Previsão de indicadores de dificuldade de questões de programação a partir de métricas do código de solução

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

[1]Instituto de Computação – Universidade Federal Amazonas (UFAM)
[2]Departamento de Ciência da Computação – Universidade Federal de Roraima (UFRR)

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

Resumo. *Juízes Online (JOs) são usados em disciplinas de programação para disponibilizar listas de exercícios e avaliações práticas. Porém, não é uma tarefa fácil criar listas selecionando exercícios com níveis de dificuldade adequados a cada turma. Uma forma de ajudar o professor nessa tarefa é apresentar indicadores de dificuldade para cada questão, tais como taxa de acerto, tempo de desenvolvimento, número de submissões, etc. No entanto, para questões novas, que ainda não foram solucionadas por nenhum aluno, não é possível calcular tais indicadores. Dessa forma, este trabalho usa aprendizagem de máquina para prever nove indicadores de dificuldade a partir de evidências extraídas de códigos de solução cadastrados pelos professores no momento de criação das questões no JO. Como exemplo de resultado, o maior f1-score obtido foi para prever a taxa de acerto (0,920) em uma classificação binária e o menor foi para o número de submissões (0,560).*

Abstract. *Online Judges (OJs) are used in programming courses to construct assigments and practical exams. However, it is not easy to design assigments with balanced exercises according to difficulty levels. One way to assist the instructor is to present difficulty indicators for each question, such as success rate, average implementation time, number of submissions, etc. However, for new questions that have yet to be solved by students, it is not possible to calculate such indicators. Thus, this paper uses machine learning to predict nine difficulty indicators from metrics extracted from solution codes provided by teachers when they crete exercises in the OJ. As examples of results, the binary classification of success rate obtained the highest f1-score (0.920), and the binary classification of the number of submissions obtained the lowest score (0.560).*

1. Introdução

Este trabalho aborda a complexidade e a dificuldade de exercícios de programação propostos a estudantes em um JO. Primeiramente, iremos clarificar a terminologia para, em seguida, estabelecer os objetivos do trabalho.

1.1. Complexidade e dificuldade de questões de programação

Juízes online (JOs) são ambientes que oferecem exercícios de escrita de códigos de programação passíveis de correção automática. Cada exercício é composto por três partes:

(i) *enunciado*: é aquilo que é mostrado aos alunos: texto em linguagem natural, equações, imagens, vídeos, entre outros elementos [Pelánek et al. 2021]. O enunciado pode ainda ser decomposto em especificação, dicas, exemplos de casos de teste (entradas e saídas correspondentes) e arquivos de apoio; (ii) *casos de teste*: são pares ordenados <entrada, saída> usados para testar os códigos de solução submetidos pelos alunos; (iii) *código do instrutor*: é um código de referência, desenvolvido pelo instrutor para resolver a questão, registrado para fins de documentação, e não para corrigir o código de solução do aluno.

Após cadastrado no JO, o exercício poderá ser disponibilizado aos estudantes, que tentarão respondê-lo por meio da escrita, teste e submissão de código no Ambiente de Desenvolvimento Integrado (IDE) normalmente embutido no JO. A partir da interação dos estudantes com o exercício, são gerados dados de log, armazenados no JO, que constituem um quarto conjunto de dados associado a cada exercício do JO.

Em JOs, instrutores precisam elaborar listas de exercícios balanceadas, com questões organizadas em ordem crescente de dificuldade, para promover uma curva de aprendizado mais suave [Effenberger et al. 2019]. Para isso, é desejável que o JO informe o grau de dificuldade das questões já resolvidas, bem como estime a dificuldade de novas questões com base na similaridade com outras já resolvidas [Effenberger et al. 2019]. Entretanto, como definir "dificuldade" em termos mais objetivos? Quais características de um exercício (enunciado, casos de teste, código do instrutor, dados de interação) contribuem para uma maior "dificuldade" ou "complexidade"? Os termos "dificuldade" e "complexidade" podem ser usados intercambiavelmente? É necessário esclarecer estes dois conceitos antes de apresentar formalmente os objetivos deste trabalho.

