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

Submission: 17/Mar/2021; 1st round notif.: 19/Apr/2021; New version: 15/Jun/2021; 2nd round notif.: 01/Jul/2021;
Camera ready: 18/Aug/2021; Edition review: 30/Aug/2021; Available online: 01/Sep/2021; Published: 01/Sep/2021;

Uso de atributos de código para classificar a dificuldade de questões de programação em juízes online

Title: Using code atributes to classify the difficulty of programming tasks in online judges

Marcos A. P. de Lima
Instituto de Computação
Universidade Federal do Amazonas
marcos.lima@icomp.ufam.edu.br

Leandro S. G. Carvalho
Instituto de Computação
Universidade Federal do Amazonas
galvao@icomp.ufam.edu.br

Elaine H. T. Oliveira
Instituto de Computação
Universidade Federal do Amazonas
elaine@icomp.ufam.edu.br

David B. F. Oliveira
Instituto de Computação
Universidade Federal do Amazonas
david@icomp.ufam.edu.br

Filipe D. Pereira
Departamento de Ciência da Computação
Universidade Federal de Roraima
filipe.dwan@ufrr.br

Resumo

Em turmas introdutórias de programação é comum o uso de juízes online como ferramenta para elaboração de avaliações por meio de um sorteio aleatório de questões de programação. Para que o sorteio aleatório de questões seja equilibrado, é necessário que as questões tenham sido classificadas segundo sua dificuldade ou facilidade. Desse modo, este trabalho apresenta duas abordagens para classificar questões de programação pelo uso de atributos extraídos automaticamente de códigos de solução para as questões, uma segundo a facilidade e outra segundo a dificuldade das questões. Foram classificadas 404 questões com implementação em Python, que foram utilizadas em avaliações de turmas de introdução à programação ministradas entre 2017 e 2019. Ambas as abordagens utilizadas apresentaram bons resultados para classificação dicotômica das questões de programação.
***:** Dificuldade, Classificação, Questões de Programação, Atributos de Código, Programação Introdutória.*

Abstract

In introductory programming classes, online judges are used as a tool for preparing assessments through a random draw of programming questions. For the random draw of questions to be balanced, questions should be classified according to their difficulty or ease. Thus, this work presents two approaches to classify programming questions by using attributes automatically extracted from solution codes for the questions. In total, 404 questions with implementation in Python were classified, which were used in evaluations of classes of introduction to programming taught between 2017 and 2019. Both approaches presented good results for dichotomous classification of programming questions.
***Keywords:** Difficulty, Classification, Programming tasks, Code atributes, Introductory programming.*

Cite as: 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 classificar a dificuldade de questões de programação em juízes online. Revista Brasileira de Informática na Educação, 29, 1137-1157. DOI: 10.5753/RBIE.2021.29.0.1137.

1 Introdução

Muitas ferramentas têm sido criadas para apoiar o processo de ensino-aprendizagem de programação. Dentre elas, os ambientes de correção automática de código (ACAC), também conhecidos como juízes online, são promissores e têm sido amplamente empregados em disciplinas de programação introdutória (CS1) (R. E. Francisco & Ambrosio, 2015; F. D. Pereira et al., 2020; Llana, Martin-Martin, & Pareja-Flores, 2012; Neves et al., 2017; Zordan Filho et al., 2020).

Os juízes online foram inicialmente desenvolvidos para uso em competições de programação. Apesar disso, eles se provaram excelentes ferramentas de apoio ao ensino e a aprendizagem de programação por permitirem a correção automática das questões, diminuindo o trabalho de instrutores e fornecendo um feedback imediato aos estudantes (R. Francisco, Júnior, & Ambrósio, 2016; Fonseca et al., 2020). Esses ambientes disponibilizam vários conjuntos de *questões de codificação*, abrangendo diferentes conceitos de programação (F. D. Pereira et al., 2021).

Devido à necessidade dos estudantes realizarem muitos exercícios práticos, é comum que, no ensino-aprendizagem de programação, sejam indicadas diversas atividades (listas de exercícios, trabalhos práticos e avaliações) envolvendo problemas de codificação (de Oliveira et al., 2020; F. D. Pereira et al., 2019; da Rocha Braz et al., 2021; Araujo et al., 2021). Contudo, isso gera uma sobrecarga de trabalho para os instrutores, tanto na elaboração quanto para a correção das atividades. Como solução, os juízes online são utilizados para facilitar a elaboração de atividades e automatizar o processo avaliativo das soluções dos estudantes (Bez, Ferreira, & Tonin, 2013; Galvão, Fernandes, & Gadelha, 2016; R. E. Francisco, Ambrósio, Junior, & Fernandes, 2018).

As atividades presenciais, em especial as *avaliações*, são realizadas geralmente em laboratórios com espaço limitado e sob supervisão de um instrutor ou monitor. Durante essas atividades os estudantes fazem uso de computadores distribuídos lado a lado. Poucos centímetros separam um colega do outro, permitindo facilmente conversas paralelas enquanto o instrutor não estiver observando. Também é possível que colegas sentados em extremidades opostas do laboratório se comuniquem por meio de aplicativos de conversa abertos em outra aba do navegador. Da mesma forma, no cenário do ensino remoto imposto pela necessidade de distanciamento social, tal possibilidade de comunicação entre os estudantes durante atividades avaliativas foge do controle do instrutor.

Para minimizar comportamentos desonestos (plágios, conversas paralelas, etc.), o instrutor cria um banco de questões que — segundo sua percepção pessoal — têm níveis semelhantes de dificuldade. Fazendo uso desse conjunto de questões, o juiz online pode sortear aleatoriamente um subconjunto de questões para cada estudante, diminuindo a possibilidade da troca de código entre eles durante a avaliação (Joy & Luck, 1999; Ullah et al., 2020). Porém, quando se trata de programação introdutória, uma simples diferença entre uma questão e outra, como por exemplo a troca da expressão “maior que” por “maior ou igual que”, pode influenciar na dificuldade da questão e alterar equidade da avaliação.

De modo análogo, quando pensamos num contexto de ensino assistido, como por exemplo em sistemas de recomendação automática de questões de codificação, é importante que o estudante receba questões adequadas ao seu nível de conhecimento (de Freitas Júnior, Pereira, de Oliveira, de Oliveira, & de Carvalho, 2020; F. Pereira et al., 2021). Recomendar questões muito difíceis pode afetar negativamente o interesse do estudante, enquanto que sugerir questões demasiada-

mente fáceis não traria avanço no conhecimento. Para conseguir recomendar questões relevantes, o sistema precisa que o conjunto de questões esteja classificado segundo sua dificuldade.

Assim, o problema de pesquisa aqui abordado é o seguinte. Quando o instrutor cadastra uma nova questão no juiz online, as únicas informações disponíveis são o enunciado, os casos de teste e o código de solução. Nenhum dado de interação dos estudantes ainda está disponível, pois essas novas questões não foram apresentadas aos estudantes. Logo, não é possível conhecer a dificuldade (ou facilidade) da questão antes que seja aplicada a uma classe. Por outro lado, a descrição de uma questão (enunciado, casos de teste e código de solução) pode fornecer indícios para estimar a dificuldade de novas questões. Neste trabalho, focamos apenas em informações que podem ser extraídas a partir do código de solução.

Portanto, o objetivo geral deste trabalho é estimar a dificuldade/facilidade de novas questões de codificação por meio de atributos de código extraídos de códigos de solução cadastrada pelo(a) instrutor(a). Visando atingir esse objetivo, buscamos responder às seguintes questões de pesquisa:

QP1: Como a dificuldade de novas questões de programação pode ser classificada por atributos extraídos do código de solução?

