foi classificar as questões com base na divisão igualitária dos dados em n partes, sendo n o número de classes desejadas. Portanto, para classificação binária ($n = 2$), a divisão foi baseada na mediana, ou seja, uma questão seria rotulada como “fácil”, se estivesse antes da mediana (ou fosse a própria mediana), e “difícil”, caso contrário. Para a classificação ternária ($n = 3$), a distribuição foi baseada em tercis, de maneira que o primeiro, segundo e terceiro tercil correspondem respectivamente às questões rotuladas como “fácil”, “médio” e “difícil”.

Um aspecto importante de se observar é o balanceamento entre as classes, pois isso influencia no desempenho dos modelos (Viloria et al., 2020). Para as métricas rotuladas com base em divisão igualitária, não há desbalanceamento. No entanto, para a taxa de acerto, na qual foi usada uma heurística de classificação diferente, as classes sofreram um desbalanceamento significativo. Isso aconteceu porque boa parte das questões possuem taxa de acerto entre 70 e 90%, como é possível observar na Figura 2. Tanto para a classificação binária (Figura 3a) quanto para a ternária (Figura 3b), somente 13% das questões foram classificadas como “difícil”. Para a classificação

ternária, o desbalanceamento é menor, com 34% das questões sendo classificadas como “fácil”, restando 53% das questões, que foram categorizadas como “média”.

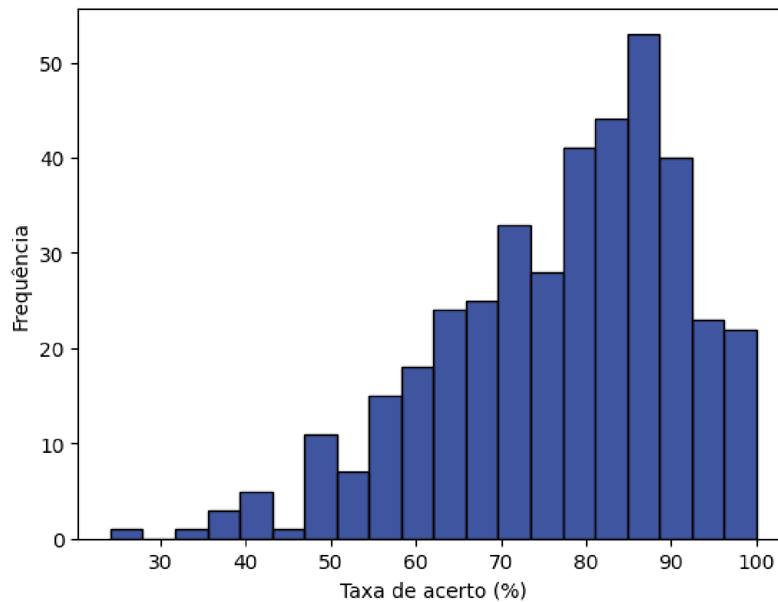


Figura 2: Distribuição de frequência da taxa acerto

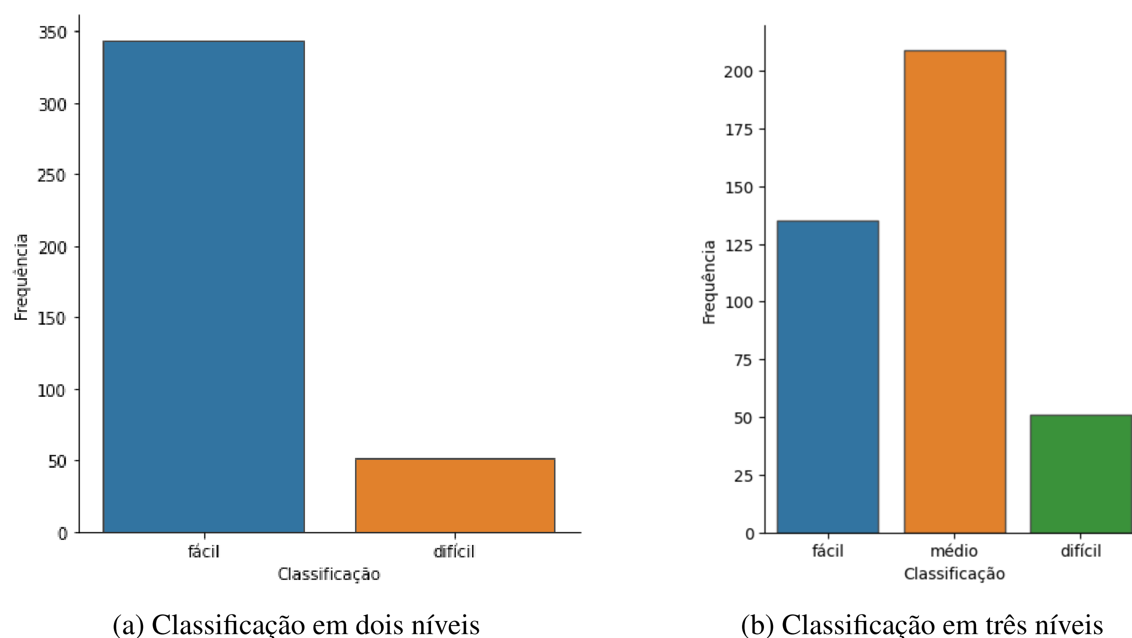
3.6 Treinamento dos modelos de classificação e regressão

Para os experimentos com classificação e regressão, foram utilizados os modelos já implementados nas bibliotecas *scikit-learn* e *xgboost* da linguagem *Python*. Os modelos baseados em árvores foram escolhidos por apresentarem bom desempenho em bases com estruturas tabulares, com variáveis significativas e sem estruturas temporais ou espaciais multiescalares (Chen & Guestrin, 2016; Lundberg et al., 2020). Já os modelos baseados em vetores de suporte foram escolhidos devido a sua robustez diante de dados de grande dimensão, sobre os quais outras técnicas de aprendizado, como as redes neurais, comumente obtêm classificadores super ou sub-ajustados (Lorena & Carvalho, 2007).

