Classificação de dificuldade de questões de programação com base em métricas de código

Marcos A. P. de Lima[1], **Leandro S. G. Carvalho**[1], **Elaine H. T. Oliveira**[1], **David B. F. Oliveira** [1], **Filipe Dwan Pereira**[2]

Instituto de Computação – Universidade Federal Amazonas (UFAM)

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

{marcos.lima,david,galvao,elaine}@icomp.ufam.edu.br,filipe.dwan@ufrr.br

Resumo. *No ensino de programação utilizando juízes online, é importante apresentar ao estudante questões de programação em nível crescente de dificuldade, bem como equilibrar os níveis das questões sorteadas pelo ambiente durante avaliações. Este trabalho propõe e valida um método para classificar automaticamente a dificuldade de questões de programação com base em métricas extraídas de códigos de soluções cadastrados pelos(as) instrutores(as). Ao todo, foram analisadas 354 questões implementadas em Python e elaboradas para turmas de introdução à programação ministradas entre 2017 e 2019. Verificou-se que o uso de métricas extraídas de código pode sim ser utilizado para estimar a dificuldade de questões, além de elucidar alguns aspectos da dificuldade encontrada pelos estudantes.*

Abstract. *While learning how to program using online judges, students should be presented to questions in an increasingly difficult level. Besides, randomized exams should be fair. This work presents a method to classify the difficulty of programming questions based on metrics extracted from instructor's solution code. This work analysed 354 questions in Python and used in some Introduction to Programming (CS1) classes taught from 2017 to 2019. It was found that the use of metrics extracted from code can indeed be used to estimate the difficulty of questions, in addition to elucidating some aspects of the difficulty encountered by students.*

1. Introdução

Ambientes de correção automática de código (ACAC), também conhecidos como *juízes online*, têm sido amplamente empregados no ensino de programação [Francisco and Ambrosio 2015, Pereira et al. 2020]. Embora haja uma variedade de tipos, consideremos aqui os que disponibilizam *questões de codificação*. Para resolvê-las, os estudantes devem escrever um código de solução e submeter à avaliação automática do ambiente. A solução é considerada correta se o código submetido gerar exatamente as saídas esperadas para um conjunto de casos de testes previamente cadastrados pelo(a) instrutor(a) da disciplina, mas ocultos aos estudantes. Caso contrário, a solução é considerada incorreta.

Tomemos como cenário um laboratório de computadores, onde uma avaliação de conhecimentos sobre programação é aplicada a uma classe de ensino superior presencial. Os estudantes resolvem as questões por meio de um ACAC aberto em seus navegadores. Poucos centímetros separam um colega do outro, mas também é possível que colegas sentados em extremidades opostas do laboratório se comuniquem facilmente por meio de

DOI: 10.5753/cbie.sbie.2020.1323

aplicativos de conversa abertos em outra aba do navegador, enquanto o(a) instrutor(a) tira eventuais dúvidas de um terceiro estudante.

Em uma configuração tão propícia ao comportamento desonesto, o(a) instrutor(a) cria um banco de questões que – segundo sua opinião pessoal – têm níveis semelhantes de dificuldade. A partir desse banco, o ACAC pode sortear diferentes questões para diferentes estudantes, minimizando a possibilidade da troca de código entre eles durante a avaliação. Porém, quando se trata de programação introdutória, uma simples diferença entre uma questão e outra, como por exemplo a inclusão de um `else`, pode prejudicar a equidade da avaliação.

Dessa forma, se pudermos estimar a dificuldade de uma questão de codificação *antes* de ela ser aplicada em uma avaliação, teremos condições de configurar o ACAC para sortear questões com níveis semelhantes de dificuldade. Na literatura, algumas abordagens de estimação da dificuldade desse tipo de questão se baseiam na Teoria de Resposta ao Item (TRI) [Zaffalon et al. 2019], e outras, em métricas de legibilidade do enunciado (no sentido de facilidade de leitura) [Santos et al. 2019]. Na abordagem por TRI, há o inconveniente de que a dificuldade só pode ser determinada depois que a questão (item) já tenha sido resolvida por um conjunto expressivo de estudantes. Na abordagem por legibilidade, enunciados curtos interferem na extração de métricas, pois se baseia na contagem de sílabas, palavras e sentenças.