QP2: Existe correlação entre a dificuldade de questões e atributos extraídos de códigos de solução?

QP3: Existe diferença de desempenho em classificar as questões por dificuldade ou por facilidade?

Como resultado, foi possível classificar as questões tanto por facilidade quanto por dificuldade, usando atributos extraídos automaticamente de códigos modelo de solução. Para estimar a facilidade e dificuldade foi utilizada a taxa de acerto das questões, discretizada dicotomicamente. A classificação por facilidade, por apresentar menor desbalanceamento entre as classes, obteve um melhor resultado final com um *f1-score* de 0,84 com 0,86 de *acurácia*. Por sua vez, a classificação por dificuldade obteve um *f1-score* de 0,82 com 0,96 de *acurácia*.

Esses resultados representam um passo inicial para tentar estimar a facilidade/dificuldade de questões introdutórias de programação. Fornecer uma estimativa da facilidade/dificuldade permite, por exemplo, que o instrutor, durante a elaboração de uma atividade de codificação, selecione o grupo de questões mais adequado para o momento da disciplina. Essa estimativa ainda pode ser utilizada em sistemas de recomendação automática de questões, onde o próprio juiz online sugere questões para um estudante, com base no seu histórico de tentativas.

Por fim, o presente artigo está estruturado da seguinte forma: a Seção 2 apresenta os trabalhos relacionados e suas abordagens; a Seção 3 esclarece o contexto das questões utilizadas; a Seção 4 descreve o método de pesquisa utilizado; a Seção 5 apresenta os resultados e discute sobre como eles respondem às questões de pesquisa; e, por fim, a Seção 6 apresenta as considerações finais e trabalhos futuros.

2 Trabalhos relacionados

Uma forma intuitiva de classificar a dificuldade de uma questão de codificação é utilizar a avaliação humana. Nessa abordagem, um grupo de avaliadores estima, com base em sua experiência pessoal, o nível de dificuldade de um conjunto de questões. Por exemplo, (R. E. Francisco & Ambrosio, 2015) propõem um ambiente no qual os exercícios são classificados em 3 níveis de dificuldade; já no ambiente proposto por (Llana et al., 2012), há 5 níveis de classificação; por sua vez, (Denny, Cukierman, & Bhaskar, 2015) classificam a dificuldade usando uma rubrica de 6 níveis. Porém, a classificação manual apresenta alguns inconvenientes: a subjetividade na interpretação dos rótulos dos níveis de dificuldade, o tempo consumido na tarefa pode ser demasiadamente alto e a indisponibilidade de avaliadores para trazer uma segunda ou terceira opinião. Em resumo, essa abordagem até pode ser aceitável para um pequeno conjunto de questões, todavia ela não é escalável, principalmente em ambientes em que novas questões de codificação são criadas e disponibilizadas constantemente.

Outro ponto negativo é a subjetividade na interpretação, pois os 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 3 pontos. Ainda que haja um consenso entre os avaliadores, talvez o resultado não corresponda ao valor que se deseja medir. Como evidenciado em (Meisalo, Sutinen, & Torvinen, 2004), a dificuldade estimada pelo instrutor, por estar relacionada com sua experiência pessoal, geralmente não condiz com a evidenciada pelos estudantes.

Recentemente, alguns trabalhos têm buscado analisar dados gerados pelos próprios usuários dos juízes online 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 (Teoria de Resposta ao Item). O modelo Elo foi desenvolvido por Arpad E. Elo (Elo, 1978) 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 (Pelánek, 2016; Vargas, dos Santos, Botelho, Tonin, & Bez, 2017; Vargas et al., 2018). 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 infelizmente 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 (Sheard et al., 2013). Nesse sentido, em um trabalho anterior (Santos, Carvalho, Oliveira, & Oliveira, 2019), buscou-se correlacionar atributos de legibilidade do enunciado com a dificuldade das questões. Porém, não foi possível encontrar correlação significativa. Foi possível apenas confirmar, por meio dos dados, a noção intuitiva de que embora questões com enunciado de difícil leitura normalmente produzam baixa taxa de acerto, questões com enunciado simples e de fácil leitura nem sempre produzem altas taxas de acerto. O inconveniente dessa abordagem é que enunciados curtos interferem na extração de atributos, pois

eles se baseiam na contagem de sílabas, palavras e sentenças.

Por sua vez, alguns trabalhos fazem uso de atributos extraídos do código de solução da questão para estimar o nível de dificuldade. Por exemplo, (Elnaffar, 2016) encontrou forte correlação (>0,9) entre a dificuldade de questões introdutórias de programação em Java e alguns atributos extraídos manualmente de códigos de solução de estudantes (complexidade ciclomática, profundidade média dos blocos, número de operadores e número de chamadas de métodos). Contudo, os resultados são limitados a uma amostra de apenas 10 questões, de uma classe de tamanho não explicitado, ensinada por um único instrutor. Além disso, a extração manual (humana) de atributos demanda muito tempo e é suscetível a erros.

De modo análogo, (Effenberger, Čechák, & Pelánek, 2019) analisaram 4 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. Diferente deste trabalho, os autores definem que a dificuldade deve ser estimada com base no comportamento dos estudantes ao resolverem a questão, enquanto que a complexidade depende apenas do código da questão, podendo ser medida sem consultar dados de desempenho. A complexidade foi estimada 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 4 métricas: taxa de erro e medianas do tempo de acerto, número de edições e número de execuções. Por fim, os autores encontraram uma correlação moderada (Spearman $\rho = 0{,}73$) entre *complexidade* e *dificuldade*, ou seja, existe uma dependência estatística entre as variáveis.

Diferente dos trabalhos citados nesta seção, o problema de estimar a dificuldade de uma nova questão de codificação inserida no juiz online será abordado neste trabalho a partir de atributos extraídos automaticamente de seu código de solução, que é um exemplo de código correto informado pelo instrutor no momento da criação do exercício no juiz online. Como exemplo de atributos de código, temos: número de atribuições, número de variáveis, número de operadores, etc. Ao todo, foram obtidos 111 atributos de código. Além disso, a dificuldade de uma questão de codificação foi expressa pela *taxa de acerto*[1], 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.

3 Contexto das questões de programação utilizadas

Esta pesquisa se baseou em dados extraídos do juiz online Codebench[2] da Universidade Federal do Amazonas. Essa ferramenta avalia automaticamente as soluções elaboradas por estudantes por meio do uso de casos de testes. Além disso, o ambiente registra em *arquivos de log* as ações dos estudantes durante as tentativas bem como os resultados das submissões/execuções dos códigos de solução por eles elaborados.

Foram consideradas apenas questões resolvidas por estudantes da disciplina de Introdução à Programação de Computadores (IPC), entre 2017 e 2019. Essa disciplina é ministrada para 17

[1] A justificativa para a escolha da taxa de acerto para mensurar a dificuldade é dada na Seção 4.1

[2] http://codebench.icomp.ufam.edu.br/

cursos de graduação nas áreas de engenharia e ciências exatas da universidade citada. Dentre as questões disponíveis, foram analisadas apenas aquelas usadas em *exames presenciais*, pois, durante sua aplicação, havia um instrutor e um tutor para tirar dúvidas e para evitar que os estudantes compartilhassem respostas entre si.