[Liu and Li 2012] encontraram e sistematizaram a variedade de entendimentos sobre complexidade e dificuldade de tarefas humanas, em diferentes contextos. Eles propõem esta distinção: "a complexidade envolve as características objetivas de uma tarefa, enquanto a dificuldade envolve a interação entre a tarefa, o executor da tarefa e as características do contexto". Diferenciação semelhante é proposta por [Beckmann et al. 2017]: "a complexidade é uma qualidade determinada pelas demandas cognitivas que as características da tarefa e da situação impõem; já a dificuldade representa o nível quantificável do sucesso de uma pessoa em lidar com tais demandas".

A partir desses trabalhos, [Sheard et al. 2013] e [Pelánek et al. 2021] instanciam esses dois conceitos no desenvolvimento de sistemas de aprendizagem. Segundo eles, a *complexidade* de uma questão é uma característica intrínseca ao item de questão, agregando aspectos do item que influenciam como os alunos o resolvem, tais como: extensão do enunciado, número de conceitos de programação trabalhados, número de estruturas de controle necessárias para a solução, complexidade ciclomática, entre outros. Já a *dificuldade* descreve o esforço dos estudantes em resolver uma questão. Em questões de programação, algumas métricas de dificuldade utilizadas são taxa de respostas incorretas (ou corretas), tempo de resposta mediano, número de tentativas, proporção de uso de dicas, e número de edições para construir a resposta.

Uma forma prática de diferenciar complexidade e dificuldade é considerar os dados usados para computar uma medida específica [Pelánek et al. 2021]: se apenas a descrição da questão for usada, então temos uma medida de complexidade; se for usado apenas o log de ações do aluno, então temos uma medida de dificuldade.

1.2. Objetivo e questões de pesquisa

Um problema comum enfrentado por educadores de programação é elaborar uma lista de exercícios ou uma avaliação com um conjunto balanceado de questões em termos de dificuldade. Não é desejável que a atividade seja composta por questões majoritariamente fáceis ou majoritariamente difíceis. Porém, a partir da definição acima, o instrutor somente conhecerá a dificuldade da questão depois que ela for resolvida pelos estudantes, momento que não atende mais aos propósitos iniciais, quando a questão é inteiramente nova no banco de itens do JO. Dessa forma, o objetivo deste trabalho é utilizar métricas de complexidade extraídas a partir de questões de programação para classificar a dificuldade dessas questões (duas classes: fácil ou difícil). Para atingir esse objetivo, estipulamos as seguintes questões de pesquisa:

QP1 : qual a correlação entre diferentes indicadores de dificuldade?
QP2 : quais indicadores de dificuldade obtém melhor precisão de previsão?

Este artigo está organizado desta forma. Na Seção 2, apresentamos os trabalhos relacionados. Na Seção 3, explicamos as variáveis envolvidas e a metodologia adotada. Na Seção 4, caracterizamos o banco de questões utilizados neste estudo. Na Seção 5, explicamos o tratamento efetuado nos dados. Na Seção 6, apresentamos e discutimos os resultados encontrados. Por fim, na Seção 7, tecemos as conclusões e trabalhos futuros.

2. Trabalhos Relacionados

Uma forma simples de classificar a complexidade de uma questão de programação é utilizar a opinião do próprio instrutor [Llana et al. 2012, Francisco and Ambrosio 2015, Denny et al. 2015]. No entanto, um problema dessa abordagem é a discordância da quantidade de níveis de complexidade entre as escalas adotadas por esses trabalhos: variam de três a seis níveis. Além disso, a classificação subjetiva apresenta outros inconvenientes: o julgamento do avaliador humano depende de sua experiência prévia como instrutor e do seu contexto educacional; o tempo e esforço para cadastrar uma questão no JO são maiores; e, como evidenciado por [Meisalo et al. 2004], a complexidade estimada pelo instrutor geralmente não condiz com a dificuldade experimentada pelos estudantes. Portanto, a avaliação subjetiva não interessa para um cenário que precise de escalabilidade, como uma disciplina ministrada para vários cursos, com uma metodologia baseada fortemente em exercícios práticos corrigidos automaticamente com o apoio de um JO.

