Correlação Entre Complexidade e Dificuldade de Questões de Programação em Juízes Online

Jackson Fernandes[1], Leandro S. G. Carvalho[1], David B. F. Oliveira[1], Elaine H. T. Oliveira[1], Filipe Dwan Pereira[2], Tanara Lauschner[1]
{jackson.fernandes,galvao,david,elaine,tanara}@icomp.ufam.edu.br,filipe.dwan@ufrr.br
[1]Instituto de Computação, Universidade Federal do Amazonas, Manaus/AM
[2]Departamento de Ciência da Computação, Universidade Federal de Roraima, Boa Vista/RR

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 essa tecnologia é determinar a dificuldade das questões cadastradas no ambiente. 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 através 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%.

CCS CONCEPTS

• **Social and professional topics** → Computing education.

PALAVRAS-CHAVE

Juízes Online, Dificuldade de questões, Métricas de dificuldade, Métricas de complexidade, Aprendizado de máquina

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 [2]. 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 [11, 21, 22].

No entanto, como o uso de JO originou-se de competições de programação, tornou-se necessário adaptá-los para uso no contexto educacional. Nesse sentido, Francisco et al. [11] elencaram os principais requisitos funcionais e não-funcionais indispensáveis para a construção de um JO. Os principais requisitos funcionais destacados foram: *feedback*, *integração do sistema com os cursos*, *análise do desempenho geral dos estudantes* e *possibilidade de diferentes tipos de atividades*. Já os principais requisitos não-funcionais identificados foram: *usabilidade*, *escalabilidade* e *disponibilidade*. 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.

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 da disciplina seleciona do banco uma lista de questões a serem resolvidas em um espaço de tempo limitado. Geralmente, as questões são embaralhadas ou sorteadas, a fim de evitar 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 [11, 18, 25].

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 esses atributos, chamados de *métricas de complexidade* [9, 13], influenciam no desempenho obtido pelos alunos ao tentarem resolver uma questão de escrita de código de programação.

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?

EduComp'23, Abril 24-29, 2023, Recife, Pernambuco, Brasil (On-line)

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 de *Spearman* 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 apresenta as definições de complexidade e dificuldade, além de uma breve revisão 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 exibe os resultados obtidos tanto para o estudo realizado sobre correlação entre variáveis dependentes e independentes, quanto para os 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, são aprofundados os conceitos de complexidade e dificuldade, além de ser feita uma breve revisão da literatura sobre as estratégias de resolução já feitas para o tema abordado.

2.1 Diferença entre complexidade e dificuldade

O primeiro ponto a ser definido é estabelecer a diferença entre complexidade e dificuldade. De acordo com a definição adotada por Effenberger et al. [9], a *complexidade* diz respeito a características intrínsecas da tarefa que influenciam o desempenho, mas são independentes do contexto de aplicação, tais como pessoas resolvendo o 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* [17], *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 [10]. 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.

2.2 Trabalhos relacionados

O problema de estabelecer uma relação entre complexidade e dificuldade em questões de programação é objeto de vários estudos. Effenberger et al. [9] 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 [10] 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. [26] propõem um método para classificar a dificuldade de questões de programação com base em métricas de inteligibilidade textual. Eles utilizaram o índice *Flesch* e 41 outras métricas extraídas pela ferramenta *Coh-Metrix-Port*. A análise abrangeu 450 questões, respondidas por 800 alunos em uma disciplina de introdução à programação, utilizando a taxa de acerto como indicador de dificuldade. 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.

Por fim, de Lima et al. [7] desenvolveram um preditor capaz de classificar a dificuldade de uma questão a partir de métricas de código, estudo semelhante ao deste trabalho. Nessa pesquisa, a taxa de acerto foi usada como indicador de dificuldade, enquanto que para as métricas de código, foram usados 92 atributos. A base usada para treino e teste foi composta por 404 questões aplicadas em avaliações presenciais de cursos de programação introdutória (IPC) cadastradas no sistema *CodeBench*. Como resultado, obteve-se um *f1-score* de 0,84 usando classificação por facilidade, e 0,82 em classificação por dificuldade. A grande limitação deste trabalho foi a exploração de apenas uma métrica de dificuldade.