Das questões disponíveis, 653 haviam sido utilizadas em *exames presenciais* e exigiam solução codificada em Python. Em seu trabalho, (Effenberger et al., 2019) afirmam que a tarefa de estimar a dificuldade requer que uma quantidade suficiente de dados tenha sido coletada, e que essa quantidade pode variar dependendo da métrica adotada. Sendo assim, foi decidido aplicar um filtro nas questões da base, selecionando uma amostra de questões que tivessem sido solucionadas por uma quantidade razoável de estudantes, com o objetivo de reduzir o viés da especifidade de resolução. Ou seja, era necessário descartar questões com poucas soluções registradas. Caso contrário, o grau de dificuldade espelharia o desempenho de poucos indivíduos, e não de uma parcela representativa dos estudantes.

A Figura 1 apresenta as curvas das distribuições da taxa de acerto das questões para alguns limiares de quantidade de resoluções (t). Nessa figura, o formato das curvas começam a estabilizar, isto é, aproximar-se de uma distribuição normal (ainda que assimétrica), quando são selecionadas questões com 16 ou mais tentativas de solução. Valores menores que 16 apresentam muito ruído (topos e fundos no gráfico) enquanto de limiares maiores começam a reduzir demasiadamente a amostra, restando poucas questões para o estudo. Portanto, foram selecionadas somente as questões com 16 ou mais tentativas. Isso resultou numa amostra com 404 questões (aproximadamente 62% do total de questões disponíveis), que apresentavam quantidade de soluções de estudantes suficiente para extrair medidas representativas da dificuldade de resolução.

Contudo, o campo "código de solução do instrutor" foi adicionado ao ambiente somente em 2019. Por isso, para o conjunto de questões selecionadas, somente 323 apresentavam um código de solução elaborada pelo instrutor. Como as questões utilizadas são de uma disciplina introdutória, seus códigos de solução são simples e com poucas variações em relação aos exercícios de fixação, logo assumiu-se não existir tanta diferença entre as soluções dos instrutores e estudantes. Dessa forma, para as questões sem código de solução do instrutor foram utilizados códigos de solução dos estudantes que acertaram as questões.

Como cada questão dispunha de diversas soluções elaboradas pelos estudantes, era necessário escolher a melhor solução disponível. Para essa escolha os seguintes critérios foram adotados, em ordem de prioridade:

1. O menor número de submissões para correção automática;
2. O menor número de execuções de código ocorridas durante a resolução da questão;
3. O menor número de erros acusados pelo interpretador Python durante a tentativa de solução.

O uso do número de submissões tem como objetivo desconsiderar primeiramente aquelas soluções que embora estejam corretas, foram obtidas por meio de "força bruta", ou seja, inúmeras submissões consecutivas com pequenas alterações no código, muitas delas sem critério algum. As execuções de código ocorrem quando o estudante está testando seu código fornecendo entradas e avaliando as saídas geradas. O uso desse critério de seleção tem por objetivo selecionar soluções em que o estudante foi capaz de ajustar ou corrigir seu código sem a necessidade de testes

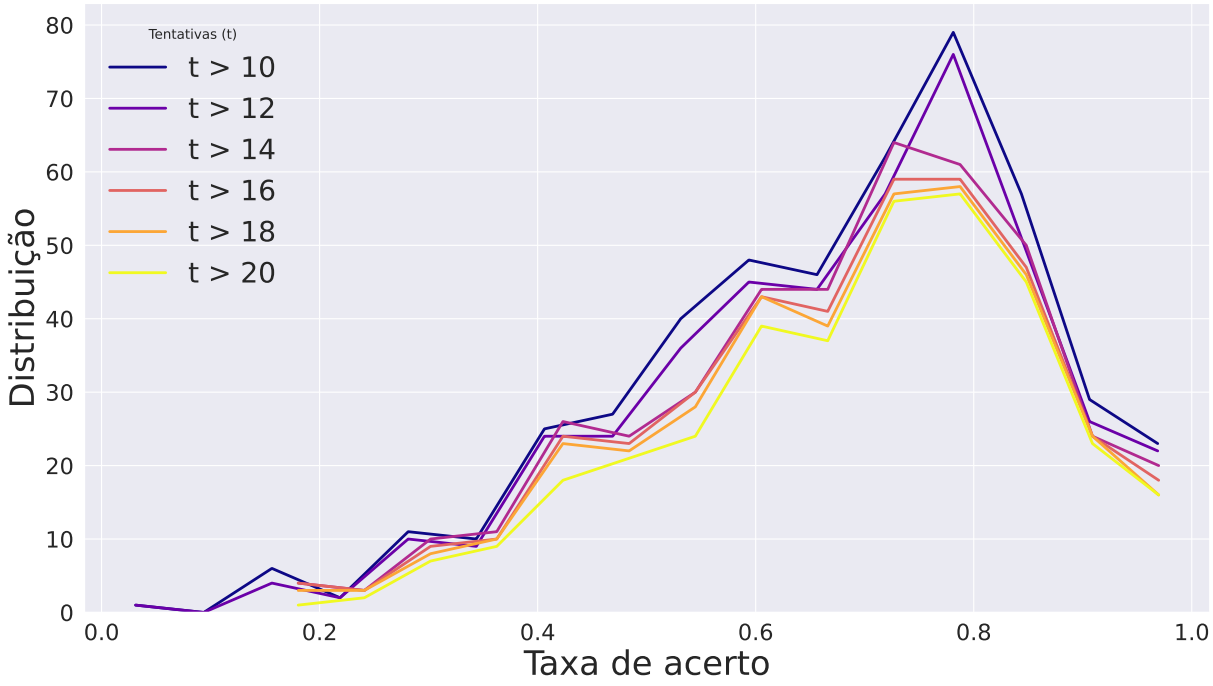

Figura 1: Distribuições da taxa de acerto por número de tentativas (t).

exaustivos de entradas e saídas. Durante as execuções de código ou submissões para a correção automática os erros obtidos pelo interpretador Python são apresentados aos estudantes. Por isso, filtrar as soluções pelo menor número de erros significa selecionar as soluções de estudantes que foram capazes de interpretar e corrigir tais erros.

Embora o juiz online permita que o estudante, mesmo após solucionar uma questão, continue a submeter códigos para a correção automática, somente o último código submetido é armazenado na base de dados. Além disso, os três critérios de desempate foram contabilizados até o exato momento em que o estudante submete a primeira versão correta. Essa abordagem visa penalizar as soluções de alunos que acertaram as questões por "tentativa e erro". Os erros são registrados pelo juiz online utilizando a resposta do interpretador Python para cada submissão ou execução de código.

4 Método de pesquisa

Esta seção descreve os passos adotados para o desenvolvimento deste trabalho. Primeiramente foram analisadas três possíveis variáveis dependentes para estimar a dificuldade e facilidade de questões, sendo por fim escolhida a taxa de acerto. Em seguida, foi inicialmente extraído um conjunto de atributos dos códigos de solução e posteriormente outros atributos foram derivados. Na etapa seguinte, a taxa de acerto por ser uma variável contínua foi discretizada em duas clas-

ses, tanto para a dificuldade quanto para a facilidade. Como passo seguinte, foram definidas as métricas de avaliação dos modelos de classificação a serem implementados. Por fim, na etapa de classificação utilizou-se da técnica *stacking* (Wolpert, 1992) para treino de classificadores para ambas as abordagens.

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: *taxa de acerto* e *tempo médio de resolução do problema*. Outra possível variável associada à dificuldade é o *número médio de submissões* necessários para acertar dada questão.

Durante a análise dos *arquivos de log* 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. Em contrapartida, outros estudantes testam exaustivamente o código antes de submeter, gerando uma ou poucas submissões até acertar. Como consequência, o uso do *número médio de submissões* como *variável alvo* poderia levar a uma classificação errônea das questões.