Neste trabalho, consideramos que a *dificuldade* de uma questão de codificação é expressa pela *taxa de acerto*, ou seja, pela razão entre o número de estudantes que submeteram códigos que passaram com sucesso em todos os casos de testes e o número de estudantes que tentaram submeter, pelo menos uma vez, um código de solução. Dessa forma, abordamos aqui o problema de prever a dificuldade de uma nova questão de codificação inserida no ACAC a partir do seu código de solução, que é um exemplo de código correto informado pelo(a) instrutor(a) no momento da criação do exercício no ACAC. O objetivo é verificar a possibilidade de determinar, com um grau de segurança, se uma questão é difícil ou não, por meio de métricas extraídas a partir dos códigos de solução cadastrados pelos instrutores. Como exemplo dessas métricas, temos: número de atribuições, número de variáveis, número de operadores, etc. Ao todo, foram obtidas 91 métricas, a maioria extraída do código de solução do instrutor, enquanto que algumas foram derivadas.

As seguintes questões de pesquisa nortearam este trabalho:

QP1: Até que ponto métricas de código extraídas da solução de um instrutor podem explicar a dificuldade observada pelos estudantes durante a resolução de questões?

QP2: Como a dificuldade de novas questões de programação pode ser classificada por métricas extraídas dos códigos de solução do instrutor?

2. Trabalhos relacionados

Uma forma de classificar a dificuldade de uma questão de codificação é pedir que o próprio instrutor indique, com base em sua experiência pessoal, o nível de dificuldade na tela de cadastro. Por exemplo, [Francisco and Ambrosio 2015] propõem um ambiente no qual os exercícios são classificados em 03 níveis de dificuldade; já no ambiente proposto por [Llana et al. 2012], há 05 níveis de classificação; por sua vez, [Denny et al. 2015] classificam a dificuldade usando uma rubrica de 06 níveis. Porém, a classificação manual apresenta alguns inconvenientes: subjetividade na interpretação dos rótulos dos níveis de dificuldade, tempo consumido na tarefa e disponibilidade de avaliadores para trazer uma segunda ou terceira opinião. Em resumo, não é uma abordagem escalável para ambientes em que novas questões de codificação são criadas e disponibilizadas constantemente.

Adicionalmente, avaliadores humanos podem não concordar entre si. Nesse sentido, [Sheard et al. 2011] verificaram que somente 43% dos tutores de uma disciplina de programação concordavam sobre o nível de dificuldade de um conjunto de 252 questões, mesmo após debaterem as classificações conflitantes, em uma escala de 03 pontos. Ainda que haja um consenso entre os avaliadores, talvez o resultado não corresponda ao valor que se deseja medir. Como evidenciado em [Meisalo et al. 2004], a dificuldade estimada pelo instrutor costuma estar aquém da dificuldade sentida pelos estudantes.

Recentemente, alguns trabalhos têm buscado analisar os dados gerados pelos próprios usuários dos ACACs como forma de estimar a dificuldade dos itens. Por usuários, entenda-se aqui tanto *instrutores*, ao cadastrarem questões, quanto *estudantes*, ao deixarem um rastro de registros na tentativa de resolver questões propostas. Por exemplo, [Zaffalon et al. 2019] compararam dois modelos de estimar o desempenho de estudantes: Elo e TRI. O modelo Elo foi desenvolvido por [Pelánek 2016] e empregado originalmente para classificar jogadores de xadrez, mas pode ser usado em sistemas de aprendizagem quando interpretamos a resposta do estudante a um item como uma partida entre o estudante e o item. Por sua vez, a TRI estima a probabilidade do indivíduo responder corretamente um item, neste caso, de natureza dicotômica (certo/errado). Os resultados são promissores para bancos de questões com itens já resolvidos, mas os modelos não podem ser aplicados para estimar a dificuldade de novas questões inseridas.