Apesar dos resultados obtidos pelos estudos citados terem sido significativos, algumas questões ficaram em aberto. Em Effenberger et al. [9], não foi feito um estudo aprofundado de como a combinação de métricas de dificuldade poderia ser feita. Já no estudo de Elnaffar [10], 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 de Lima et al. [7], não foi feita uma análise de desempenho com outras métricas de dificuldade, tais como o número de submissões feitas para uma dada questão, ou o tempo necessário para codificar a solução. Além disso, tanto nos estudos de de Lima et al. [7] quanto no trabalho de Santos et al. [26], seria importante também verificar o desempenho dos modelos de regressão utilizando outras métricas de dificuldade, pois apesar de de Lima et al. [7] 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.

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 Geração da base de questões

A base de questões utilizada foi extraída do JO *CodeBench*, o qual é 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 Universidade Federal do Amazonas. 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). Em uma avaliação presencial, além do tempo reduzido, os alunos são supervisionados pelo professor e tutor da disciplina, na tentativa de reduzir a prática de cola. Como as avaliações da disciplina foram feitas de forma remota durante o ano letivo de 2020 (realizado entre 2020 e 2021), resolveu-se remover da base os dados das turmas desses anos por haver possibilidade de cola.

Ao final dessa extração, 709 questões foram selecionadas. No entanto, como visto por de Lima et al. [7], 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 de Lima et al. [7] 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.

3.2 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*[2] disponibilizados por de Lima et al. [7]. Ao final, a complexidade de cada código da base seria descrita por meio de 130 atributos[3]. 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 costumam ser abordados em cursos introdutórios de programação. 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 de Lima et al. [7]: (1) solução correta; (2) menor número de submissões; (3) menor número de testes; e (4) menor número de erros de sintaxe.

3.3 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 de maneira sucinta 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*. Das variáveis definidas durante esta pesquisa, é importante destacar a motivação para a criação de algumas:

- ***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***: tem como objetivo 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.

3.4 Categorização das variáveis dependentes

Com a finalidade de responder à QP2, um dos estudos necessários é treinar modelos de aprendizado de máquina que possam prever

[1]Os códigos utilizados nos experimentos estão disponíveis em: https://github.com/Jacksonfern/codebench-analytics

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

[3]A descrição das variáveis independentes pode ser encontrada em: encurtador.com.br/lyzRS

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 [7].
Número de submissões	num_submissoes	Quantidade média de submissões feitas para uma questão. Essa métrica é definida como *attempts* por Pereira et al. [23].
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. [28].
Número de consultas	num_consultas	Somatório entre o número de submissões e o número de testes da questão. Métrica proposta por Silva et al. [28].
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 et al. [23].
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 et al. [23].
Número de eventos de deleção	num_eventos_del	Média de vezes que a tecla *backspace* ou *del* foi pressionada. Métrica proposta por Pereira et al. [23].
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 et al. [23].

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 definí-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.

Para a *taxa de acerto*, utilizou-se a convenção de "índice de facilidade", do Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira (INEP) [6]. 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.

Já para as outras métricas de dificuldade, a heurística adotada foi classificar as questões baseado 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".

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

Valor	Classificação INEP	Binária	Ternária
$\geq 0,86$	Muito fácil	Fácil	Fácil
$0,61$ a $0,85$	Fácil		Médio
$0,41$ a $0,60$	Médio	Difícil	Difícil
$0,16$ a $0,40$	Difícil		
$\leq 0,15$	Muito difícil		

Um aspecto importante de se observar é o balanceamento entre as classes, pois isso influencia no desempenho dos modelos [29]. 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 pois boa parte das questões possuem taxa de acerto entre 70 e 90%, como é possível observar na Figura 1. Tanto para a classificação binária (Figura 2a) quanto para a ternária (Figura 2b), 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".