Outras abordagens, como [Whalley and Kasto 2014] e [Elnaffar 2016], verificaram a correlação entre a dificuldade de questões de escrita de código e um pequeno conjunto de métricas de software (cinco a sete), extraídas a partir do código do instrutor. A métrica de dificuldade era o percentual de acerto de cada exercício. Ambos os trabalhos encontraram correlação significativa (p <0,001) e alta (r >0,83) entre dificuldade e duas métricas de software: complexidade ciclomática e número de operadores. Porém, utilizaram um número muito reduzido de questões (onze e dez), tornando questionável o pressuposto de normalidade dos dados para aplicação da correlação de Pearson, adotada por ambos (correlação de Spearman seria mais indicado). Além disso, as métricas de software foram calculadas manualmente pelos pesquisadores, tornando a solução não escalável. Ainda assim, esses dois trabalhos avançaram no sentido de proporem métricas objetivas para estimar a complexidade de exercícios de programação.

Em vez do código do instrutor, [Santos et al. 2019] extraíram métricas do enunciado para estimar a dificuldade, indicada pela taxa de acerto. Entre elas, havia o índice de inteligibilidade textual Flesch e a proporção de incidência de algumas classes de palavra, tais como adjetivos, advérbios, pronomes, verbos e operadores lógicos. Eles analisaram 187 questões resolvidas por 800 estudantes em um JO. Como resultado, encontraram uma correlação fraca. Porém, eles verificaram que questões com enunciado de leitura difícil normalmente produzem baixas taxas de acerto, enquanto questões com enunciados de fácil leitura nem sempre produzem altas taxas de acerto.

[Effenberger et al. 2019] tentaram estimar a dificuldade usando métricas de complexidade. Eles analisaram 124 questões de programação textual e outras 162 de programação em bloco, resolvidas por milhares de estudantes (varia conforme o *dataset*). Eles usaram apenas duas métricas de complexidade: número de linhas de código e número de conceitos de programação envolvidos. A dificuldade foi medida pela média entre três medidas: taxa de falha (% de alunos que erraram a questão), tempo mediano de resolução (somente dos alunos que acertaram a questão) e número mediano de tentativas (submissão de código para avaliação automática). Os autores identificaram que nem o número de linhas de código, nem de conceitos são um preditor preciso de dificuldade. O número de conceitos mostrou-se uma medida mais útil para dividir o banco de questões em conjuntos de problemas, enquanto o número de linhas de código foi mais útil para ordenar as questões em grau de dificuldade nos conjuntos de problemas.

Similarmente, [Lima et al. 2021] classificaram a dificuldade de questões de programação, indicada pela taxa de acerto, mas desta vez usando, como métricas de complexidade, 92 atributos de software extraídos dos códigos do instrutor de 404 questões do JO abordado. Eles usaram os módulos Radon e Tokenize da linguagem Python para extrair as métricas de complexidade, tais como complexidade ciclomática, número de linhas de código, linhas lógicas, linhas em branco, comentários, condicionais, laços, operadores, funções embutidas, etc. Entre eles, o atributo mais importante encontrado pelo modelo classificador foi o *has_loops*, que indica se o código de solução apresenta laços de repetição. Uma limitação desse trabalho foi ter usado apenas uma variável (taxa de acerto) como indicador de dificuldade.

Este trabalho utiliza o mesmo conjunto ampliado de 92 métricas de complexidade adotado por [Lima et al. 2021] para classificar a dificuldade de questões de programação, que neste trabalho será expressa por nove indicadores, detalhados na Seção 3.1, em vez de apenas duas indicadores, como em [Effenberger et al. 2019].

3. Metodologia

Nesta seção, serão apresentados os indicadores de dificuldade considerados neste trabalho, as métricas de complexidade usadas para prever os indicadores de dificuldade, e as estratégias de previsão adotadas.

3.1. Variáveis dependentes: os indicadores de dificuldade

Como já dito, o objetivo deste trabalho é prever indicadores de dificuldade de exercícios de programação a partir de métricas de complexidade do código dos instrutores. Para compor a lista de indicadores, identificaram-se variáveis que podem ser calculadas a partir dos registros de *log* de JOs e com relação direta com os níveis de esforço e dificuldade

sentidos pelos alunos durante o desenvolvimento do código de solução. A partir desses critérios, foram selecionados nove indicadores de dificuldade: (i) *taxa de acerto*, percentual de alunos que acertaram o exercício; (ii) *tempo de implementação*, mediana do tempo usado pelos alunos para resolver o exercício; (iii) *número de testes*, mediana da quantidade de vezes os alunos testaram seus códigos com suas próprias entradas no exercício; (iv) *número de submissões*, mediana do número de vezes em que os alunos submeteram seus códigos para avaliação de corretude pelo JO; (v) *número de consultas*, soma do número de testes com o número de submissões; (vi) *taxa de consultas com erro*, percentual de testes e submissões (consultas) que apresentaram erro sintático ou lógico durante a resolução do exercício; (vii) *tamanho do log*, mediana do número total de registros de *logs* gerados pelos alunos durante a resolução do exercício; (viii) *eventos de remoção*, mediana da quantidade de vezes em que as teclas *delete* e *backspace* foram pressionadas pelos alunos durante a resolução do exercício; (ix) *quantidade de mudanças*, mediana do número de registros de *logs* que denotam mudanças feitas no código do exercício.