Por outro lado, a complexidade linguística do enunciado (frases longas e palavras incomuns) pode afetar a compreensão do estudante sobre o problema a ser resolvido e, consequentemente, sua habilidade em respondê-lo corretamente [Simon et al. 2013]. Nesse sentido, em um trabalho anterior [Santos et al. 2019], procuramos correlacionar métricas de legibilidade do enunciado com a dificuldade das questões. Porém, não foi possível encontrar correlação significativa. Conseguimos apenas confirmar, por meio dos dados, a noção intuitiva de que embora questões difíceis de ler normalmente produzam baixa taxa de acerto, questões fáceis nem sempre produzem altas taxas de acerto. O inconveniente dessa abordagem é que enunciados curtos interferem na extração de métricas, pois elas se baseiam na contagem de sílabas, palavras e sentenças.

Por sua vez, o presente artigo utiliza métricas extraídas do código de solução do instrutor para estimar o nível de dificuldade. Ainda há poucos trabalhos na literatura que tenham seguido essa abordagem. Por exemplo, [Elnaffar 2016] encontrou forte correlação ($>$0,9) entre a dificuldade de questões de programação e algumas métricas extraídas do código de solução (complexidade ciclomática, profundidade média dos blocos, número de operadores e número de chamadas de métodos em Java). Contudo, os resultados são limitados a uma amostra de apenas 10 questões, de uma classe de tamanho não explicitado, ensinada por um único instrutor. Já [Effenberger et al. 2019] analisaram 04 conjuntos de questões de programação introdutória, sendo um deles semelhante ao contexto deste trabalho, programação Python, composto de 73 questões, resolvidas por 2.000 estudantes em 10.700 tentativas. Os autores distinguem dois conceitos: *complexidade*, que depende apenas do código da questão, podendo ser medida sem consultar dados de desempenho; e *dificuldade*, estimada com base no comportamento dos estudantes ao resolverem a questão. Os autores estimaram a complexidade a partir do número de conceitos de programação abordados, do número de estruturas de controle de fluxo e do número total de linhas de código. Já a dificuldade foi estimada a partir da média de 04 métricas: taxa de erro e medianas do tempo de acerto, número de edições e número de execuções. Por fim, encontraram uma correlação moderada (ρ_S = 0,73) entre complexidade e dificuldade.

3. Contexto das questões de programação utilizadas na pesquisa

Esta pesquisa se baseou nos dados extraídos do ACAC Codebench[1] da Universidade Federal do Amazonas. Foram consideradas apenas questões resolvidas por estudantes da disciplina de Introdução à Programação de Computadores (IPC), entre 2017 e 2019. Ela é ministrada para 17 cursos de graduação nas áreas de engenharia e ciências exatas dessa instituição. Dentre as questões disponíveis, foram analisadas apenas aquelas usadas em *exames presenciais*, pois, durante sua aplicação, não era permitido que os estudantes compartilhassem respostas entre si.

Das questões disponíveis, 653 são em Python e foram aplicadas em *exames presenciais*. [Effenberger et al. 2019] afirmam que a tarefa de estimar a dificuldade requer que uma quantidade suficiente de dados tenha sido coletado, e que a quantidade necessária pode variar dependendo da métrica adotada. Logo, das 653 questões foram selecionadas somente aquelas que foram resolvidas por 20 ou mais estudantes, a fim de reduzir o viés dos dados de uso da questão e obter métricas de dificuldade mais estáveis. Dessa forma, restaram 354 questões.

Contudo, o campo "código de solução do instrutor" foi adicionado ao ambiente somente em 2019. Por isso, do conjunto de questões selecionadas, somente 278 apresentavam um código de solução elaborada pelo instrutor. Como as questões utilizadas em disciplinas de programação introdutória são normalmente diretas, não existe tanta diferença entre as soluções dos instrutores e estudantes. Dessa forma, completamos a amostra com os códigos do estudantes que acertaram as questões. Com efeito, em caso de mais de uma solução correta, adotamos os seguintes critérios de escolha, em ordem de prioridade: 1) menor número de submissões feitas pelo estudante; 2) menor número de execuções feitas pelo estudante; e 3) menor quantidade de erros obtidos.