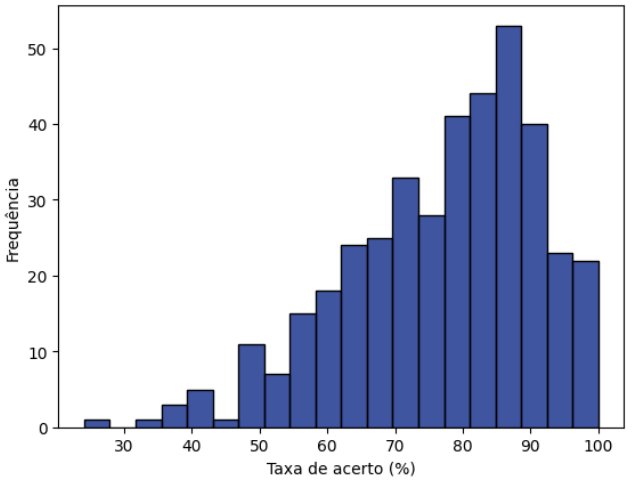


Figura 1: Distribuição de frequência da taxa acerto

3.5 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 [4, 15]. 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 comumente obtêm classificadores super ou sub-ajustados [14].

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:

- *Árvores de regressão* (**RT**)
- *Support vector regression* (**SVR**) [4]
- *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* [3], dado o problema de desbalanceamento das classes para esta métrica. Essa técnica foi escolhida pois 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* [1], com $k = 4$, ou seja, três partições para treino e uma para teste.

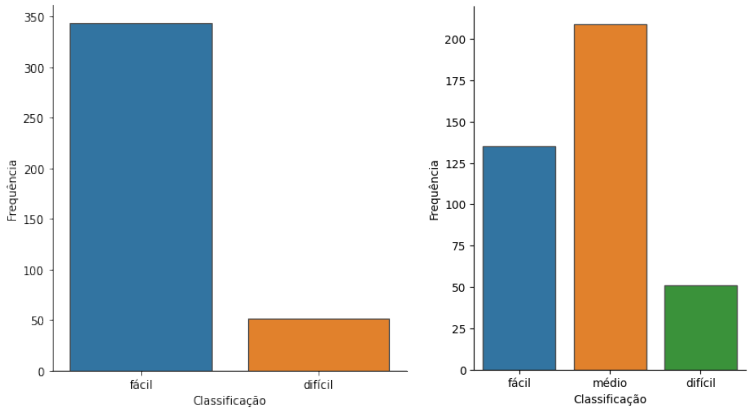
3.6 Métricas para avaliação dos modelos

Uma etapa fundamental no processo de avaliar corretamente um modelo de aprendizagem de máquina consiste em definir bem as métricas a serem utilizadas [20]. Para os modelos de classificação, as métricas mais comumente usadas são *precisão*, *revocação*, *acurácia* e *f1-score*. No entanto, 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* [16]. A desvantagem de usar *macro averaging* é que o cálculo desta métrica não leva em consideração o desbalanceamento entre as classes [16], enquanto que *micro averaging*, apesar de ser atribuído o mesmo peso para cada classificação, as métricas de *precisão*, *revocação* e *acurácia* acabam sendo iguais [12]. É importante notar que, quanto mais balanceada são as classes, mais o valor de *micro averaging* e *macro averaging* são próximos, o que torna irrelevante o modelo a ser escolhido. Portanto, 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 [20], 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 [8].