Desses indicadores, a taxa de acerto foi selecionada a partir de [Lima et al. 2021]; já o tempo de implementação e número de submissões foram selecionados a partir de [Effenberger et al. 2019]. Os demais indicadores estão sendo propostos pelo presente trabalho, com base nos dois anteriores. Por exemplo, os indicadores tamanho do log, eventos de remoção e quantidade de mudanças são derivados do indicador número de edições, sugerido por [Effenberger et al. 2019].

Com exceção da taxa de acerto e da taxa de consultas com erro, os demais indicadores utilizam a mediana como medida de centralidade. Por exemplo, considerando o indicador *número de testes*, a quantidade de testes feitos para um dado exercício é diferente para cada aluno, por isso a mediana é usada para representar esse indicador. A mediana foi escolhida porque os indicadores selecionados possuem um limite inferior (zero), mas não possuem um limite superior. Por exemplo, os alunos podem testar seus códigos quantas vezes quiserem, sem penalidades ou milites no JO. Dessa forma, como as distribuições desses indicadores tendem a ser assimétricas [Van Der Linden 2009], a mediana é uma medida de centralidade preferível à média [Pelánek et al. 2021].

A partir do cálculo dos indicadores, cada exercício da base passa a possuir um conjunto de valores representando um indício da dificuldade de resolução do exercício. No entanto, apesar dos valores desses indicadores serem facilmente calculados para um dado exercício a partir dos logs dos JOs, eles só estão disponíveis se um conjunto de alunos já tiver tentado solucionar esse exercício. Para exercícios criados recentemente, que não foram solucionados por nenhum aluno, não é possível calcular os indicadores apresentados. Desta forma, neste trabalho apresentamos uma estratégia de previsão desses indicadores para exercícios que ainda não foram solucionados por nenhum aluno.

No entanto, experimentos preliminares [Lima et al. 2021] mostraram que prever o valor exato dos indicadores para cada exercício, usando estratégias de regressão, tende a não dar bons resultados. Desta forma, neste trabalho, a previsão de indicadores foi mapeada para um problema de classificação binária. Para viabilizar essa classificação, a escala de cada indicador foi transformada em duas classes binárias distintas, tendo suas respectivas medianas como critério de divisão de classes.

Para exemplificar, a mediana do indicador *número de testes* é calculada com base em todos os exercícios que já foram solucionados por um dado conjunto de alunos. Quando um novo exercício for criado no JO, estratégias de classificação binária serão utilizadas para identificar se o valor do *número de testes* para esse exercício será abaixo ou acima da mediana encontrada para os demais exercícios já presentes na base.

3.2. Variáveis preditoras: métricas de complexidade de código

No formulário de criação de exercícios em JOs, normalmente há um campo onde os instrutores informam um exemplo de *código de solução* para o exercício em criação. Esse código é usado, por exemplo, para ajudar a criar os casos de teste para correção automática. Neste trabalho, as métricas de complexidade (variáveis preditoras) foram extraídas dos códigos de solução dos exercícios. Mais precisamente, foram utilizadas as mesmas 92 métricas adotadas por [Lima et al. 2021], que disponibilizaram seus *scripts* para extração desses atributos.

Alguns exemplos de variáveis preditoras são: (1) número de linhas lógicas de código, (2) quantidade de estruturas de repetição, (3) quantidade de identificadores, (4) quantidade de funções de entrada de dados, (5) número de linhas de documentação, (6) número de operadores distintos, (7) quantidade de operadores relacionais distintos, (8) quantidade de operandos, (9) quantidade de funções de conversão entre tipos de dados, e (10) quantidade de funções da biblioteca padrão do Python. Após extrair tais métricas, elas foram utilizadas como evidências em estratégias de classificação delineadas para prever as classes binárias geradas a partir das variáveis dependentes. Com isso, este trabalho relaciona métricas de complexidade com métricas de dificuldade.