4. Metodologia

4.1. Variável dependente (alvo)

Na literatura, não há consenso quanto à definição de *dificuldade* de questões de programação. Segundo [Effenberger et al. 2019] existem duas dimensões de dificuldade que estão disponíveis na maioria dos contextos são *taxa de acerto* e *tempo médio de resolução do problema*. Outra possível *variável dependente* elencada para este trabalho foi o *número médio de submissões* necessários para acertar dada questão.

Durante a análise dos *arquivos de logs* das tentativas dos estudantes, observou-se que alguns estudantes não chegam a sequer testar seu código antes de submetê-lo à correção automática e que, por vezes, submetem um código não funcional, aumentando assim o número de submissões contabilizadas. Ademais, alguns estudantes, mesmo após acertar a questão, continuavam a realizar submissões para correção automática. Como consequência, o uso do *número médio de submissões* como *variável alvo* poderia levar a uma classificação errônea da real dificuldade das questões.

O *tempo médio de resolução* de uma questão, por sua vez, foi calculado por meio de um *arquivo de log* que registra com data e hora, as interações do estudante (eventos) com o ACAC. Observou-se que durante a resolução de um problema, existiam longos intervalos sem nenhuma interação. Tais intervalos podem significar um momento de distração, alternância para outra questão, ou até mesmo que o estudante desistiu da

[1] `http://codebench.icomp.ufam.edu.br/`

questão. Consequentemente, mensurar de forma precisa o real *tempo de resolução* seria muito dificultoso e provavelmente impreciso.

Desse modo, adotou-se como *variável dependente* para mensurar a dificuldade a *taxa de acerto*, aqui expressa pela razão entre o número de alunos que conseguiram solucionar um dado problema e o total de alunos que tentaram resolver o problema. Não foram consideradas as tentativas em que o código do estudante não era funcional.

4.2. Variáveis independentes (preditoras)

Foram utilizadas como *variáveis independentes* um conjunto de métricas extraídas do código de solução elaborado do instrutor e outras métricas geradas por meio de um processo de *engenharia de atributos*.

No processo de extração de métricas de código, foram utilizados dois módulos da linguagem Python. O módulo *Radon*[2] foi usado para extrair métricas de software [Halstead 1977], "métricas baseadas em tamanho" (linhas de código, linhas lógicas, linhas em branco e comentários) e também a *complexidade ciclomática* total do código [McCabe 1976]. Por meio do módulo *Tokenize*[3], foi feita uma *análise léxica* do código, gerando então *tokens*, que posteriormente foram transformados em "métricas derivadas". As "métricas derivadas" estão relacionadas com a quantidade de ocorrência de: estruturas condicionais e repetição, operadores, comandos de importação, funções embutidas (*built-in*), constantes e palavras-chave (*keywords*). As estruturas condicionais e de repetição foram contabilizadas individualmente (quantidade de `ifs`, `whiles`, `elifs`, etc.) e também em sua totalidade. Os operadores, por sua vez, além de serem contabilizados individualmente, também foram agrupados por categoria (*aritméticos*, *lógicos*, *relacionais*, etc.). Por fim, as funções embutidas foram contabilizadas em sua totalidade, com exceção das funções `input` e `print` que, por serem funções relacionadas com as entradas e saídas das questões, foram então contabilizadas de forma separada. A listagem completa de métricas extraídas está disponível na paǵina do Extrator[4].

Após a extração das métricas, foi realizada uma *engenharia de atributos*, com o objetivo de gerar o máximo de informações distintas e que pudessem ser utilizadas na classificação das questões. Foram criadas variáveis *booleanas* indicando se no código analisado haviam estruturas condicionais, estruturas de repetição, operadores, constantes, comandos de importação, palavras-chave, etc. Além disso, algumas variáveis foram derivadas diretamente das "métricas baseadas em tamanho": *média de identificadores por linha de código*, *média de caracteres por identificador*, por exemplo.

Como as questões abordadas foram utilizadas somente em disciplinas introdutórias de programação, é normal a ausência de algumas construções mais avançadas. Portanto, foram desconsideradas as variáveis que apresentavam variância zero, como por exemplo *operadores bit-a-bit*, *classes* e *expressões lambda*. Ao final de todo o processo, restaram 91 métricas a serem utilizadas como *variáveis preditoras*.