Como as métricas de dificuldade também foram categorizadas, tornou-se possível usar os modelos de regressão também como preditores de dificuldade. A ideia é que, além da informação da

[4] Além do SVR padrão, também foi testada uma variação chamada de NuSVR. Nessa versão, os vetores de suporte e as margens de erro são controlados por um parâmetro denominado *nu* [27]



(a) Classificação em dois níveis **(b) Classificação em três níveis**

Figura 2: Distribuição de frequência das classes de dificuldade baseado na classificação do INEP para a taxa de acerto

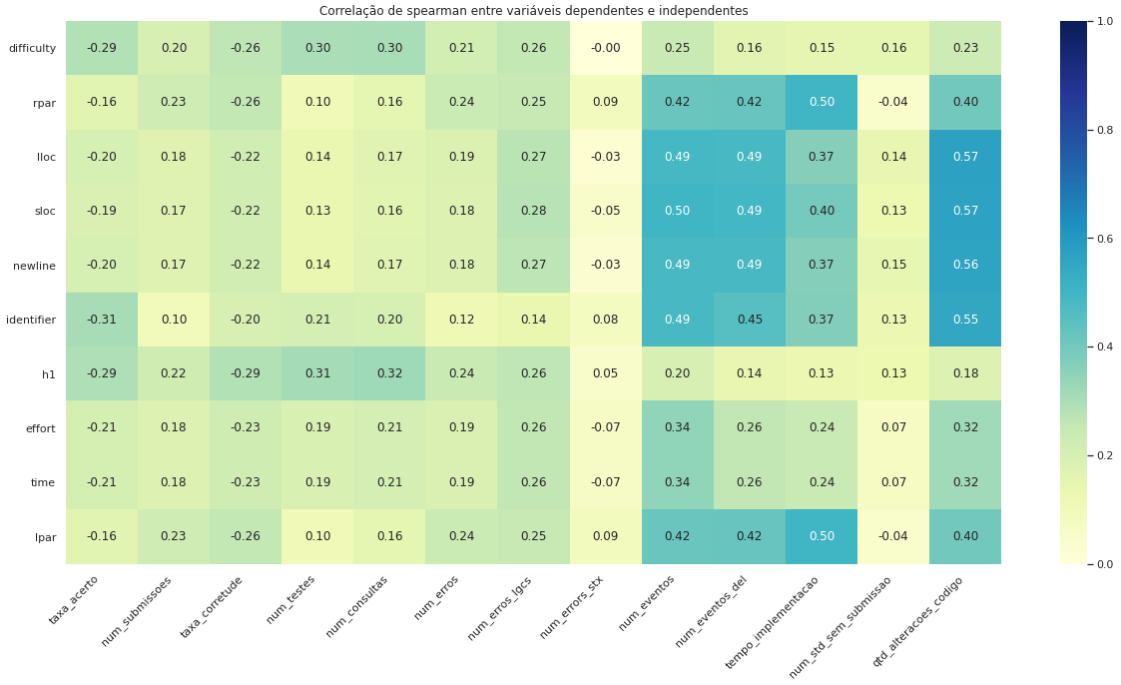


Figura 3: Correlação de *Spearman* entre variáveis independentes e dependentes

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 independentes e dependentes

O primeiro estudo realizado foi a análise da correlação entre as variáveis dependentes e independentes da base. Como os dados não seguem uma distribuição normal, optou-se por utilizar correlação de *Spearman* em vez de *Pearson* [19]. A Figura 3 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

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

e nenhum caso de correlação forte ou perfeita[5]. Considerando apenas as 10 métricas da Figura 3, 85% das correlações foram fracas ou inexistentes, 15% de correlações moderadas e nenhum caso de correlação forte ou perfeita. Portanto, pode-se concluir que nenhum dos atributos individuais de complexidade de código considerados conseguem indicar a dificuldade de uma questão, respondendo assim a QP1.

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}.

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.

As Tabelas 4 e 5 mostram o desempenho obtido pelos modelos de regressão quando as predições, originalmente valores contínuos, foram categorizadas em dois e três níveis de dificuldade, 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 6 e 7, 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

[5]A intensidade das correlações é baseada em Dancey and Reidy [5].

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

Tabela 5: Resultados para o modelo de regressão com as predições categorizadas 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

níveis, a melhor métrica foi o *tempo de implementação*, com *f1-score* de 67%, seguida do *número de eventos de deleção*, com 61%.