O *tempo médio de resolução* de uma questão, por sua vez, foi calculado por meio de um *arquivo de log* que registra a data e hora de todas as interações do estudante (eventos) com o juiz online. No entanto, 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 de solucionar a questão. Consequentemente, o tempo calculado por essa abordagem é subjetivo e não confiável.

Desse modo, para mensurar a dificuldade de questões de programação em um juiz online, adotou-se como *variável dependente* somente 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)

Foi utilizado como *variáveis independentes* um conjunto de atributos extraídos dos códigos de solução das questões e também alguns atributos gerados por meio de um processo de *engenharia de atributos*. Conforme dito anteriormente, um código de solução nada mais é do que um código fornecido pelo professor durante o cadastro de uma dada questão, representando uma possível solução para tal questão.

No processo de extração automática de atributos de código, foram utilizados dois módulos da linguagem Python: *Radon*[3] e *Tokenize*[4]. O módulo *Radon* foi usado para extrair atributos referentes a métricas de software (Halstead, 1977), atributos baseados no tamanho do código (linhas de código, linhas lógicas, linhas em branco e comentários) e também a *complexidade ciclomática*

[3] https://pypi.org/project/radon/
[4] https://docs.python.org/3/library/tokenize.html

total do código (McCabe, 1976). O módulo *Tokenize*, por sua vez, foi utilizado para a *análise léxica* do código de solução, gerando então *tokens*, que posteriormente foram transformados em "atributos derivados". Os "atributos derivados" estão relacionados com a quantidade de ocorrência dos seguintes elementos: estruturas condicionais, estruturas de repetição, operadores, comandos de importação, funções embutidas (*built-in functions*), 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 e relacionais). 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.

Após a extração dos atributos, foi realizado um processo de *engenharia de atributos*, com o objetivo de gerar o máximo de informações distintas e que pudessem ser utilizadas na classificação da dificuldade. Foram então criados atributos *booleanos* indicando se no código de solução analisado havia estruturas condicionais, estruturas de repetição, operadores, constantes, comandos de importação, palavras-chave, etc. Além disso, alguns atributos foram derivados diretamente dos "atributos baseados em tamanho": *média de identificadores por linha de código*, *média de caracteres por identificador*, por exemplo.

Como as questões abordadas foram utilizadas somente em uma disciplina introdutória de programação, é esperado a ausência de algumas construções mais avançadas. Com isso, alguns atributos apresentaram pouca ou até mesmo nenhuma variância nos seus valores, como por exemplo *operadores bit-a-bit*, *declarações de classes* e *expressões lambda*. Por isso, foram removidos todos os atributos que apresentavam variância zero, restando então 92 atributos dos 111 extraídos.

A listagem completa dos atributos extraídos e também os scripts utilizados para extração estão disponíveis no repositório da ferramenta de mineração desenvolvida para processar os dados do dataset do juiz online Codebench[5].

4.3 Discretização da taxa de acerto

A variável dependente *taxa de acerto*, por ser contínua, conduziria este trabalho para um problema de regressão, resultando na proposição de uma escala de dificuldade. Embora viável, para o instrutor ou estudante é mais informativo saber se dada questão é ou não difícil, além de que um valor contínuo é bem menos interpretativo. Portanto, optou-se por discretizar a taxa de acerto em classes, tornando então este um problema 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 (INEP, 2017). Este índice classifica as questões em cinco níveis de *facilidade* por meio da *taxa de acerto* (Tabela 1).

Contudo, a discretização da base segundo o índice original era muito desbalanceada, com mais da metade das questões pertencentes a classe "Fácil", enquanto que a classe "Muito Difícil" apresentava apenas duas questões. Assim sendo, o índice original foi adaptado para dois níveis, de forma a obter uma tarefa de classificação binária e menos desbalanceada.

Para a adaptação do índice INEP foram utilizadas duas abordagens: a primeira dividiu as

[5] https://github.com/marcosmapl/codebench-extractor

questões segundo sua facilidade e a outra segundo sua dificuldade. A classificação segundo a facilidade divide dicotomicamente o conjunto de questões em "Fácil" e "Não Fácil". A classificação por dificuldade, por sua vez, também divide o conjunto de questões em duas classes: "Difícil" e "Não Difícil". A Tabela 1 apresenta o mapeamento adotado em ambas as abordagens.

Tabela 1: Discretização da taxa de acerto.

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

Embora a simplificação do índice original ainda tenha resultado numa distribuição desbalanceada de questões em ambas as propostas de classificação, mesmo as classes minoritárias apresentavam uma boa quantidade de amostras de questões, o que permite uma boa generalização pelos modelos de classificação. A classificação segundo a dificuldade apresentou o maior desbalanceamento, tendo a classe "Difícil" como minoritária apresentando apenas 28 questões (6,93%), enquanto que a classe "Não Difícil" continha 376 questões (93,07%). A classificação segundo a facilidade, por sua vez, teve a classe "Não Fácil" como minoritária com 130 questões (32,18%), enquanto que a classe "Fácil" apresentou 274 questões (67,82%).

4.4 Métricas para avaliação de modelos de classificação

Para avaliar o desempenho dos modelos de classificação, 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 por exemplo no problema de classificação segundo a dificuldade de questões, prever uma questão "Difícil" como sendo "Não Difícil" pode prejudicar os estudantes lhe dando um conjunto de questões mais complexas do que o desejado. Por outro lado, prever uma questão "Não Difícil" como sendo "Difícil" pode acabar gerando exames sem questões desafiadoras. Com o objetivo de ter um equilíbrio na predição para ambas as classes, adotou-se como métrica principal para mensurar o desempenho dos modelos de classificação o *f1-score*, 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 calculadas, a título de completude.

Como critério de desempate, caso dois ou mais modelos obtivessem o mesmo *f1-score*, foi utilizada a métrica *log-loss* (ou *cross-entropy*). Essa métrica mensura a perda logarítmica entre as probabilidades para cada classe de saída. Quanto menor o *log-loss*, mais assertivas são as

probabilidades estimadas pelo modelo para cada classe.

4.5 Processo de Classificação

Para o processo de classificação foi utilizado o ambiente em nuvem *Google Colaboratory* e a biblioteca de aprendizagem de máquina *scikit-learn*. A classificação por dificuldade e também segundo a facilidade ocorreu em duas etapas, utilizando uma adaptação da técnica de *ensemble* chamada *stacking* (Wolpert, 1992).

Numa primeira etapa, foram treinados 6 modelos classificadores-base distintos para cada uma das duas classificações abordadas. Os classificadores-base são assim chamados pois suas predições servirão como dados de treino para outros classificadores, numa etapa posterior. Tanto as predições quanto as probabilidades servirão como dados de entrada para o treino de novos classificadores, que nessa etapa são chamados de meta-classificadores. O objetivo dessa técnica é que os meta-classificadores consigam melhorar as predições e estimativas geradas pelos classificadores-base.

Os seguintes modelos foram utilizados como classificadores-base:

- *K-Nearest Neighbor* (**knn**)
- *Support Vector Machine* (**svm**)
- *GradientBoosting* (**gb**)
- *Extreme Gradient Boosting* (**xgb**)
- *Gausian Naive Bayes* (**gnb**)
- *Logistic Regression* (**logr**)

Cada classificador-base foi treinado de forma independente dos demais por meio de validação cruzada (*cross-validation*) para minimizar a possibilidade de *overfitting* nos dados. Devido à pouca quantidade de amostras para as classes minoritárias, os dados foram divididos em apenas 5 partições (subconjuntos), escolhidas de forma *estratificada* e utilizando o número 0 (zero) como *semente de aleatoriedade*. A cada rodada de treino por validação cruzada, 4 partições foram selecionadas para treino e 1 para validação. Antes do treino do modelo, as 4 partições de treino foram balanceadas utilizando a técnica SMOTE (*Synthetic Minority Over-Sampling Technique*) (Chawla, Bowyer, Hall, & Kegelmeyer, 2002). Tanto as predições quanto as probabilidades estimadas para a partição de validação foram 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. Para a seleção, foi utilizado o método chamado de `SelectKBest` que faz uso de uma função de pontuação para ranquear os atributos e então selecionar os k melhores. Como função de pontuação, foi utilizada a função *f_classif*, que calcula o valor F da análise de variância (ANOVA).