Para o experimento com classificação, os seguintes modelos foram testados:

- *Support vector machine* (**SVM**)
- *Árvores de decisão* (**DT**)
- *Random forest* (**RF**)
- *Gradient boosting* (**GB**)
- *Extreme gradient boosting* (**XGBoost**)

Já para os experimentos com regressão, os modelos usados foram:



(a) Classificação em dois níveis

(b) Classificação em três níveis

Figura 3: Distribuição de frequência das classes de dificuldade baseado na classificação do INEP para a taxa de acerto

- *Árvores de regressão* (**RT**)
- *Support vector regression* (**SVR**) [6]
- *Random forest* (**RF**)
- *Extreme gradient boosting* (**XGBoost**)

Além disso, utilizou-se também a função *RandomizedSearchCV*, fornecida pela própria *scikit-learn*, a qual seleciona e combina aleatoriamente os hiperparâmetros do modelo dentro de combinações pré-definidas até atingir um limite de iterações, quando este retorna a combinação que gerou o melhor *score*. Neste trabalho, o limite definido foi de 50 iterações.

Para a *taxa de acerto*, foi utilizada a técnica *SMOTE* (Chawla et al., 2002), dado o problema de desbalanceamento das classes para esta métrica. Essa técnica foi escolhida porque a quantidade de amostras da base não era adequada para subamostragem.

Para a obtenção de métricas mais confiáveis, os treinos foram realizados usando a técnica *k-fold cross-validation* (Anguita et al., 2009), com $k = 4$, ou seja, três partições para treino e uma para teste.

[6] Além do SVR padrão, também foi testada uma variação chamada de NuSVR. Nessa versão, o número de vetores de suporte e as margens de erro são controlados por um parâmetro denominado *nu* (Schölkopf et al., 2000)

3.7 Métricas para avaliação dos modelos

Uma etapa fundamental no processo de avaliar corretamente um modelo de aprendizagem de máquina consiste em definir bem as métricas a serem utilizadas (Naser & Alavi, 2021). Para os modelos de classificação, as métricas mais comumente usadas são *precisão*, *revocação*, *acurácia* e *f1-score*. A métrica de precisão mede o percentual de predições corretas feitas para uma dada classe em relação ao total de elementos que pertencem a tal classe. Já a métrica de revocação mede o percentual de predições corretas de uma classe em relação ao total de elementos que foram classificados como correspondentes a tal classe. A métrica de acurácia mede o percentual de predições corretas em relação ao total de predições feitas. Por fim, a métrica de f1-score é calculada a partir da média harmônica entre os valores de precisão e revocação, de maneira que ambas as variáveis são descritas a partir de um único valor.

Com exceção da *acurácia*, todas as métricas são originalmente utilizadas para classificação binária. Portanto, foi necessário adaptá-las para serem compatíveis com classificação em três níveis. Isso pode ser feito usando *macro averaging* ou *micro averaging* (Manning et al., 2008). Com *macro averaging*, cada métrica (precisão, revocação ou *f1-score*) é calculada individualmente para cada classe, sendo a média entre esses valores o resultado final. Para *micro averaging*, os valores de verdadeiros-positivos e verdadeiros-negativos de cada classe são somados, sendo divididos pelo total de exemplos de teste.

A desvantagem de usar *macro averaging* é que o cálculo desta métrica não leva em consideração o desbalanceamento entre as classes (Manning et al., 2008), enquanto que *micro averaging*, apesar de ser atribuído o mesmo peso para cada classificação, as métricas de *precisão*, *revocação* e *acurácia* acabam sendo iguais (Grandini et al., 2020). É importante notar que, quanto mais balanceadas são as classes, mais o valor de *micro averaging* e *macro averaging* são próximos, o que torna irrelevante o modelo a ser escolhido, o que é o caso de todas as variáveis dependentes usadas, com exceção da taxa de acerto. Dessa forma, somente a *micro averaging* foi considerada como métrica principal de avaliação, sendo necessário o apoio da matriz de confusão gerada pelas predições para se ter uma análise mais aprofundada, principalmente para a *taxa de acerto*.

Para os modelos de regressão, foi realizada uma análise baseada em três métricas: erro médio absoluto (**MAE**), erro relativo absoluto (**RAE**), e coeficiente de determinação ajustado ($\mathbf{R^2_{adj}}$). A opção pelo *RAE* deve-se ao fato de que seus valores são independentes de escala (Naser & Alavi, 2021), tornando viável a comparação entre as métricas de dificuldade que trabalham com proporções distintas, como entre a *taxa de acerto* e o *tempo de implementação*. Já a escolha do R^2 teve como fundamento a própria finalidade da métrica: calcular o quão bem o modelo explica a variação dos dados. Como o coeficiente de determinação original nunca decai quando o número de *features* aumenta, optou-se por usar o coeficiente de determinação ajustado (R^2_{adj}), o qual penaliza modelos com alto número de *features* possivelmente irrelevantes (Di Bucchianico, 2008).

Como as métricas de dificuldade foram discretizadas para os experimentos com classificação, tornou-se possível usar os modelos de regressão também como preditores de dificuldade, utilizando a mesma estratégia descrita em 3.5. A ideia é que, além da informação da dificuldade da questão em si, o modelo consiga prever as próprias métricas de dificuldade, como, por exemplo, o tempo médio de implementação da questão.

4 Resultados

Esta seção descreve os resultados obtidos tanto para os experimentos com a correlação entre métricas de complexidade e dificuldade, quanto para os experimentos com modelos de regressão e classificação.

4.1 Correlação entre variáveis

Nesta subseção, são apresentados os resultados para os estudos de correlação entre as variáveis dependentes, bem como entre dependentes e independentes. Para esse experimento, usou-se a correlação de *Spearman*, uma medida não paramétrica que descreve a relação entre as variáveis descritas a partir de uma função monótona. Partindo do pressuposto de que as relações podem não ser lineares, além dos dados não seguirem uma distribuição normal, optou-se por utilizar essa medida ao invés da correlação de *Pearson* (Myers & Sirois, 2006).

4.1.1 Correlação entre variáveis dependentes

A primeira análise feita foi entre variáveis dependentes. O objetivo desse estudo é validar algumas hipóteses de como os estudantes costumam interagir com o CodeBench durante o processo de resolução de uma questão. A Figura 4 exibe uma matriz de correlação, na qual os valores abaixo da diagonal principal correspondem à correlação obtida, enquanto os valores acima referem-se aos níveis de significância (p-valores). Para cada célula, quanto mais próxima da cor amarela, mais fraca a correlação, e quanto mais próximas de azul, mais forte a correlação.