4.3. Discretização da taxa de acerto

Como a variável dependente *taxa de acerto* é contínua, ela foi discretizada para a tarefa de classificação. Isso foi feito com base no "índice de facilidade", adotado pelo Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira (INEP) no Enade

[2] `https://pypi.org/project/radon/`
[3] `https://docs.python.org/3/library/tokenize.html`
[4] `https://github.com/marcosmapl/codebench-extractor`

[INEP 2017]. Esse índice classifica as questões em cinco níveis de facilidade por meio da *taxa de acerto*. Nossa abordagem foi mapear os cinco níveis em apenas em dois, de forma a obter uma tarefa de classificação binária (veja Tabela 1).

Tabela 1. Discretização da taxa de acerto

Taxa de Acerto	Classificação Inep	Classificação Binária
>0,86	Muito Fácil	Não Difícil
0,61 a 0,85	Fácil	
0,41 a 0,60	Média	
0,16 a 0,40	Difícil	Difícil
<0,15	Muito Difícil	

Após a discretização da *taxa de acerto*, observou-se que o conjunto de questões era muito desbalanceado. A classe "Não Difícil" era majoritária com 326 observações (92,1%), enquanto que a classe "Difícil" apresentava apenas 28 observações (7,9%).

4.4. Métricas para avaliação de modelos de classificação

Para avaliar a performance dos modelos gerados era necessário escolher uma métrica principal. *Acurácia*, *precisão*, *revocação* e *f1-score* são algumas das mais utilizadas. A *acurácia* consiste somente no percentual de acertos do modelo, e não é recomendada para cenários em que a base de dados é desbalanceada. A *precisão* fornece uma ideia do quão efetivo é um modelo ao predizer exemplos de uma classe, porém um alto valor de *precisão* não significa uma boa completude. A *revocação* mensura a frequência com que o modelo encontra exemplos de uma classe, sem trazer informação de exatidão na classificação.

Além disso, quando pensamos no problema de classificação da dificuldade de questões utilizadas em exames, prever uma questão "Difícil" como sendo "Não Difícil" pode prejudicar os estudantes lhe dando um conjunto de questões mais complexas que o desejado. Por outro lado, prever uma questão "Não Difícil" como sendo "Difícil" pode acabar por beneficiá-los, gerando exames sem questões desafiadoras. Com o objetivo de ter um equilíbrio na predição de todas as classes, adotamos *f1-score* como métrica principal para mensurar o desempenho dos modelos de classificação, por combinar *precisão* e *revocação* de modo a trazer um único valor que indique a qualidade geral do modelo. Apesar disso, os valores obtidos para as métricas de *precisão*, *revocação* e *acurácia* também serão empregadas, a título de completude.

4.5. Processo de Classificação

A classificação quanto à dificuldade das questões foi feita em duas etapas utilizando uma adaptação da técnica de *ensemble* chamada *stacking* [Wolpert 1992].

Na primeira etapa foram, treinados 11 classificadores-base distintos: *K-Nearest Neighbor* (**knn**), *RandomForest* (**rf**), *ExtraTrees* (**et**), *GradientBoosting* (**gb**), *AdaBoosting* (**ada**), *Support Vector Machine* (**svm**), *Ridge Regression* (**rdg**), *Logistic Regression* (**logr**), *Gaussian Naive Bayes* (**gnb**), *Multi-Layer Perceptron* (**mlp**) e *XGBoost* (**xgb**).

Cada classificador-base foi treinado de forma independente por meio de validação cruzada (*cross-validation*) para minimizar a possibilidade de *overfitting* nos dados. Devido à pouca quantidade de amostras para a classe minoritária, os dados foram divididos em apenas 05 partições (subconjuntos), escolhidas de forma *estratificada* e utilizando o número 42 como *semente de aleatoriedade*. A cada rodada de treino, 04 partições eram

utilizadas para treino e 01 para validação, sendo que tanto as predições quanto as probabilidades estimadas para a partição de validação eram salvas, para uso na etapa seguinte do processo de classificação.