3.3. Algoritmos de classificação e calibração dos hiperparâmetros

Com a definição das variáveis independentes e a extração das métricas de complexidade que serão usadas como variáveis dependentes, foi selecionado um conjunto de classificadores das bibliotecas *sklearn* e *xgboost* para condução do processo de previsão proposto neste trabalho. Os classificadores testados foram: RandomForest, KNeighbours, ExtraTree, AdaBoost, LogisticRegression, LinearSVC, SVC e XGBoost. Esses algoritmos são recomendados para um conjunto de dados tabulares como o usado neste trabalho, com variáveis significativas e sem estrutura temporais ou espaciais multiescalares [Chen and Guestrin 2016, Lundberg et al. 2020]. Cada um deles utiliza um seletor de variáveis independentes distinto. Isso significa que cada classificador pode selecionar as variáveis que melhor se adéquam ao seu algoritmo de classificação.

Para treinar cada classificador, foram utilizadas diferentes combinações de hiperparâmetros selecionados automaticamente através do método RandomSearchCV da biblioteca *sklearn*. Esse seletor de hiperparâmetros utiliza um espaço de busca de hiperparâmetros possíveis, e testa combinações aleatórias até atingir um limite de tentativas. No experimento conduzido neste trabalho, o limite de tentativas foi 67.

3.4. Avaliação dos classificadores

Neste trabalho, queremos prever indicadores de dificuldade de exercícios, para os instrutores criarem avaliações práticas balanceadas por níveis de dificuldade. Para escolher as métricas que mais se adéquam a esse objetivo, é necessário avaliar o efeito de previsões errôneas, tanto em relação a falsos-positivos quanto a falsos-negativos.

No caso de uma classificação falso-positiva, que é quando os classificadores apontam que o exercício possui indicadores elevados de dificuldade quando na realidade ele não deveria ter, poderia induzir o professor a compor provas mais fáceis do que gostaria, prejudicando a avaliação da turma. Por outro lado, uma classificação falso-negativa poderia induzir o professor a criar avaliações mais difíceis do que às adequadas às suas turmas, o que também prejudicaria a avaliação das turmas.

Desta forma, como tanto falsos-positivos quanto falsos-negativos causam prejuízo aos usuários finais, optamos por usar a métrica f1-score, que é a média harmônica entre a precisão e a revocação e que é adequada aos casos em que falsos-positivos e falsos-negativos possuem impactos similares dentro do problema de classificação apresentado. Como critério de desempate, optou-se por usar a métrica acurácia.

Para validar as métricas extraídas, usou-se a validação cruzada com 4 partições. Esse passo adiciona mais confiabilidade às métricas coletadas, pois verifica se os classificadores funcionam mesmo selecionando partições aleatórias dos dados. Foi escolhido um número de partições igual a 4 porque, conforme será detalhado posteriormente, o número de questões usadas nos experimentos foi 379, que é uma quantidade relativamente pequena para uma divisão em 10 partições, mais comum neste tipo de experimento.

4. Caracterização dos dados

Os dados dos exercícios de programação utilizados neste trabalho foram extraídos do JO CodeBench[1] desenvolvido pela Universidade Federal do Amazonas. O JO em questão mantém uma base de logs com registros das atividades dos alunos, de onde foram extraídos os logs de resolução de exercícios por parte de alunos de turmas de Introdução à Programação de Computadores (CS1), ministradas para cursos STEM (Ciência, Tecnologia, Engenharia e Matemática), entre os anos 2017 e 2019.

Para este estudo, foram considerados apenas exercícios resolvidos por alunos durante avaliações somativas presenciais, durante as quais os alunos precisam resolver as questões individualmente, sem consulta a colegas ou à materiais de aula, e com a presença de um professor e um tutor. Os exercícios usados nas atividades formativas e nas avaliações somativas de 2020 e 2021 foram descartadas, porque foram resolvidas remotamente[2], não havendo garantia de que tenham sido resolvidas sem ajuda externa. Dessa forma, temos uma maior probabilidade de que cada resposta a cada exercício corresponde a uma interação única entre o aluno e o exercício.

No JO utilizado, os estudantes podem enviar seu código-fonte para correção automática quantas vezes quiserem, sem penalidades. Além disso, os problemas são listados sem indicação clara de dificuldade para o aluno.