Considerando-se apenas os casos de correlação entre variáveis distintas, houve 47% de casos de correlação moderada, 33% de correlações fracas, 15% de correlações fortes, 3% de correlações nulas e 1% de correlação perfeita[7]. Além disso, houve apenas 3 casos em que o nível de significância foi maior que 5%.

4.1.2 Correlação entre variáveis independentes e dependentes

Após a análise de correlação entre variáveis dependentes, foi realizado um estudo sobre a correlação entre as variáveis dependentes e independentes da base. A Figura 5 mostra as correlações para os 10 atributos com mais ocorrências no *top-10* de correlações de *Spearman* para cada métrica de dificuldade.

Considerando todos os 63 atributos de código, viu-se que 96% das correlações são fracas ou inexistentes, 4% de correlações moderadas e nenhum caso de correlação forte ou perfeita. Considerando apenas as 10 métricas da Figura 5, 85% das correlações foram fracas ou inexistentes, 15% de correlações moderadas e nenhum caso de correlação forte ou perfeita.

[7] A intensidade das correlações é baseada em Dancey e Reidy (2007).

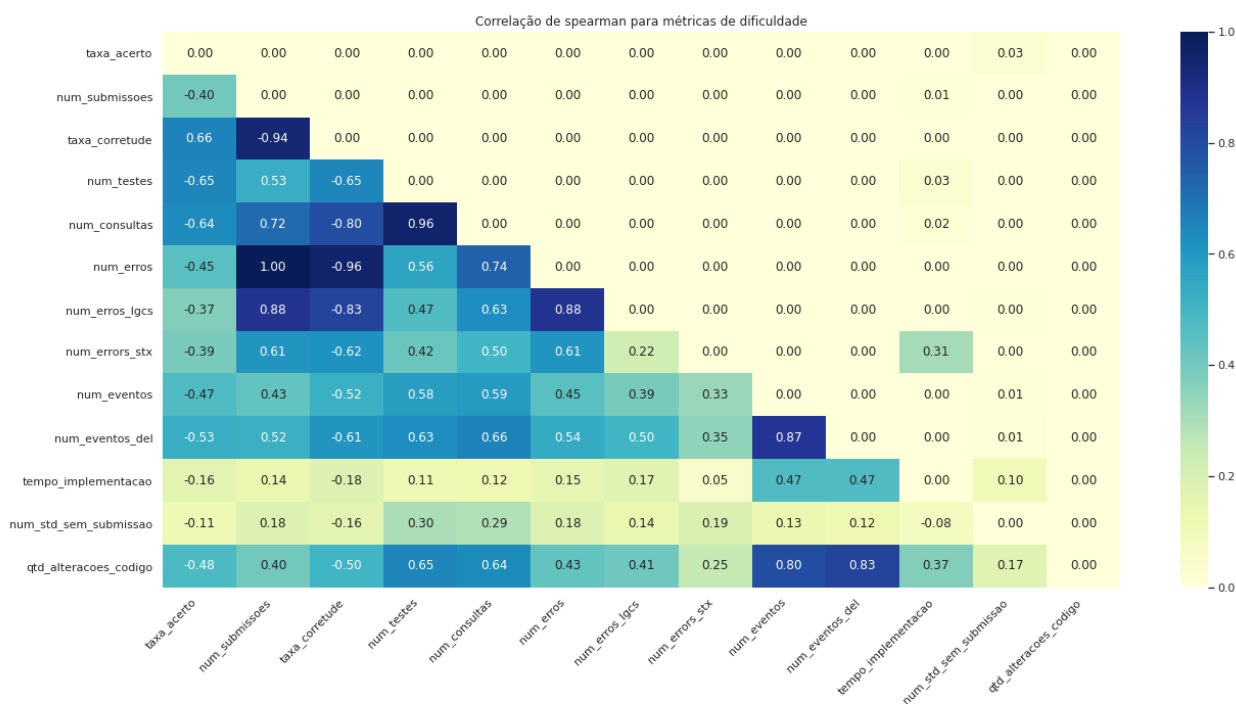


Figura 4: Matriz de correlação de *Spearman* entre as variáveis dependentes

Figura 5: Correlação de *Spearman* entre variáveis independentes e dependentes

4.2 Regressão

A Tabela 3 apresenta os resultados obtidos para os experimentos com modelos de regressão. Para cada métrica de dificuldade, é apresentado o modelo que gerou o melhor resultado, junto com o *RAE*, *MAE* e o R^2_{adj} correspondentes. As linhas estão ordenadas de forma não-crescente pelo R^2_{adj}.

Tabela 3: Resultados para o modelo de regressão

Métrica de dificuldade	Modelo	MAE	RAE	R^2_{adj}
Tempo de implementação	XGBoost	146,15	0,54	0,63
Quantidade de alterações no código	RF	170,10	0,68	0,47
Número de eventos	RF	193,96	0,68	0,46
Número de eventos de deleção	RF	24,48	0,72	0,43
Taxa de acerto	RF	0,09	0,74	0,37
Número de testes	RF	3,75	0,77	0,32
Número de consultas	RF	4,65	0,77	0,32
Número de estudantes sem submissão	RF	6,18	0,84	0,22
Número de erros de lógica	RF	1,29	0,85	0,16
Taxa de corretude	RF	0,08	0,86	0,13
Número de erros	RF	1,63	0,88	0,12
Número de submissões	RF	1,61	0,89	0,09
Número de erros de sintaxe	RF	0,70	0,90	0,08

Como é possível observar, o modelo de *XGBoost* para a métrica *tempo de implementação* gerou os melhores resultados para o R^2_{adj}, conseguindo explicar 63% da variação dos dados. A métrica *quantidade de alterações no código* foi a segunda melhor, explicando 47% da variação dos dados.

A Tabela 4 mostra o desempenho obtido pelos modelos de regressão quando as predições, originalmente valores contínuos, foram categorizadas em dois e três níveis de dificuldade (Tabelas 4a e 4b, respectivamente). Para a categorização em dois níveis, a variável *taxa de acerto*, usando o modelo de *Random Forest*, foi a métrica com o melhor desempenho, tendo um *f1-score* de 87%. Já para a categorização em três níveis, a melhor métrica foi o *tempo de implementação*, com *f1-score* de 64%. É importante observar a queda de desempenho dos modelos, quando se compara predição com três classes com a classificação binária, principalmente da taxa de acerto, que decaiu em 23%. Este problema será discutido com detalhes na Seção 5.