Na segunda etapa de classificação, para cada uma das propostas de classificação, 3 meta-classificadores foram treinados utilizando as predições dos classificadores-base e outros 3 meta-classificadores foram treinados utilizando as probabilidades estimadas pelos classificadores-base. O processo de treino dos meta-classificadores também ocorreu por meio de validação cruzada com

05 partições *estratificadas*, porém agora utilizando o número 1 (um) como *semente de aleatoriedade*, evitando assim que os meta-classificadores fossem treinados com as mesmas partições que os classificadores-base. Os modelos utilizados como meta-classificadores foram:

- *K-Nearest Neighbor* (**knn**)
- *Support Vector Machine* (**svm**)
- *Extreme Gradient Boosting* (**xgb**)

5 Resultados e análises

A análise da importância dos atributos e os resultados obtidos serão apresentados e discutidos nesta seção. Primeiramente, foram analisadas as correlações dos atributos extraídos dos códigos com a variável dependente, de modo que possamos identificar se existe relevância estatística para seus valores. Num segundo momento, utilizou-se de um modelo *ensemble* para pontuar a relevância de cada atributo para ambas as abordagens de classificação, pois isso permite vislumbrar o que os modelos de classificação estão considerando relevante para a classificação e também como estão aprendendo. Em seguida, são apresentados e discutidos os resultados obtidos tanto para a dificuldade quanto para a facilidade. Por fim, são apresentadas as premissas que serviram de base para este trabalho e também quais as limitações.

5.1 Correlação e atributos importantes

Para responder a **QP2** foram testadas as correlações entre as *variáveis independentes* (atributos) com a *variável dependente* (taxa de acerto) segundo o coeficiente de *Spearman*. Esse coeficiente foi escolhido porque a maioria das variáveis encontradas não apresentavam uma distribuição normal. Observou-se em geral uma correlação fraca, com exceção dos atributos *has_loops* e *loops*. O atributo *has_loops* indica se existe laço(s) de repetição no código e apresentou $\rho_S = -0,41$ e valor-$p = 4,39 \times 10^{-16}$. O atributo *loops* contabiliza a quantidade de estruturas de repetição no código e apresentou $\rho_S = -0,38$ e valor-$p = 2,27 \times 10^{-13}$. Embora os valores de correlação encontrados tenham sido moderados, ambas são estatisticamente significantes.

Verificou-se que a 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 9 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 Análise da importância dos atributos

Além da análise dos resultados é importante compreender também a lógica do modelo. Isso possibilita não apenas verificar o funcionamento correto do modelo, mas também permite aprimorá-lo, observando quais os atributos mais importantes para o modelo.

Essa análise pode ser feita utilizando os scores gerados pelos modelos *Extreme Gradient Boosting* de cada uma das abordagens propostas, classificação por facilidade e classificação por dificuldade respectivamente. Para a classificação por facilidade os atributos com maiores scores (Figura 2) foram os baseados na presença de *operadores lógicos*, de *palavras-chave*, de *laços de repetição* e nos atributos relacionados com o tamanho do código-fonte (linhas lógicas e linhas totais).

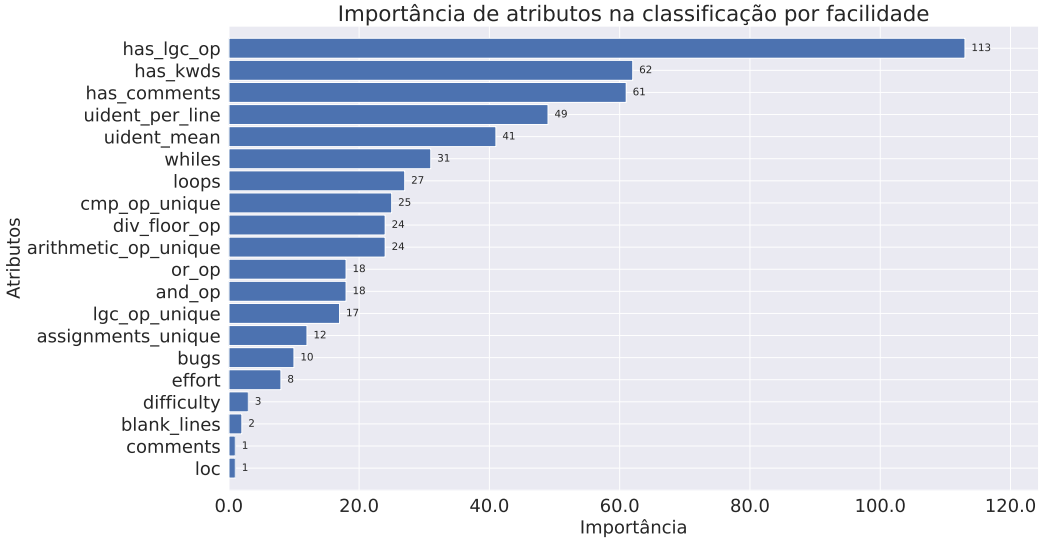

Figura 2: Importância dos atributos para a classificação por facilidade.

Dentre eles, o atributo mais importante foi o *has_lgc_op*, que indica se o código de solução possui *operadores lógicos*. Ocorre que os códigos de solução de 251 das questões da classe "Fácil" ($62,13\%$ das questões na amostra) não apresentam nenhum operador lógico, o que indica que os estudantes podem ter mais dificuldade em questões que abordem a combinação de expressões condicionais. Outro atributo importante para o modelo foi o *has_kwds*, que indica se o código de solução faz uso de *palavras-chave* da linguagem Python. Combinando esses dois atributos foi observado que os códigos de solução de 202 das questões da classe "Fácil" ($50,00\%$ das questões na amostra) fazia uso de *palavras-chave* e não continha *operadores lógicos*.

De modo análogo, a classificação por dificuldade também atribuiu maior importância aos atributos baseados na presença de *operadores lógicos*, de *palavras-chave*, de *laços de repetição* e nos atributos relacionados com o tamanho do código-fonte (Figura 3).