Além disso, com intuito de melhorar os resultados e diminuir o custo computacional durante o treino de cada classificador-base foi feito um processo de seleção de atributos (variáveis independentes). Para a seleção foram avaliados dois métodos, escolhendo sempre o melhor para cada classificador-base. O primeiro método, chamado de `SelectKBest`, faz uso de uma função de pontuação para ranquear e então selecionar um número k de atributos. Como funções de pontuação, foram avaliadas as funções *f_classif* e *mutual_info_classif*, sendo que a primeira função foi melhor em todos os casos. O segundo método, o `SelectFromModel`, usa outro classificador para gerar *scores* para os atributos, e então efetua a seleção dos k melhores. Em ambas as abordagens o parâmetro k foi testado no intervalo entre 01 e 91.

Na segunda etapa, os meta-classificadores foram treinados utilizando as predições e probabilidades estimadas pelos classificadores-base. Utilizando as predições, o meta-classificador que obteve o melhor desempenho foi uma *RandomForest* (**rf2**) que fez uso de 07 das 11 predições geradas pelos classificadores-base. Utilizando as probabilidades, o melhor meta-classificador foi um *XGBoost* (**xgb2**) fazendo uso de apenas 03 das 22 das probabilidades (duas classes para cada um dos onze classificadores-base) estimadas pelos classificadores-base. Para o treino de ambos, novamente utilizou-se a técnica de validação cruzada, com 05 partições estratificadas, e para evitar *vazamento de informação* entre as etapas, o número 43 foi escolhido como *semente de aleatoriedade*, garantindo assim partições diferentes em ambas as etapas.

5. Resultados e Análise

5.1. Correlação e atributos importantes

Foram testadas as correlações entre as *variáveis independentes* (métricas extraídas e geradas a partir do código de solução) com a *variável dependente* (taxa de acerto) segundo a correlação de *Spearman*, pois a grande maioria das variáveis encontradas não apresentavam uma distribuição normal. Observou-se em geral uma correlação fraca, com exceção das variáveis "existem estruturas de repetição" ($\rho_S = -0,41$ e valor-$p = 4,39 \times 10^{-16}$) e "quantidade de estruturas de repetição" ($\rho_S = -0,38$ e valor-$p = 2,27 \times 10^{-13}$), ou seja, ambas as correlações encontradas são estatisticamente significantes.

Verificou-se que variável *quantidade de operadores maior que* foi selecionada como atributo importante em 10 dos classificadores-base e que variáveis relacionadas com estruturas de repetição, em especial a *quantidade de laços while*, tiveram importância para 09 dos classificadores-base. Isso pode significar que em questões introdutórias de programação uma parcela da dificuldade encontrada pelos estudantes pode estar relacionada com problemas envolvendo repetições e/ou intervalos numéricos.

5.2. Resultados da classificação e discussão

Analisando os resultados obtidos (Tabela 2), a *acurácia* obtida em todos os modelos foi alta, o que reforça a não adoção da mesma como métrica principal. Ambos os meta-classificadores apresentaram resultados semelhantes e satisfatórios. A *RandomForest* (**rf2**) obteve *f1-score* de 80%, enquanto que o *XGBoost* (**xgb2**) obteve *f1-score* de 81%, porém utilizando bem menos atributos, e sendo portanto escolhido como modelo final.

Tabela 2. Resultados da classificação.

	knn	rf	et	gb	ada	svc	gnb	rdg	logr	mlp	xgb	rf2	xgb2
F1-score	0,72	0,73	0,69	0,74	0,69	0,66	0,72	0,67	0,69	0,70	0,74	0,80	**0,81**
Precisão	0,78	0,80	0,77	0,79	0,74	0,67	0,70	0,86	0,77	0,75	0,77	0,83	**0,90**
Revocação	0,69	0,69	0,65	0,70	0,66	0,66	0,74	0,62	0,65	0,67	0,72	0,77	**0,76**
Acurácia	0,93	0,94	0,93	0,94	0,92	0,91	0,91	0,94	0,93	0,93	0,93	0,95	**0,95**

Analisando a matriz de confusão do meta-classificador XGBoost (Figura 1), observamos que a precisão é alta, isto é, o nível de confiança do meta-classificador é de $83,33\%$ quando ele aponta que uma questão é "Difícil". Para questões "Não Difíceis", a precisão obtida também foi alta $(96,13\%)$. Observamos também que, mesmo com *trade off* existente entre *precisão* e *revocação*, ainda assim a *revocação* obtida para questões "Difíceis" foi de $53,57\%$, o que significa que o modelo conseguiu reconhecer mais da metade das questões da classe minoritária.