5. Tratamento de dados

5.1. Filtro de tentativas e questões

Como já dito, foram consideradas apenas questões aplicadas em exames presenciais. Para que uma tentativa de solução de aluno pudesse ser computada como uma amostra válida

[1] https://codebench.icomp.ufam.edu.br
[2] As avaliações somativas de 2020 e 2021 foram realizadas remotamente devido à pandemia de covid-19.

de solução da questão, os seguintes filtros foram aplicados: ter pelo menos uma submissão e tempo de resolução não nulo.

Questões resolvidas por menos de 16 alunos foram removidas, pois, como [Lima et al. 2021] demonstram, questões com menos de 16 soluções não produzem resultados estáveis. Ou seja, abaixo desse limiar, torna-se estatisticamente inviável diferenciar se os dados de desempenho (métricas de dificuldades) se referem a observações anedóticas dos poucos estudantes que resolveram a questão, ou a uma descrição confiável da questão. Após aplicar esse filtro, foram selecionadas 379 questões de programação para a classificação de dificuldade. Ao todo, 1474 alunos resolveram essas questões.

5.2. Soluções dos instrutores

Um obstáculo encontrado na realização do experimento foi a ausência de soluções do instrutor em algumas das questões selecionadas após aplicação dos filtros descritos na subseção anterior. Isso aconteceu porque o código de solução para uma questão é um campo opcional no JO utilizado, cuja correção se baseia em comparação de casos de teste. Dessa forma, verificou-se que 81 questões das 379 questões com dados estáveis de desempenho não possuíam código de solução cadastrado.

Para não descartar essas questões do experimento, utilizou-se o código de alunos que resolveram corretamente a questão como código de solução do instrutor. Os critérios de seleção do código de referência entre os códigos de solução dos alunos foram estes: (1) solução correta; (2) tempo de solução maior que um minuto; e (3) código com menor tempo de implementação.

6. Resultados

6.1. Distribuição e correlação entre as variáveis dependentes

Os gráficos da Figura 1 mostram a distribuição dos valores de cada um dos nove indicadores de dificuldade estudados, extraídos dos dados de interação dos estudantes com as 379 questões de programação filtradas. Verifica-se que nenhum dos indicadores apresenta distribuição normal. Alguns indicadores (por exemplo, eventos de remoção, número de testes e tempo de implementação) apresentam bastante *outliers*, limitados pelo tempo de duração da avaliação de onde os exercícios foram filtrados (máximo de duas horas).

Já o mapa de calor da Figura 2 apresenta a correlação de Spearman entre os indicadores de dificuldade (diagonal inferior) e os valores-*p* correspondentes (diagonal superior). Observam-se correlações fortes[3] e positivas entre alguns indicadores, como entre quantidade de remoções e tamanho do *log* (0,82, $p \ll$0,001), o que se explica pelo evento remoção ser registrado em arquivo de *log*. Também se observam correlações fracas, como entre taxa de consultas com erro e a maioria dos demais indicadores.

Além disso, foram encontras correlações moderadas e negativas, como entre taxa de acerto e número de testes ($-$0,58; $p \ll$0,001). Uma leitura apressada pode gerar a crença de que deveria haver uma correlação positiva entre os dois indicadores: quanto mais se testa uma questão, mais ela é acertada. Porém, essa leitura tem o aluno como foco, ao passo que o problema abordado tem o exercício como unidade de observação.

[3]Os rótulos “forte”, “moderada” e “fraca” seguem a categorização proposta em `gpestatistica.netlify.app/blog/correlacao/`