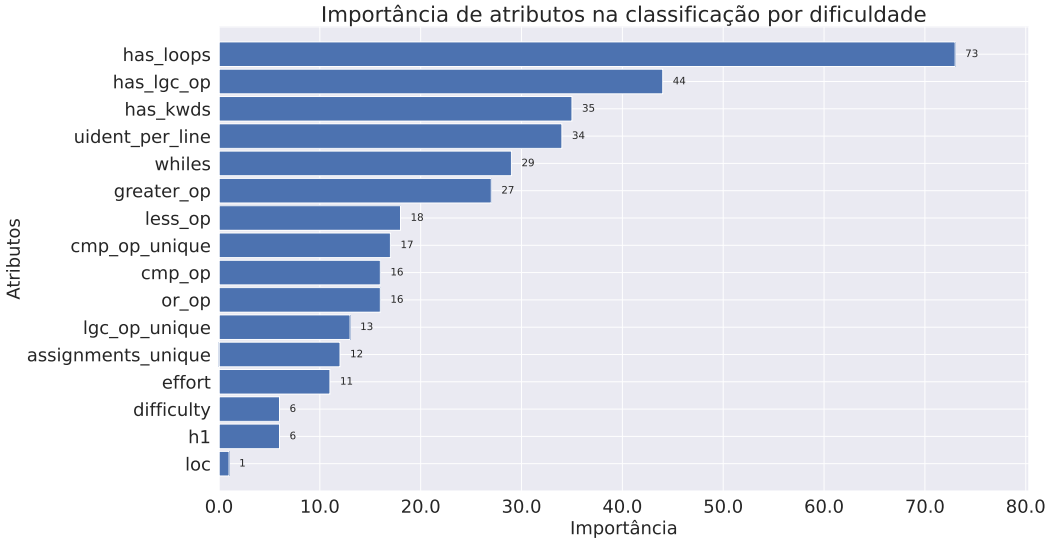

Figura 3: Importância dos atributos para a classificação por dificuldade.

Para a classificação por dificuldade, o atributo mais importante foi o *has_loops*, que indica se o código de solução apresenta *laços de repetição*. Além disso, o atributo *has_lgc_op* foi escolhido como segundo mais importante. Isso evidencia que as questões de maior dificuldade abordam *laços de repetição*, principalmente *laços de repetição por condição*, o que justifica o atributo *whiles* (quantidade de laços enquanto) como um dos mais importantes.

5.3 Resultados da classificação e discussão

Os resultados dos modelos de classificação-base para as classificações por dificuldade e facilidade são apresentados nas Tabelas 2 e 3, respectivamente. Na classificação por dificuldade, o melhor classificador-base foi um modelo *K-Nearest Neighbors* (Fix & Hodges Jr, 1952), utilizando a *Distância de Manhattan* dos 7 vizinhos mais próximos como métrica. Adicionalmente, foi aplicado o inverso da distância do vizinho como peso na equação de *Manhattan*. Desse modo, o modelo deu maior importância para vizinhos muito próximos, ao passo que vizinhos mais distantes contribuíram menos para a classificação. Esse modelo obteve um *f1-score* de 0,78 e *acurácia* de 0,94, utilizando 19 atributos dos 95 originais. Embora tenha obtido o maior valor de *log-loss* dentre os classificadores-base, esse valor não deve ser levado em consideração pois o modelo *K-Nearest Neighbors* estima a probabilidade para uma classe de acordo com a quantidade de vizinhos mais próximos escolhida, ou seja, o valor do *log-loss* é altamente correlacionado com a quantidade de vizinhos escolhida.

Para a classificação por facilidade, os classificadores-base tiveram melhor desempenho que os classificadores-base para a dificuldade. Muito disso é consequência do balanceamento entre as classes, que foi mais equilibrado para a classificação por facilidade. O melhor classificador-base foi um modelo *Logistic Regression*, utilizando regularização *L2* e $\lambda = 21,80$ ($C = 0.0459$). Esse modelo obteve um *f1-score* de 0,82, *acurácia* de 0,83 e *log-loss* de 0,43, utilizando para isso 39 atributos dos 95 originais.

Por sua vez, os resultados dos meta-classificadores para as classificações por dificuldade e

Tabela 2: Resultados dos classificadores-base para Dificuldade.

	knn	**svm**	**gb**	**xgb**	**gnb**	**logr**
f1-score	0,78	0,76	0,76	0,77	0,70	0,74
log-loss	0,98	0,23	0,40	0,22	0,88	0,32
acurácia	0,94	0,93	0,94	0,95	0,89	0,90
precisão	0,75	0,73	0,76	0,80	0,67	0,69
revocação	0,82	0,81	0,75	0,76	0,79	0,85

Tabela 3: Resultados dos classificadores-base para Facilidade.

	knn	**svm**	**gb**	**xgb**	**gnb**	**logr**
f1-score	0,80	0,77	0,79	0,81	0,79	0,82
log-loss	2,94	0,45	0,49	0,54	0,55	0,43
acurácia	0,82	0,80	0,82	0,83	0,80	0,83
precisão	0,80	0,77	0,80	0,81	0,78	0,81
revocação	0,80	0,78	0,79	0,81	0,81	0,84

facilidade são apresentados nas Tabelas 4 e 5, respectivamente. Na classificação por dificuldade o melhor meta-classificador foi um modelo *Extreme Gradient Boosting* (Chen & Guestrin, 2016) treinado com as *predições* de todos os classificadores-base, utilizando 7 estimadores (*weak leaners*) e o valor 1,62 como *learning rate*. Esse modelo obteve um *f1-score* de 0,82 e *acurácia* de 0,96. Além disso, o modelo também obteve um *log-loss* de 0,15, o menor valor dentre todos os modelos que foram utilizados.

Tabela 4: Resultados dos meta-classificadores para Dificuldade.

	Predições			Probabilidades		
	knn	**svm**	**xgb**	**knn**	**svm**	**xgb**
f1-score	0,81	0,82	0,82	0,75	0,79	0,79
log-loss	0,91	0,24	0,15	0,47	0,24	0,26
acurácia	0,96	0,95	0,96	0,95	0,94	0,95
precisão	0,88	0,79	0,87	0,83	0,78	0,81
revocação	0,76	0,86	0,78	0,71	0,80	0,76

Para a classificação por facilidade, dois meta-classificadores tiveram os melhores resultados. O primeiro foi um modelo *K-Nearest Neighbors* utilizando a *Distância de Manhattan* dos 21 vizinhos mais próximos como métrica. Diferentemente do melhor classificador-base para a "Dificuldade", não foram utilizados pesos na equação de *Manhattan*, pois o meta-classificador deveria atribuir a mesma importância para cada predição. O segundo classificador foi um modelo *Extreme Gradient Boosting* (Chen & Guestrin, 2016) treinado com as *probabilidades* estimadas pelos classificadores-base, utilizando 60 estimadores (weak leaners) e o valor 0,39 como *learning rate*.

Embora ambos os meta-classificadores tenham obtido o mesmo desempenho (*f1-score* de

Tabela 5: Resultados dos meta-classificadores para Facilidade.

	Predições			Probabilidades		
	knn	**svm**	**xgb**	**knn**	**svm**	**xgb**
f1-score	0,84	0,83	0,83	0,79	0,81	0,84
log-loss	0,76	0,37	0,38	0,83	0,39	0,41
acurácia	0,86	0,84	0,85	0,82	0,83	0,86
precisão	0,83	0,82	0,83	0,79	0,80	0,84
revocação	0,84	0,85	0,83	0,78	0,83	0,83

0,84 e *acurácia* de 0,86), o modelo *K-Nearest Neighbors* (knn) apresentou melhor revocação, ao passo que o modelo *Extreme Gradient Boosting* obteve melhor precisão. Além disso, comparando as matrizes de confusão de ambos os modelos (Figura 4), observa-se que o modelo *K-Nearest Neighbors* acertou mais questões da classe minoritária. Sendo por isso considerado o melhor meta-classificador para a facilidade.