4.3 Classificação

Os resultados para os modelos de classificação com dois e três níveis são mostrados nas Tabelas 5 e 6, respectivamente. Para cada métrica, há o classificador que obteve o melhor *f1-score*, além do valor obtido. Para a classificação em dois níveis, a variável com melhor predição foi a *taxa de acerto*, com *f1-score* de 88%, seguida do *tempo de implementação*, com 80%. Para a classificação em três níveis, a melhor métrica foi o *tempo de implementação*, com *f1-score* de 67%, seguida do *número de eventos de deleção*, com 61%.

Tabela 4: Resultados para o modelo de regressão com as predições categorizadas

(a) Categorização em dois níveis de dificuldade

Métrica de dificuldade	***f1-score***
Taxa de acerto	0,87
Tempo de implementação	0,80
Quantidade de alterações no código	0,78
Número de eventos	0,75
Número de consultas	0,73
Número de eventos de deleção	0,73
Número de testes	0,72
Taxa de corretude	0,70
Número de estudantes sem submissão	0,68
Número de erros	0,66
Número de submissões	0,64
Número de erros de lógica	0,64
Número de erros de sintaxe	0,61

(b) Categorização em três níveis de dificuldade

Métrica de dificuldade	***f1-score***
Tempo de implementação	0,64
Taxa de acerto	0,62
Quantidade de alterações no código	0,61
Número de eventos de deleção	0,61
Número de eventos	0,60
Número de testes	0,53
Número de consultas	0,52
Número de estudantes sem submissão	0,49
Número de erros de lógica	0,49
Número de submissões	0,47
Número de erros	0,47
Número de erros de sintaxe	0,44
Taxa de corretude	0,44

Tabela 5: Resultados para o modelo de classificação com dois níveis de dificuldade

Métrica de dificuldade	**Classificador**	***f1-score***
Taxa de acerto	SVM	0,88
Tempo de implementação	XGBoost	0,80
Número de eventos	RF	0,79
Quantidade de alterações no código	RF	0,77
Número de testes	RF	0,74
Número de consultas	XGBoost	0,74
Número de estudantes sem submissão	XGBoost	0,72
Número de eventos de deleção	SVM	0,71
Taxa de corretude	RF	0,71
Número de erros	RF	0,68
Número de submissões	SVM	0,67
Número de erros de lógica	RF	0,66
Número de erros de sintaxe	XGBoost	0,63

5 Discussão

Nesta seção é feita uma análise sobre os resultados obtidos na Seção 4. Em 5.2 é feita uma análise sobre a classificação em dois e três níveis para a *taxa de acerto*. Em 5.3 é feita uma análise em cima das outras métricas, com ênfase em *tempo de implementação*, visto que esta métrica esteve entre os melhores resultados nos experimentos realizados. Cabe ressaltar que a análise da *taxa de acerto* deve ser diferente das outras métricas de dificuldade, pois foi a única em que as classes ficaram desbalanceadas. Em 5.4, serão discutidos os resultados para os modelos de regressão.

Tabela 6: Resultados para o modelo de classificação com três níveis de dificuldade

Métrica de dificuldade	**Classificador**	***f1-score***
Tempo de implementação	XGBoost	0,67
Número de eventos	RF	0,61
Número de eventos de deleção	XGBoost	0,61
Quantidade de alterações no código	RF	0,60
Taxa de acerto	RF	0,58
Número de testes	RF	0,54
Número de consultas	RF	0,54
Número de erros de lógica	GB	0,50
Número de estudantes sem submissão	RF	0,50
Número de submissões	XGBoost	0,49
Número de erros de sintaxe	GB	0,48
Taxa de corretude	XGBoost	0,47
Número de erros	RF	0,47

5.1 Análise das correlações entre variáveis

5.1.1 Variáveis dependentes × dependentes

Como é possível observar pela Figura 4, muitas correlações ficaram entre fracas e moderadas. Entretanto, também houve vários casos de correlação forte, apesar de muitas delas serem esperadas, como entre *número de consultas* e *número de testes*, visto que o valor do *número de consultas* é baseado no valor do *número de testes*. Também houve um caso de correlação perfeita, entre *número de erros* e *número de submissões*, a qual ocorreu devido o *número de submissões* ser o *número de erros* acrescido do número de acertos.

Baseado nas correlações mostradas, algumas ponderações podem ser feitas. A primeira é que o fato do *número de consultas* ter maior correlação com o *número de testes* do que com *número de submissões*, pode ser um indício de que os alunos costumam testar mais a questão antes de submetê-la. A segunda diz respeito ao fato da correlação de *Spearman* para o *número de erros* ser maior com o *número de erros lógicos* do que com *número de erros de sintaxe*, indicando que uma questão submetida costuma ter mais erros de lógica do que de sintaxe. Combinando essas duas ponderações, é válido concluir que os alunos costumam submeter códigos funcionais justamente porque testam mais.

5.1.2 Variáveis dependentes × independentes

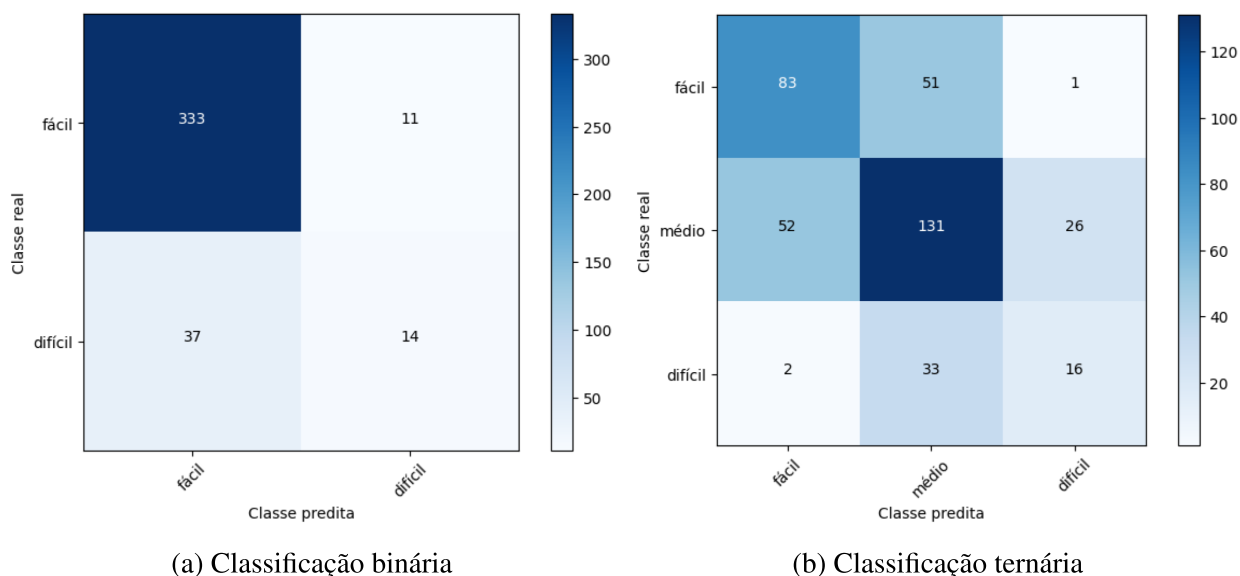
Baseado nas correlações mostradas na Figura 5, pode-se concluir que nenhum dos atributos individuais de complexidade de código considerados conseguem indicar a dificuldade de uma questão, o que responde a QP1. Entretanto, ponderações podem ser feitas para algumas métricas de dificuldade. Para a métrica de *quantidade de alterações de código*, é possível verificar várias correlações moderadas com variáveis independentes relacionadas com o tamanho do código, como o *número de linhas lógicas* (loc) ou o *número de linhas de código-fonte* (sloc), o que pode indicar que o tamanho do código influencia na quantidade de modificações que são feitas nele.