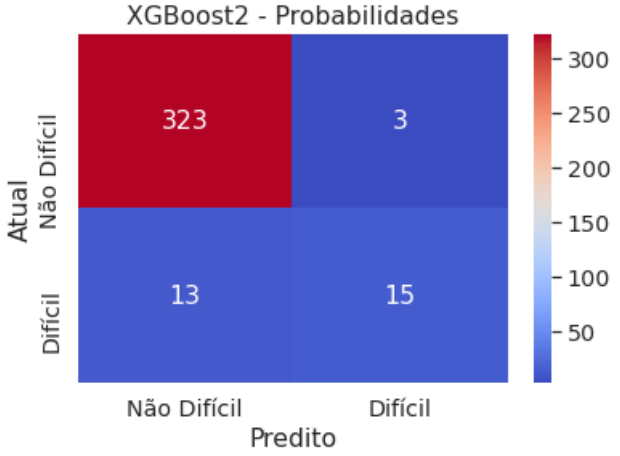

Figura 1. Matriz de confusão do meta-classificador XGBoost2

Com base no resultados apresentados, pode-se concluir que, apesar do problema de estimar a dificuldade de questões ser complexo, métricas extraídas de códigos de solução podem potencialmente ser utilizadas para classificação da dificuldade de questões. Seu resultado supera até mesmo os níveis de concordância de avaliadores humanos, como demonstrado por [Sheard et al. 2011], mesmo que utilizando um sistema de classificação um pouco mais simples. Outro ponto importante é que, enquanto a abordagem utilizando avaliadores não permite fácil escalabilidade, principalmente em grandes bases de questões, o método proposto neste trabalho pode rapidamente classificar novas questões conforme o crescimento da base.

A classificação de questões de forma automática pode ainda ser aplicada em diversos cenários de ensino, desde formulação de provas mais equilibradas, ordenação de questões por nível de dificuldade e até mesmo o uso em sistemas de recomendação automática de questões conforme o nível de habilidade do estudante.

5.3. Premissas e Limitações

A *taxa de acerto*, embora tenha sido a única *variável dependente* viável, dificilmente conseguirá captar integralmente a dificuldade encontrada pelos estudantes ao resolver questões de codificação. Em trabalhos futuros, pretende-se verificar a viabilidade da combinação de outras variáveis com a *taxa de acerto*.

Além disso, o conjunto de questões está dividido em sete tópicos, abordados pelo instrutor durante a disciplina. Isso dificulta a criação de um classificador capaz de generalizar para novas questões, uma vez que questões de diferentes tópicos podem abordar os mesmos conceitos de programação, apresentando portanto estruturas semelhantes no código, dificultando a tarefa de classificação.

Por fim, [Effenberger et al. 2019] demonstram que para mensurar de forma estável a dificuldade encontrada em questões introdutórias de programação são necessárias mais de cem soluções corretas para cada questão, e devido ao pequeno conjunto de questões disponíveis foi necessário utilizar um limiar menor, com o intuito de não diminuir tanto a quantidade de questões disponíveis para análise.

6. Conclusão e trabalhos futuros

Turmas introdutórias de programação apresentam alta taxa de reprovação, por isso é tão importante investir em ferramentas que melhorem o ensino. Nesse sentido, este artigo apresentou um método de classificação de questões por dificuldade, definida aqui como *taxa de acerto*, utilizando um conjunto de métricas obtidas a partir dos códigos de solução cadastrados pelo(a) instrutor(a) durante a criação das questões no ACAC adotado.