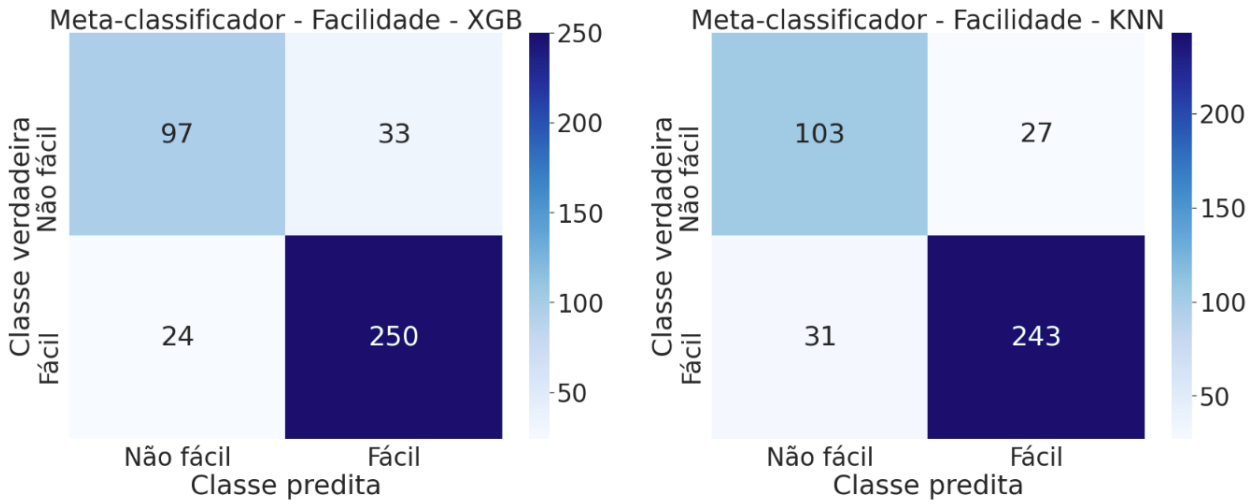

Figura 4: Matrizes de confusão dos melhores meta-classificadores para a facilidade.

A partir dos resultados apresentados, pode-se concluir que, apesar do problema de estimar a dificuldade de questões ser complexo, atributos extraídos de códigos de solução podem de fato ser utilizados por modelos de aprendizagem de máquina para classificar não somente a dificuldade mas também a facilidade de questões. Isso responde a **QP1**. Ambas as abordagens têm potencial para superar níveis de concordância de avaliadores humanos, como demonstrado por (Sheard et al., 2011), mesmo que utilizando um sistema de classificação um pouco mais simples.

Neste trabalho a classificação por facilidade resultou num conjunto de questões de menos desbalanceado. Com isso, essa abordagem apresentou resultados melhores por disponibilizar mais amostras da classe minoritária, o que possibilitou uma melhor generalização dos modelos dessa classificação quando comparados com os da classificação por dificuldade. Assim sendo, a **QP3** é respondida.

Outro ponto importante é que, enquanto a abordagem utilizando avaliadores humanos 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. Por fim, 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.

5.4 Premissas e limitações

A *taxa de acerto*, embora tenha sido a única *variável dependente* viável de ser escolhida num primeiro momento, dificilmente conseguirá captar integralmente a real dificuldade das questões de codificação. Outras variáveis podem ser analisadas e combinadas com a *taxa de acerto*, como por exemplo métricas de inteligibilidade textual ou até mesmo uma estimativa de dificuldade fornecida pelos próprios estudantes logo após resolverem as questões. Isso garante mais abrangência dos aspectos da dificuldade.

Além disso, as questões de codificação abordam um ou mais conceitos de programação. Essa diferença entre os conceitos abordados leva a uma diferença na ocorrência de construções no código de solução e consequentemente numa diferença nos atributos extraídos. Concomitante a isso, questões que abordem os mesmos conceitos podem apresentar diferentes níveis de dificuldade. Isso dificulta a criação de um classificador capaz de generalizar para novas questões, dificultando assim a tarefa de classificação.

Os códigos de solução utilizados foram elaborados para questões de uma disciplina introdutória de programação em Python. Logo, esses códigos são mais simples e diretos quando comparados com soluções para questões de disciplinas mais avançadas ou códigos implementados noutras linguagens de programação. É esperado, neste caso, que não haja grandes diferenças entre os códigos de solução de instrutores e estudantes, o que permitiu o uso das soluções de instrutores em nossa abordagem. Entretanto, não é garantido que essa mesma abordagem seja válida para disciplinas mais avançadas ou que utilizem outras linguagens de programação, pois haverão diferenças significativas entre as soluções de instrutores e estudantes, e consequentemente o conjunto de atributos utilizado será diferente.

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. Porém, poucas questões da base atendiam à esse critério, optou-se então pelo uso de um limiar menor, desse modo reduzindo o viés da amostra e ainda assim mantendo uma boa quantidade de questões 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 processo de ensino-aprendizagem. Nesse sentido, este artigo apresentou duas abordagens para classificar questões de escrita de código segundo sua dificuldade e facilidade, ambas expressas aqui por uma discretização da *taxa de acerto* das ques-

tões. Foi utilizado um conjunto de atributos extraídos automaticamente dos códigos de solução cadastrados pelo instrutor da disciplina, durante a criação das questões no juiz online adotado.

Como trabalho futuro, pretende-se conjugar atributos extraídos do código de solução com atributos de inteligibilidade textual extraídos do enunciado da questão. Nossa hipótese é que a dificuldade das questões também esteja relacionada com características do enunciado da questão. O uso de termos ambíguos ou técnicos pode resultar numa interpretação errônea do objetivo da questão, influenciando assim na dificuldade da questão, mesmo naqueles casos em que a implementação da solução é simples. Além disso, pretende-se também verificar se de fato existe similaridade entre os códigos de solução de instrutores e os melhores códigos de solução de estudantes.

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 e do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) à Elaine Oliveira (Processo 308513/2020-7).

Artigo Premiado Estendido

Esta publicação é uma versão estendida do 3º melhor artigo do Simpósio Brasileiro de Informática na Educação (SBIE 2020), intitulado *"Classificação de dificuldade de questões de programação com base em métricas de código"*, DOI: cbie.sbie.2020.1323.

Referências

Araujo, A., Filho, D., Oliveira, E., Carvalho, L., Pereira, F., & Oliveira, D. (2021). Mapeamento e análise empírica de misconceptions comuns em avaliações de introdução à programação. In *Anais do Simpósio Brasileiro de Educação em Computação* (pp. 123–131). Porto Alegre, RS, Brasil: SBC. doi: 10.5753/educomp.2021.14478 [GS Search]

Bez, J. L., Ferreira, C. E., & Tonin, N. (2013). Uri online judge academic: A tool for professors. In *Proceedings of the 2013 International Conference on Advanced ICT and Education* (pp. 744–747). Hainan, China: Atlantis Press. doi: 10.5753/educomp.2021.14478 [GS Search]

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, *16*(1), 321—357. doi: 10.1613/jair.953 [GS Search]

Chen, T., & Guestrin, C. (2016, Aug). Xgboost: A scalable tree boosting system. *In Proceedings*

of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. doi: 10.1145/2939672.2939785 [GS Search]

da Rocha Braz, A. C., de Carvalho, L. S. G., de Oliveira, E. H. T., de Oliveira, D. B. F., Pereira, F. D., Bittencourt, R. A., & Santana, B. L. (2021). Validação e análise de um inventário de conceitos sobre programação introdutória. In *Anais Estendidos do Simpósio Brasileiro de Educação em Computação* (pp. 27–28). [GS Search]

de Freitas Júnior, H. B., Pereira, F. D., de Oliveira, E. H. T., de Oliveira, D. B. F., & de Carvalho, L. S. G. (2020). Recomendação automática de problemas em juízes online usando processamento de linguagem natural e análise dirigida aos dados. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação* (pp. 1152–1161). doi: 10.5753/cbie.sbie.2020.1152 [GS Search]

Denny, P., Cukierman, D., & 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* (p. 13—22). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2828959.2828967 [GS Search]

de Oliveira, J., Salem, F., de Oliveira, E. H. T., Oliveira, D. B. F., de Carvalho, L. S. G., & Pereira, F. D. (2020). Os estudantes leem as mensagens de feedback estendido exibidas em juízes online? In *Anais do XXXI Simpósio Brasileiro de Informática na Educação* (pp. 1723–1732). doi: 10.5753/cbie.sbie.2020.1723 [GS Search]

Effenberger, T., Čechák, J., & Pelánek, R. (2019). Measuring difficulty of introductory programming tasks. In *Proceedings of the Sixth (2019) ACM Conference on Learning @ Scale*. New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3330430.3333641 [GS Search]

Elnaffar, S. (2016). Using software metrics to predict the difficulty of code writing questions. In *2016 IEEE Global Engineering Education Conference (EDUCON)* (pp. 513–518). doi: 10.1109/EDUCON.2016.7474601 [GS Search]

Elo, A. E. (1978). *The rating of chessplayers, past and present*. Arco Pub.