Outra observação é sobre a métrica de *número de erros de sintaxe*. Observando-se a coluna correspondente, é possível notar que todas as correlações com todas as outras variáveis foram

nulas. Isso leva à conclusão de que a complexidade do código não exerce influência sobre a quantidade de submissões que levarão a erros de sintaxe acusados pelo JO. A principal razão é que essa métrica tem mais a ver com a dificuldade do aluno com a linguagem de programação do que propriamente com o problema.

5.2 Classificação em dois níveis e três níveis para a taxa de acerto

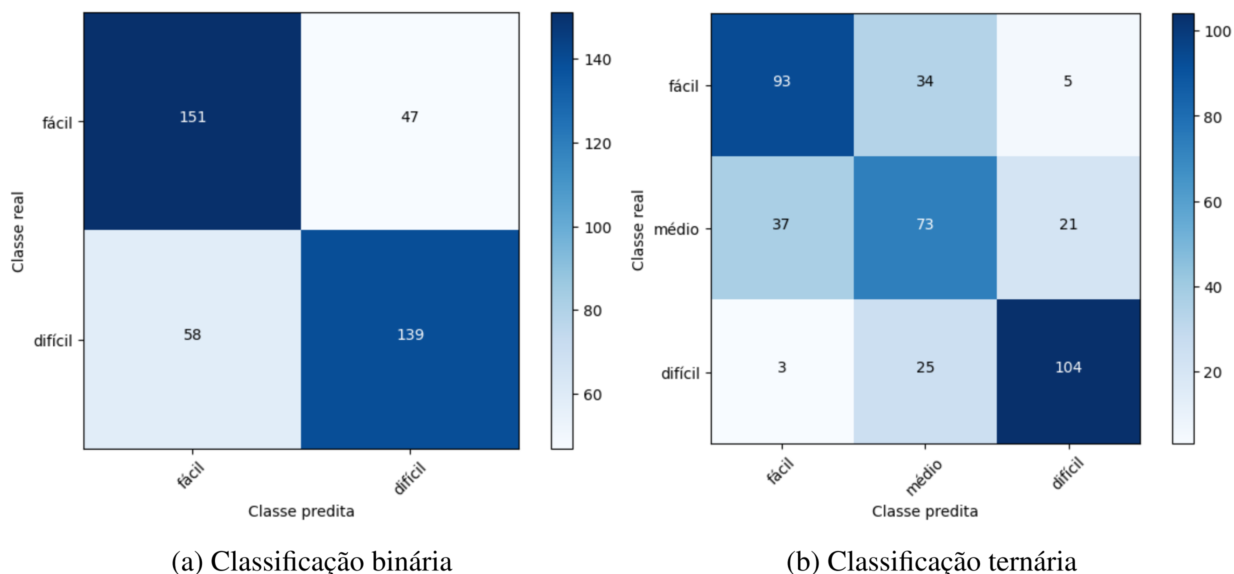
Uma das perguntas a se responder é por que a predição com dois níveis de dificuldade obteve desempenho muito superior àquela com três níveis. Essa pergunta pode ser respondida quando se observa a matriz de confusão gerada para esses casos. Para a taxa de acerto, a matriz de confusão para a classificação com duas categorias é mostrada na Figura 6a. Observando-se a figura, percebe-se que o modelo teve mais falsos-negativos do que verdadeiros-negativos, o que pode indicar que o modelo não conseguiu aprender bem como classificar uma questão como "difícil", sendo bastante induzido a classificar as questões como "fácil" em várias ocasiões. Como 87% da amostra é composta por questões categorizadas como "fácil", esse viés incorporado pelo modelo acaba passando despercebido pela métrica de *micro averaging f1-score*, sendo necessário verificar a matriz de confusão para detectar esse problema. Mesmo empregando a técnica *SMOTE* para balancear as amostras de cada classe, isso não foi suficiente para fazer os modelos de classificação aprenderem a diferenciar bem as duas classes.



(a) Classificação binária (b) Classificação ternária

Figura 6: Matriz de confusão para a métrica de taxa de acerto

Para a categorização em três níveis, a matriz de confusão é mostrada na Figura 6b. Como é possível notar, boa parte das predições foram "média", sendo 54% das predições, mas com precisão de 63% e revocação de 61%. Observando as predições erradas, é possível ver que muitas aconteceram entre as classes "fácil" e "médio". Além disso, nota-se também que a classe "difícil" teve mais predições erradas do que corretas, tendo precisão de 31% e revocação de 37%. Esses dois fatos evidenciam o porquê do modelo de classificação com três níveis de dificuldade ter um desempenho baixo para a taxa de acerto.



(a) Classificação binária

(b) Classificação ternária

Figura 7: Matriz de confusão para a métrica de tempo de implementação

Dessa forma, é possível responder a QP2 para a taxa de acerto da seguinte forma: para o conjunto de dados analisado, a classificação tanto em dois níveis de dificuldade quanto em três não é eficiente para identificar questões difíceis.