Um total de 91 métricas foram geradas a partir dos códigos de solução disponíveis. Em uma etapa posterior, essas métricas foram usadas como *variáveis preditoras* por um amplo conjunto de modelos de classificação combinados por meio de *stacking*. Como resultado, o meta-classificador *XGBoost* proposto apresentou o melhor desempenho dentre as abordagens testadas, alcançando um *f1-score* de 81% e *acurácia* de 95% durante o processo de classificação das questões em duas categorias, "Difícil" e "Não Difícil".

Como trabalho futuro, pretende-se conjugar as métricas extraídas do código de solução com as métricas de inteligibilidade textual extraídas do enunciado da questão. A hipótese é que o impacto das métricas sobre a estimativa da dificuldade de questão pode variar ao longo da disciplina. Por exemplo, a métrica "quantidade de operadores maior que" pode representar um importante indício de dificuldade durante as avaliações envolvendo estruturas condicionais, mas ela pode não ser tão importante em conteúdos mais avançados, quando boa parte dos alunos já aprendeu a trabalhar com tais operadores. Da mesma forma, as métricas envolvendo o laço *while* só fazem sentido a partir do momento em que esse tipo de estrutura é ensinado na disciplina.

7. Agradecimentos

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

Referências

Denny, P., Cukierman, D., and Bhaskar, J. (2015). Measuring the effect of inventing practice exercises on learning in an introductory programming course. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pages 13–22.

Effenberger, T., Čechák, J., and Pelánek, R. (2019). Measuring difficulty of introductory programming tasks. In *Proceedings of the Sixth (2019) ACM Conference on Learning @ Scale*, L@S '19, New York, NY, USA. Association for Computing Machinery.

Elnaffar, S. (2016). Using software metrics to predict the difficulty of code writing questions. In *2016 IEEE Global Engineering Education Conference (EDUCON)*, pages 513–518. IEEE.

Francisco, R. E. and Ambrosio, A. P. (2015). Mining an online judge system to support introductory computer programming teaching. In *EDM (Workshops)*.

Halstead, M. H. (1977). *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA.

INEP (2017). Relatório síntese de Área – Ciência da Computação. Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira.

Llana, L., E., M.-M., and Pareja-Flores (2012). Flop, a free laboratory of programming. *In Proceedings of the 12th Koli Calling International Conference on Computing Education Research*.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Meisalo, V., Sutinen, E., and Torvinen, S. (2004). Classification of exercises in a virtual programming course. In *Frontiers in Education Conference, FIE*, volume 3.

Pelánek, R. (2016). Applications of the Elo rating system in adaptive educational systems. *Computers & Education*, 98:169–179.

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

Santos, P., Carvalho, L. S. G., Oliveira, E., and Fernandes, D. (2019). Classificação de dificuldade de questões de programação com base na inteligibilidade do enunciado. In *Simpósio Brasileiro de Informática na Educação (SBIE)*, volume 30, pages 1886–1895.

Sheard, J., Simon, Carbone, A., Chinn, D., Laakso, M.-J., Clear, T., Raadt, M., D'Souza, D., Harland, J., Lister, R., Philpott, A., and Warburton, G. (2011). Exploring programming assessment instruments: A classification scheme for examination questions. In *7th Intern. Workshop on Computing Education Research (ICER)*, Providence, USA.

Simon, Sheard, J., Carbone, A., Chinn, D., Clear, T., Corney, M., D'Souza, D., Fenwick, J., Harland, J., Laakso, M.-J., and Teague, D. (2013). How difficult are exams? a framework for assessing the complexity of introductory programming exams. In *15th Australasian Computing Education Conference (ACE2013)*.

Wolpert, D. (1992). Stacked generalization. *Neural Networks*, 5:241–259.

Zaffalon, F., Vargas, A. P., Souza, R. L., Penna, R., Bez, J., Tonin, N., and Costa Botelho, S. S. (2019). Um estudo comparativo entre dois modelos que estimam a habilidade dos estudantes: Elo e teoria de resposta ao item. In *Simpósio Brasileiro de Informática na Educação (SBIE)*, volume 30, pages 459–468.