Tabela 6: 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

Tabela 7: 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 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.

5.1 Análise da correlação entre variáveis dependentes e independentes

Baseado nas correlações mostradas na Figura 3, 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

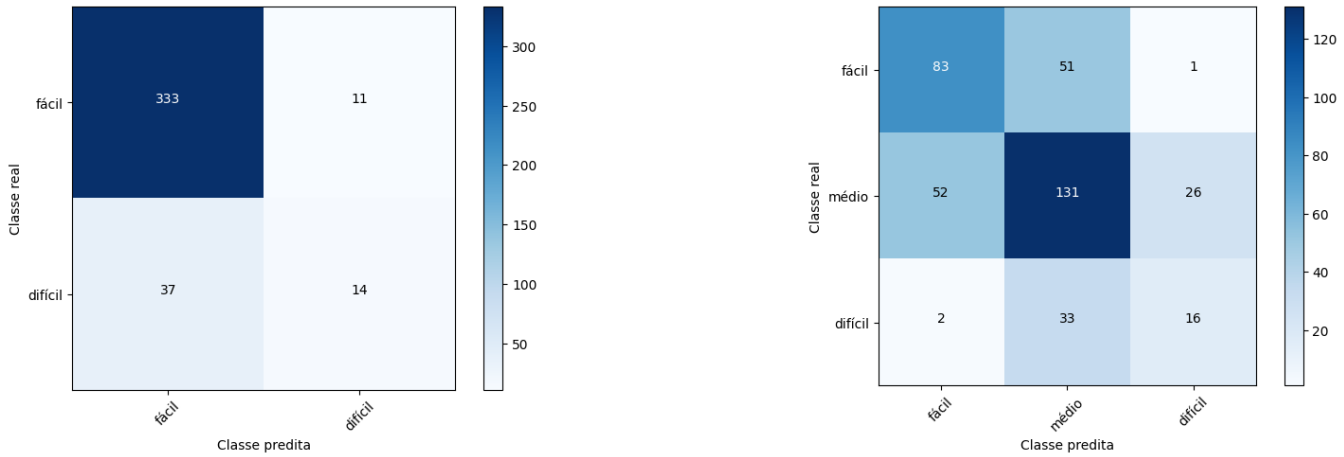
Uma das perguntas a se responder é por que a predição com dois níveis de dificuldade obteve desempenho muito superior àquela com três níveis. Essa pergunta pode ser respondida quando se observa a matriz de confusão gerada para esses casos. Para a taxa de acerto, a matriz de confusão para a classificação com duas categorias é mostradas na Figura 4a. 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.

Para a categorização em três níveis, a matriz de confusão é mostrada na Figura 4b. 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.

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.

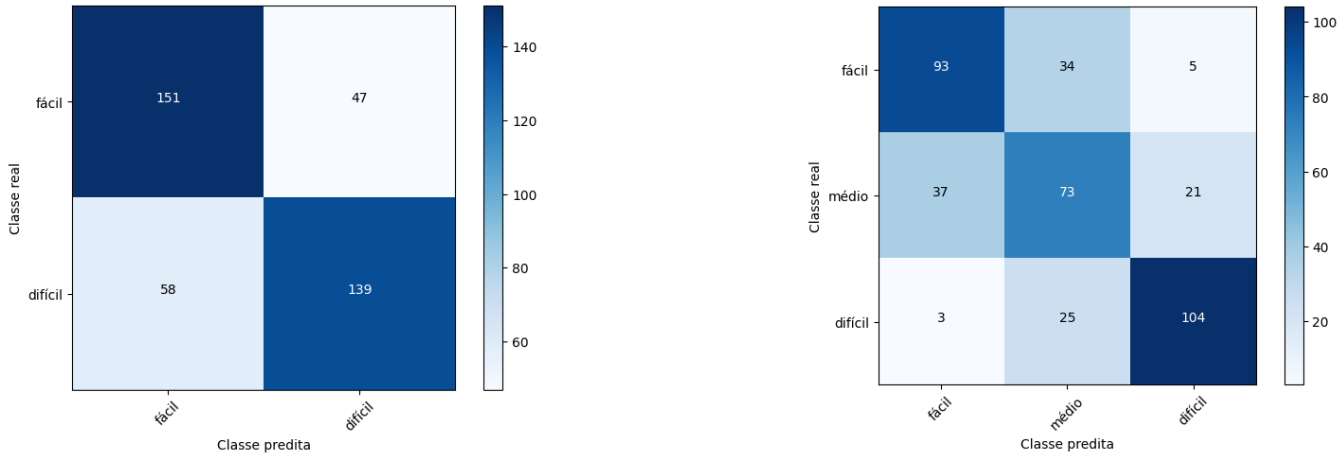
5.3 Classificação em dois níveis e três níveis para outras métricas de dificuldade