5.3 Classificação em dois níveis e três níveis para outras métricas de dificuldade

Tomando como base a métrica de *tempo de implementação*, por ter obtido o melhor desempenho, a matriz de confusão para o modelo de classificação com duas classes é mostrada na Figura 7a. Como é possível observar, 73% das predições foram corretamente feitas. Além disso, houve um certo equilíbrio entre a quantidade de predições corretas e incorretas para cada classe. Portanto, ao contrário do que ocorreu com a *taxa de acerto*, a classificação usando a métrica de *tempo de implementação* não enviesou para uma classe em específica, apenas não conseguiu distinguir com eficácia as classes.

Observando a matriz de confusão para os modelos de classificação com três classes (Figura 7b), nota-se que o mesmo problema ocorreu. No entanto, neste caso é possível notar um detalhe importante: 94% das classificações erradas foram entre classes adjacentes, ou seja, para vários casos o modelo classificou como "fácil" quando deveria ser "médio" (e vice-versa), ou o modelo rotulou como "médio" quando deveria "difícil" (e vice-versa). Isso de certa forma aconteceu também para classificação binária. Três possíveis motivos não concorrentes podem explicar esse problema: o modelo não conseguiu aprender bem como diferenciar a dificuldade das questões, a própria estrutura espacial dos dados não possui um padrão bem-definido para diferenciar as classes, ou a falta de mais exemplos fizeram com que o modelo não aprendesse a diferenciar as classes.

Por fim, respondendo a QP2 para as outras métricas de dificuldade, a classificação em dois níveis de dificuldade é razoavelmente adequada e ainda inviável para três níveis, considerando-se o conjunto de dados analisado.

5.4 Análise sobre os resultados com modelos de regressão

Para os experimentos com modelos de regressão, constatou-se que o *tempo de implementação* foi a métrica que obteve o melhor coeficiente de determinação, além de também ter os menores valores de *RAE* e *MAE*. Para as outras métricas, os valores de R^2_{adj} ficaram bem abaixo se comparado com *tempo de implementação*. No entanto, quando se observa os valores de *MAE*, é possível notar que algumas métricas possuem valores razoáveis, como a *taxa de acerto*, o qual obteve um erro de 9%. Portanto, conclui-se que os modelos de regressão são mais adequados para a predição do tempo de implementação de uma questão e, em menor escala, para outras métricas de dificuldade, o que leva à resposta da QP3.

Para os experimentos com regressores sendo usados como classificadores, notou-se pouca diferença em relação aos modelos de classificação em si. Assim como os classificadores, os regressores tiveram valores mais altos para a classificação binária (*f1-score* máximo de 0,87), mas houve uma queda de desempenho significativa quando usado com classificação ternária (*f1-score* máximo de 0,64). Portanto, a escolha entre um modelo de regressão a ser usado como classificador e um classificador propriamente dito é irrelevante quando o objetivo é apenas obter um rótulo de dificuldade, mas o regressor para o *tempo de implementação* pode ser útil caso se queira se obter uma previsão do tempo de implementação de uma questão. Uma aplicação prática dessa predição poderia ser útil para o professor ao elaborar uma prova, pois assim é possível ter um controle mais preciso do tempo esperado para sua resolução.

5.5 Limitações

Pelos resultados do modelo de regressão, algumas métricas de dificuldade, como *tempo de implementação* e a *taxa de acerto*, parecem ter um erro aceitável. No entanto, pelo fato de não ter sido feito experimentos reais com os alunos para uma validação mais concreta, ainda pode haver a chance de, em uma eventual aplicação deste modelo em um cenário real, seu erro não ser satisfatório na perspectiva do professor.

Além disso, a base de questões foi retirada de apenas uma universidade. O problema disso é que, em uma eventual aplicação do modelo em um JO em outra universidade, as chances dele desempenhar bem são desconhecidas, dado que o ambiente aplicado é bem diferente.

Outra limitação importante detectada foi a falta de dados para construção de bons preditores. Para a taxa de acerto, a falta de exemplos de questões rotuladas como "difícil" induziu ao viés de classificar as questões como "fácil", enquanto para as outras métricas, a falta de dados implicou que os modelos não conseguiram discernir as classes com a eficácia esperada.

Outro problema detectado neste trabalho foi a rotulação manual das métricas de dificuldade. Para a taxa de acerto, a classificação do INEP foi adotada, enquanto que, para as outras métricas, aplicou-se a divisão em mediana e tercis. No entanto, apesar do amplo uso de divisão em mediana e tercis na literatura educacional (Pereira, Junior et al., 2021), não há garantia de que essas rotulações são as mais adequadas para este contexto, nem as que vão gerar as melhores classificações.

6 Considerações finais

Este trabalho teve como objetivo analisar a correlação entre métricas de código e métricas de dificuldade em questões de programação. Para isso, foram usadas três abordagens: correlação entre variáveis (Spearman), predição da dificuldade usando modelos de classificação e predição de métricas de dificuldade por meio de regressão.

Para a correlação de *Spearman* entre métricas de dificuldade, observou-se muitos valores moderados entre variáveis dependentes, exceto quando as duas variáveis tinham alguma relação matemática, como entre o *número de consultas* e *número de testes*. No entanto, foi possível notar alguns padrões de comportamento dos alunos, tais como uma tendência maior a testar os códigos antes de submeterem ao JO. Já para a correlação entre métricas individuais de complexidade de código e métricas de dificuldade, observaram-se muitas correlações fracas, o que indica que não é possível explicar a dificuldade de uma questão a partir de um único atributo de código, o que levou aos experimentos com classificação e regressão, os quais são baseados na composição de atributos.

Para a predição da dificuldade, notou-se que a falta de exemplos da classe “difícil” prejudicou o desempenho dos modelos de classificação para a taxa de acerto, que ficaram enviesados em rotular as questões como “fácil” (classificação binária) ou “médio” (classificação em três níveis). Para outras métricas, a falta de exemplos implicou na classificação errônea para várias amostras usadas para os testes, sobretudo entre classes adjacentes. Além disso, para a amostra utilizada, a classificação em três níveis, apesar de ser mais adequada dada sua maior expressividade, obteve desempenho baixo para todas as métricas de dificuldade analisadas, sendo inadequada em uma eventual aplicação do modelo em produção. A classificação em dois níveis, apesar de menos informativa, aparece como solução alternativa, enquanto não se consegue obter predições melhores com mais classes.