Fix, E., & Hodges Jr, J. L. (1952). *Discriminatory analysis-nonparametric discrimination: Small sample performance* (Technical Report Project 21-49-004 No. 11). Randolf Field, Texas: USAF School of Aviation Medicine.

Fonseca, S. C., Pereira, F. D., Oliveira, E. H., Oliveira, D. B., Carvalho, L. S., & Cristea, A. I. (2020). Automatic subject-based contextualisation of programming assignment lists. EDM. [GS Search]

Francisco, R., Júnior, C., & Ambrósio, A. (2016). Juiz online no ensino de programação introdutória - uma revisão sistemática da literatura. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação - SBIE)*, *27*(1), 11. doi: 10.5753/cbie.sbie.2016.11 [GS Search]

Francisco, R. E., & Ambrosio, A. P. (2015, jun). Mining an online judge system to support introductory computer programming teaching. In *SMLIR: Workshop on Tools and Technologies in Statistics, Machine Learning and Information Retrieval for Educational Data Mining* (pp. 93–98). [GS Search]

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. *Revista Brasileira de Informática na Educação*, *26*(03), 163. doi: 10.5753/rbie.2018.26.03.163 [GS Search]

Galvão, L., Fernandes, D., & Gadelha, B. (2016). Juiz online como ferramenta de apoio a

uma metodologia de ensino híbrido em programação. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação - SBIE)*, *27*(1), 140. doi: 10.5753/cbie.sbie.2016.140 [GS Search]

Halstead, M. H. (1977). *Elements of software science (operating and programming systems series)*.

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

Joy, M., & Luck, M. (1999). Plagiarism in programming assignments. *IEEE Transactions on education*, *42*(2), 129–133. doi: 10.1109/13.762946 [GS Search]

Llana, L., Martin-Martin, E., & Pareja-Flores, C. (2012). Flop, a free laboratory of programming. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research* (p. 93–99). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2401796.2401807 [GS Search]

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, *SE–2*(4), 308–320. doi: 10.1109/TSE.1976.233837 [GS Search]

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

Neves, A., Oliveira, M., França, H., Lopes, M., Reblin, L., & Oliveira, E. (2017). Pcodigo ii: O sistema de diagnóstico da aprendizagem de programação por métricas de software. *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*, *6*(1), 339. doi: 10.5753/cbie.wcbie.2017.339 [GS Search]

Pelánek, R. (2016). Applications of the elo rating system in adaptive educational systems. *Computers & Education*, *98*, 169–179. doi: 10.1016/j.compedu.2016.03.017 [GS Search]

Pereira, F., Junior, H., Rodriquez, L., Toda, A., Oliveira, E., Cristea, A., ... Isotani, S. (2021). A recommender system based on effort: Towards minimising negative affects and maximising achievement in cs1 learning. In *Intelligent Tutoring Systems: 17th International Conference, ITS 2021, Virtual Event, June 7–11, 2021, Proceedings* (p. 466). doi: 10.1007/978-3-030-80421-$3_5$1 [GS Search]

Pereira, F. D., Oliveira, E., Cristea, A., Fernandes, D., Silva, L., Aguiar, G., ... Alshehri, M. (2019). Early dropout prediction for programming courses supported by online judges. In *International Conference on Artificial Intelligence in Education* (pp. 67–72). doi: 10.1007/978-3-030-23207-$8_1$3 [GS Search]

Pereira, F. D., Oliveira, E. H. T., Oliveira, D. B. F., Cristea, A. I., Carvalho, L. S. G., Fonseca, S. C., ... Isotani, S. (2020). Using learning analytics in the amazonas: understanding students' behaviour in introductory programming. *British Journal of Educational Technology*, *51*(4), 955–972. doi: 10.1111/bjet.12953 [GS Search]

Pereira, F. D., Pires, F., Fonseca, S. C., Oliveira, E. H., Carvalho, L. S., Oliveira, D. B., & I, A. (2021). Towards a human-ai hybrid system for categorising programming problems. New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3408877.3432422 [GS Search]

Santos, P., Carvalho, L. S. G., Oliveira, E. H. T., & Oliveira, D. B. F. (2019). Classificação de dificuldade de questões de programação com base na inteligibilidade do enunciado. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação – SBIE)*, *30*(1), 1886-1895. doi: 10.5753/cbie.sbie.2019.1886 [GS Search]

Sheard, J., Simon, Carbone, A., Chinn, D., Clear, T., Corney, M., ... Teague, D. (2013). How difficult are exams? a framework for assessing the complexity of introductory programming exams. In *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136* (p. 145—154). AUS: Australian Computer Society, Inc. [GS Search]

Sheard, J., Simon, Carbone, A., Chinn, D., Laakso, M.-J., Clear, T., ... Warburton, G. (2011). Exploring programming assessment instruments: A classification scheme for examination questions. In *Proceedings of the Seventh International Workshop on Computing Education Research* (p. 33—38). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2016911.2016920 [GS Search]

Ullah, F., Wang, J., Farhan, M., Jabbar, S., Wu, Z., & Khalid, S. (2020). Plagiarism detection in students' programming assignments based on semantics: multimedia e-learning based smart assessment methodology. *Multimedia tools and applications*, *79*(13), 8581–8598. doi: 10.1007/s11042-018-5827-6 [GS Search]

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

Vargas, A. P., Penna, R., Evandro, Botelho, S. S. C., Tonin, N. A., & Bez, J. L. (2018). A multi-dimensional elo model for matching learning objects. In *2018 IEEE Frontiers in Education Conference (FIE)* (p. 1-9). doi: 10.1109/FIE.2018.8658847 [GS Search]

Wolpert, D. H. (1992). Stacked generalization. *Neural Networks*, *5*(2), 241–259. doi: 10.1016/S0893-6080(05)80023-1 [GS Search]

Zaffalon, F., Vargas, A. P., Souza, R. L., Penna, R., Bez, J. L., Tonin, N. A., & Botelho, S. S. C. (2019). Um estudo comparativo entre dois modelos que estimam a habilidade dos estudantes: Elo e teoria de resposta ao item. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação - SBIE)*, *30*(1), 459–468. doi: 10.5753/cbie.sbie.2019.459 [GS Search]

Zordan Filho, D. L., de Oliveira, E. H. T., de Carvalho, L. S. G., Pessoa, M., Pereira, F. D., & de Oliveira, D. B. F. (2020). Uma análise orientada a dados para avaliar o impacto da gamificação de um juiz on-line no desempenho de estudantes. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação* (pp. 491–500). doi: 10.5753/cbie.sbie.2020.491 [GS Search]