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 5a. Como é possível observar, 73% das predições foram corretamente feitas.



(a) Classificação binária **(b) Classificação ternária**

Figura 4: Matriz de confusão para a métrica de taxa de acerto



(a) Classificação binária **(b) Classificação ternária**

Figura 5: Matriz de confusão para a métrica de tempo de implementação

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 o modelos de classificação com três classes (Figura 5b), 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.

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 [24], 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 com 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 através de regressão.

Para a correlação de *Spearman* entre métricas individuais de complexidade de código e métricas de dificuldade, observou-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 fazem 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.

Com os estudos feitos, foi possível perceber 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.

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.

REFERÊNCIAS

[1] Davide Anguita, Alessandro Ghio, Sandro Ridella, and Dario Sterpi. 2009. K-Fold Cross Validation for Error Rate Estimate in Support Vector Machines. In *DMIN*.

291–297.
[2] Leandro Silva Galvão Carvalho, David Braga Fernandes Oliveira, and Bruno Gadelha. 2016. Juiz online como ferramenta de apoio a uma metodologia de ensino híbrido em programação. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*, Vol. 27. SBC, Uberlândia, MG, 140–149.
[3] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16, 321–357.
[4] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794.
[5] C.P. Dancey and J. Reidy. 2007. *Statistics Without Maths for Psychology*. Pearson/Prentice Hall.
[6] Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira. 2017. *Relatório Síntese de Área – Ciência da Computação (Bacharelado/Licenciatura)*. Technical Report. Retrieved August 27, 2022 from https://download.inep.gov.br/educacao_superior/enade/relatorio_sintese/2017/Ciencia_da_Computacao.pdf
[7] Marcos Avner Pimenta de Lima, Leandro Silva Galvão Carvalho, Elaine Harada Teixeira Oliveira, David Braga Fernandes Oliveira, and Filipe Dwan Pereira. 2021. Uso de atributos de código para classificação da facilidade de questões de codificação. In *Anais do Simpósio Brasileiro de Educação em Computação*. SBC, SBC, online, 113–122.
[8] Alessandro Di Bucchianico. 2008. Coefficient of determination (R 2). *Encyclopedia of statistics in quality and reliability* 1.
[9] Tomáš Effenberger, Jaroslav Cechák, and Radek Pelánek. 2019. Difficulty and Complexity of Introductory Programming Problems.
[10] Said Elnaffar. 2016. Using software metrics to predict the difficulty of code writing questions. In *2016 ieee global engineering education conference (educon)*. IEEE, 513–518.
[11] Rodrigo Elias Francisco, Ana Paula Laboissière Ambrósio, Cleon Xavier Pereira Junior, and Márcia Aparecida Fernandes. 2018. Juiz online no ensino de CS1-lições aprendidas e proposta de uma ferramenta. *Revista Brasileira de Informática na Educação* 26, 03, 163.
[12] Margherita Grandini, Enrico Bagli, and Giorgio Visani. 2020. Metrics for multi-class classification: An overview. arXiv:2008.05756 [stat.ML]
[13] Peng Liu and Zhizhong Li. 2012. Task complexity: A review and conceptualization framework. *International Journal of Industrial Ergonomics* 42, 6, 553–568.