Para a predição das métricas de dificuldade através de regressão, o melhor resultado obtido foi usando a métrica de *tempo de implementação*, cujo R^2_{adj} foi de 63%, obtido através do treino com o modelo de *XGBoost*. Usando os regressores como classificadores, o desempenho obtido foi similar aos métodos de classificação propriamente ditos. Portanto, o uso de métodos de regressão é mais encorajado, visto que além da classe, também é possível prever um valor referente à métrica de dificuldade da questão em si. No entanto, é importante ressaltar que a eficácia da predição dos valores não é boa para todas as métricas de dificuldade, como foi possível observar na Tabela 3 da Seção 4.2.

Baseado nos estudos feitos, foi possível confirmar a viabilidade de se implementar sistemas de classificação de predição de questões em ACAC. Tanto na classificação em níveis quanto na predição das próprias métricas de dificuldade, este sistema permite que o professor tenha mais controle do nível de dificuldade desejado ao elaborar atividades de programação, podendo levar a um aumento significativo da curva de aprendizado dos alunos e alinhando a expectativa de desempenho com a realidade.

6.1 Trabalhos futuros

Um trabalho de validação importante a ser feito é implementar um modelo de predição em um JO real, monitorando o seu desempenho através dos dados fornecidos pelos alunos e pelo *feedback* dos próprios professores.

Já pensando em possíveis melhorias para os modelos de predição, pensa-se em combinar métricas de código com métricas de enunciado da questão, partindo do pressuposto de que, além do código em si, os alunos podem ter dificuldade em entender os aspectos linguísticos que compõem a questão. Outra possibilidade é usar a Teoria de Resposta ao Item (TRI) no lugar da Teoria Clássica dos Testes (TCT) como método de estimação de desempenho dos estudantes, com o objetivo de estimar com mais assertividade a dificuldade de uma questão.

Além disso, outra possibilidade de estudo é experimentar estratégias de aprendizagem não-supervisionada e comparar seu desempenho aos modelos experimentados nesse e em trabalhos anteriores. A ideia seria utilizar as métricas de dificuldade já estudadas e analisar possíveis agrupamentos que possam existir entre elas quando seus dados são projetados em um plano k-dimensional, sendo k a quantidade de métricas de dificuldade. Nesse caso, ao invés de rotular manualmente os dados, eles seriam rotulados com base nos agrupamentos encontrados.

Agradecimentos

Esta pesquisa, realizada no âmbito do Projeto Samsung-UFAM de Ensino e Pesquisa (SUPER), de acordo com o Artigo 39 do Decreto n°10.521/2020, foi financiada pela Samsung Eletrônica da Amazônia Ltda, nos termos da Lei Federal n°8.387/1991, através do convênio 001/2020 firmado com a UFAM e FAEPI, Brasil. Também recebeu apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq (Processo 308513/2020-7) e da Fundação de Amparo à Pesquisa do Estado do Amazonas - FAPEAM (Processo 01.02.016301.02770/2021-63).

Artigo Premiado Estendido

Esta publicação é uma versão estendida de artigo premiado no III Simpósio Brasileiro de Educação em Computação (EduComp 2023), intitulado "Correlação Entre Complexidade e Dificuldade de Questões de Programação em Juízes Online", DOI: 10.5753/educomp.2023.228217.

Referências

Anguita, D., Ghio, A., Ridella, S., & Sterpi, D. (2009). K-Fold Cross Validation for Error Rate Estimate in Support Vector Machines. *DMIN*, 291–297. [GS Search].

Bonner, S. E. (1994). A model of the effects of audit task complexity [GS Search]. *Accounting, organizations and society*, *19*(3), 213–234. https://doi.org/10.1016/0361-3682(94)90033-7

Carvalho, L. S. G., Oliveira, D. B. F., & Gadelha, B. (2016). Juiz online como ferramenta de apoio a uma metodologia de ensino híbrido em programação [GS Search]. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*, *27*(1), 140–149. https://doi.org/10.5753/cbie.sbie.2016.140

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: synthetic minority over-sampling technique [GS Search]. *Journal of artificial intelligence research*, *16*, 321–357. https://doi.org/10.1613/jair.953

Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System [GS Search]. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794. https://doi.org/10.1145/2939672.2939785

Dancey, C., & Reidy, J. (2007). *Statistics Without Maths for Psychology* [GS Search]. Pearson/-Prentice Hall.

Di Bucchianico, A. (2008). Coefficient of determination (R 2) [GS Search]. *Encyclopedia of statistics in quality and reliability*, *1*. https://doi.org/10.1002/9780470061572.eqr173

Effenberger, T., Cechák, J., & Pelánek, R. (2019). Difficulty and Complexity of Introductory Programming Problems [GS Search].

Elnaffar, S. (2016). Using software metrics to predict the difficulty of code writing questions [GS Search]. *2016 ieee global engineering education conference (educon)*, 513–518. https://doi.org/10.1109/EDUCON.2016.7474601

Francisco, R. E., Ambrósio, A. P. L., Junior, C. X. P., & Fernandes, M. A. (2018). Juiz online no ensino de CS1-lições aprendidas e proposta de uma ferramenta [GS Search]. *Revista Brasileira de Informática na Educação*, *26*(03), 163. https://doi.org/10.48550/arXiv.2008.05756

Grandini, M., Bagli, E., & Visani, G. (2020). Metrics for multi-class classification: An overview [GS Search]. https://doi.org/10.48550/arXiv.2008.05756

INEP. (2017). *Relatório Síntese de Área – Ciência da Computação (Bacharelado/Licenciatura)* (rel. técn.) (Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira. Link).

Lima, M. A. P., Carvalho, L. S. G., Oliveira, E. H. T., Oliveira, D. B. F., & Pereira, F. D. (2021). Uso de atributos de código para classificação da facilidade de questões de codificação [GS Search]. *Anais do Simpósio Brasileiro de Educação em Computação*, 113–122. https://doi.org/10.5753/educomp.2021.14477

Liu, P., & Li, Z. (2012). Task complexity: A review and conceptualization framework [GS Search]. *International Journal of Industrial Ergonomics*, *42*(6), 553–568. https://doi.org/10.1016/j.ergon.2012.09.001

Llana, L., Martin-Martin, E., & Pareja-Flores, C. (2012). FLOP, a free laboratory of programming [GS Search]. *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, 93–99. https://doi.org/10.1145/2401796.2401807

Lorena, A. C., & Carvalho, A. C. (2007). Uma introdução às support vector machines [GS Search]. *Revista de Informática Teórica e Aplicada*, *14*(2), 43–67.