[14] Ana Carolina Lorena and André CPLF Carvalho. 2007. Uma introdução às support vector machines. *Revista de Informática Teórica e Aplicada* 14, 2, 43–67.
[15] Scott M Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. 2020. From local explanations to global understanding with explainable AI for trees. *Nature machine intelligence* 2, 1, 56–67.
[16] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schutze. 2008. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, England.
[17] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4, 308–320.
[18] V. Meisalo, E. Sutinen, and S. Torvinen. 2004. Classification of exercises in a virtual programming course. In *34th Annual Frontiers in Education, 2004. FIE 2004.* S3D–1.
[19] Leann Myers and Maria J. Sirois. 2006. Spearman Correlation Coefficients, Differences between. In *Encyclopedia of Statistical Sciences*. John Wiley & Sons, Ltd.
[20] M Z Naser and Amir H Alavi. 2021. Error metrics and performance fitness indicators for artificial intelligence and machine learning in engineering and sciences. *Archit. Struct. Constr.*
[21] Rodrigo Barros Paes, Romero Malaquias, Márcio Guimarães, and Hyggo Almeida. 2013. Ferramenta para a Avaliação de Aprendizado de Alunos em Programação de Computadores. In *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*, Vol. 2. SBC, Campinas, SP, 203–212.
[22] Fillipi D Pelz, Elieser A Jesus, and André LA Raabe. 2012. Um mecanismo para correção automática de exercícios práticos de programação introdutória. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educaçao-SBIE)*, Vol. 23.
[23] Filipe Dwan Pereira, Samuel C Fonseca, Elaine HT Oliveira, Alexandra I Cristea, Henrik Bellhäuser, Luiz Rodrigues, David BF Oliveira, Seiji Isotani, and Leandro SG Carvalho. 2021. Explaining Individual and Collective Programming Students' Behavior by Interpreting a Black-Box Predictive Model. *IEEE Access* 9, 117097–117119.
[24] Filipe Dwan Pereira, Hermino BF Junior, Luiz Rodriguez, Armando Toda, Elaine HT Oliveira, Alexandra I Cristea, David BF Oliveira, Leandro SG Carvalho, Samuel C Fonseca, Ahmed Alamri, et al. 2021. A recommender system based on effort: Towards minimising negative affects and maximising achievement in CS1 learning. In *International Conference on Intelligent Tutoring Systems*. Springer, 466–480.
[25] André Prisco, Rafael dos Santos, Silvia Botelho, Neilor Tonin, and Jean Bez. 2017. Using information technology for personalizing the computer science teaching. In *2017 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–7.
[26] Pedro Santos, Leandro Silva Galvão Carvalho, Elaine Oliveira, and David Fernandes. 2019. Classificação de dificuldade de questões de programação com base na inteligibilidade do enunciado. In *Simpósio Brasileiro de Informática na Educação (SBIE)*, Vol. 30. SBC, Brasília, DF, 1886.
[27] Bernhard Schölkopf, Alex J Smola, Robert C Williamson, and Peter L Bartlett. 2000. New support vector algorithms. *Neural computation* 12, 5, 1207–1245.
[28] Élrik Souza Silva, Leandro S. Galvão Carvalho, David B. F. Oliveira, Elaine H. T. Oliveira, Tanara Lauschner, Lima Marcos A. P., and Filipe Dwan Pereira. 2022. Previsão de indicadores de dificuldade de questões de programação a partir de métricas extraídas do código de solução. In *Anais do XXXIII Simpósio Brasileiro de Informática na Educação (SBIE)*. SBC, Manaus, AM.
[29] Amelec Viloria, Omar Bonerge Pineda Lezama, and Nohora Mercado-Caruzo. 2020. Unbalanced data processing using oversampling: Machine learning. *Procedia Computer Science* 175, 108–113.