Lundberg, S. M., Erion, G., Chen, H., DeGrave, A., Prutkin, J. M., Nair, B., Katz, R., Himmelfarb, J., Bansal, N., & Lee, S.-I. (2020). From local explanations to global understanding with explainable AI for trees [GS Search]. *Nature machine intelligence*, *2*(1), 56–67. https://doi.org/10.1038/s42256-019-0138-9

Manning, C. D., Raghavan, P., & Schutze, H. (2008, julho). *Introduction to Information Retrieval* [GS Search]. Cambridge University Press.

McCabe, T. J. (1976). A complexity measure [GS Search]. *IEEE Transactions on software Engineering*, (4), 308–320. https://doi.org/10.1109/TSE.1976.233837

Meisalo, V., Sutinen, E., & Torvinen, S. (2004). Classification of exercises in a virtual programming course [GS Search]. *34th Annual Frontiers in Education, 2004. FIE 2004.*, S3D–1. https://doi.org/10.1109/FIE.2004.1408764

Myers, L., & Sirois, M. J. (2006). Spearman Correlation Coefficients, Differences between [GS Search]. *Encyclopedia of Statistical Sciences*. https://doi.org/10.1002/0471667196.ess5050.pub2

Naser, M. Z., & Alavi, A. H. (2021). Error metrics and performance fitness indicators for artificial intelligence and machine learning in engineering and sciences [GS Search]. *Archit. Struct. Constr.* https://doi.org/10.1007/s44150-021-00015-8

Paes, R. B., Malaquias, R., Guimarães, M., & Almeida, H. (2013). Ferramenta para a Avaliação de Aprendizado de Alunos em Programação de Computadores [GS Search]. *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*, *2*, 203–212. https://doi.org/10.5753/CBIE.WCBIE.2013.203

Pelz, F. D., Jesus, E. A., & Raabe, A. L. (2012). Um mecanismo para correção automática de exercícios práticos de programação introdutória [GS Search]. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educaçao-SBIE)*, *23*(1).

Pereira, F. D., Fonseca, S. C., Oliveira, E. H., Cristea, A. I., Bellhäuser, H., Rodrigues, L., Oliveira, D. B., Isotani, S., & Carvalho, L. S. (2021). Explaining Individual and Collective Programming Students' Behavior by Interpreting a Black-Box Predictive Model [GS Search]. *IEEE Access*, *9*, 117097–117119. https://doi.org/10.1109/ACCESS.2021.3105956

Pereira, F. D., Junior, H. B., Rodriguez, L., Toda, A., Oliveira, E. H., Cristea, A. I., Oliveira, D. B., Carvalho, L. S., Fonseca, S. C., Alamri, A., et al. (2021). A recommender system based on effort: Towards minimising negative affects and maximising achievement in CS1 learning [GS Search]. *International Conference on Intelligent Tutoring Systems*, 466–480. https://doi.org/10.1007/978-3-030-80421-3_51

Petit, J., Giménez, O., & Roura, S. (2012). Jutge. org: an educational programming judge [GS Search]. *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, 445–450. https://doi.org/10.1145/2157136.2157267

Prisco, A., dos Santos, R., Botelho, S., Tonin, N., & Bez, J. (2017). Using information technology for personalizing the computer science teaching [GS Search]. *2017 IEEE Frontiers in Education Conference (FIE)*, 1–7. https://doi.org/10.1109/FIE.2017.8190727

Robinson, P. (2001). Task complexity, task difficulty, and task production: Exploring interactions in a componential framework [GS Search]. *Applied linguistics*, *22*(1), 27–57. https://doi.org/10.1093/applin/22.1.27

Rouse, W. B., & SH, R. (1979). Measures of Complexity of Fault Diagnosis Tasks [GS Search]. *IEEE Transactions on Systems, Man, and Cybernetics*, *9*(11), 720–727. https://doi.org/10.1109/TSMC.1979.4310112

Santos, P., Carvalho, L. S. G., Oliveira, E., & Fernandes, D. (2019). Classificação de dificuldade de questões de programação com base na inteligibilidade do enunciado [GS Search]. *Simpósio Brasileiro de Informática na Educação (SBIE)*, *30*(1), 1886. https://doi.org/10.5753/cbie.sbie.2019.1886

Scarton, C. E., & Aluísio, S. M. (2010). Análise da Inteligibilidade de textos via ferramentas de Processamento de Língua Natural: adaptando as métricas do Coh-Metrix para o Por-

tuguês [GS Search]. *Linguamática*, *2*(1), 45–61. https://linguamatica.com/index.php/linguamatica/article/view/44

Schölkopf, B., Smola, A. J., Williamson, R. C., & Bartlett, P. L. (2000). New support vector algorithms [GS Search]. *Neural computation*, *12*(5), 1207–1245. https://doi.org/10.1162/089976600300015565

Silva, É. S., Carvalho, L. S. G., Oliveira, D. B. F., Oliveira, E. H. T., Lauschner, T., P., L. M. A., & Pereira, F. D. (2022). Previsão de indicadores de dificuldade de questões de programação a partir de métricas extraídas do código de solução [GS Search]. *Anais do XXXIII Simpósio Brasileiro de Informática na Educação (SBIE)*. https://doi.org/10.5753/sbie.2022.224724

Vihavainen, A., Luukkainen, M., & Pärtel, M. (2013). Test my code: An automatic assessment service for the extreme apprenticeship method [GS Search]. *2nd International Workshop on Evidence-based Technology Enhanced Learning*, 109–116. https://doi.org/10.1007/978-3-319-00554-6_14

Viloria, A., Lezama, O. B. P., & Mercado-Caruzo, N. (2020). Unbalanced data processing using oversampling: Machine learning [GS Search]. *Procedia Computer Science*, *175*, 108–113. https://doi.org/10.1016/j.procs.2020.07.018

Wang, T., Su, X., Ma, P., Wang, Y., & Wang, K. (2011). Ability-training-oriented automated assessment in introductory programming course [GS Search]. *Computers & Education*, *56*(1), 220–226. https://doi.org/10.1016/j.compedu.2010.08.003

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering [GS Search]. *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, 1–10. https://doi.org/10.1145/2601248.2601268