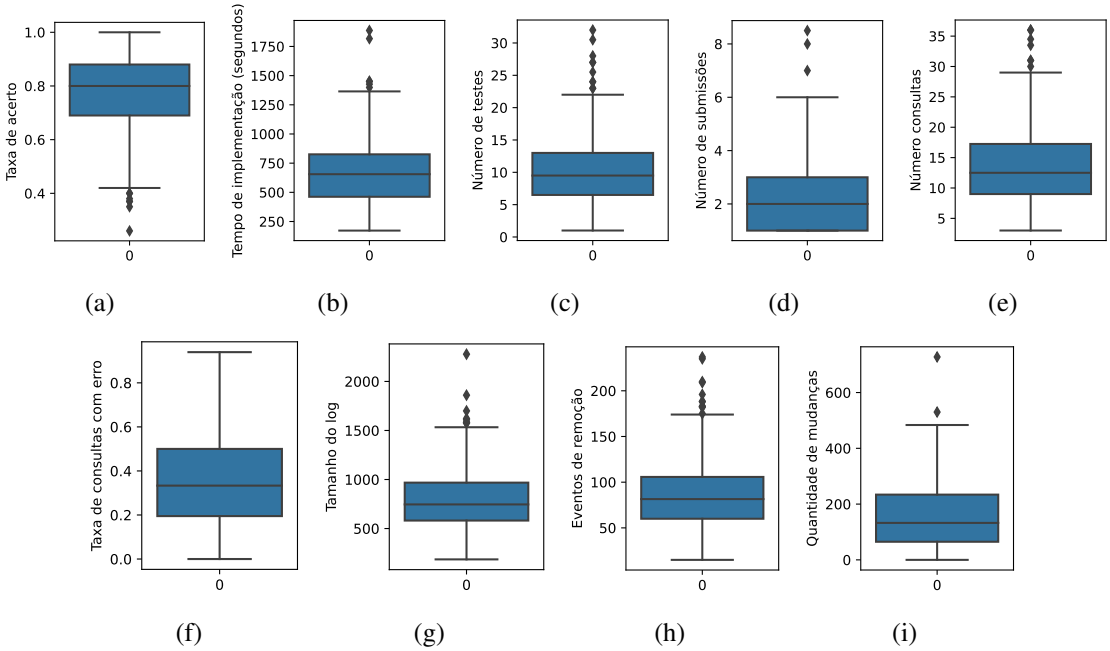

Figura 1. Distribuição dos indicadores de dificuldade: (a) taxa de acerto, (b) tempo de implementação (segundos), (c) número de testes, (d) número de submissões, (e) número de consultas, (f) taxa de consultas com erro (g) tamanho dos logs, (h) eventos de remoção, e (i) quantidade de mudanças

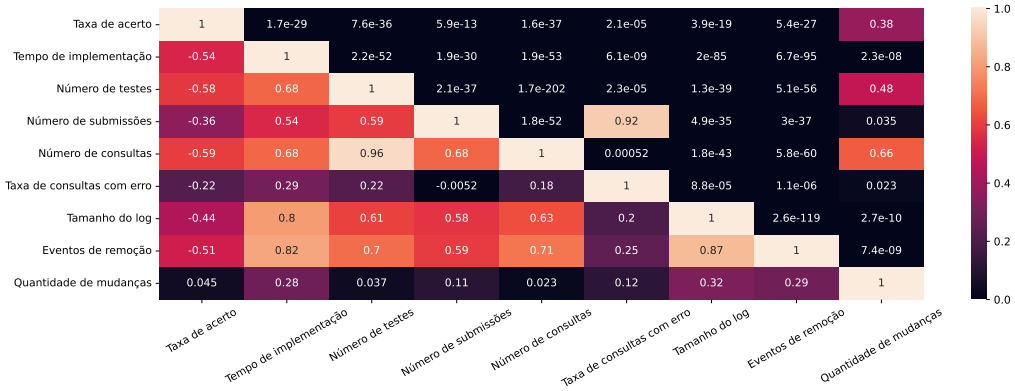

Figura 2. Correlações de Spearman e valores-*p* das variáveis dependentes

Assim, o que a correlação nos diz de fato é que exercícios mais acertados moderadamente requerem menos testes pelos alunos.

Outro achado interessante foi a correlação entre número de testes e de consultas (0,96; $p \ll$0,001). O estudo tinha como premissa a observação anedótica apontada por [Lima et al. 2021], de que nem o número de testes, nem o número de submissões seriam indicadores confiáveis de dificuldade, pois nem todos os estudantes seguem boas práticas de programação: testar exaustivamente o código no IDE antes de submeter um código livre de erros sintáticos para o avaliador do JO. Porém, o que se observou foi que os alunos testam muito mais código (mediana de 9,5 testes por questão e por aluno) do que submetem (mediana de 2,0 submissões por questão e por aluno). Por isso, a correlação do número de consultas (testes + submissões) teve correlação quase perfeita com o número de testes. Ou seja, no conjunto de dados estudados, é mais oportuno trabalhar diretamente com o indicador *número de testes* do que com o indicador derivado, *número de consultas*.

6.2. Resultados da classificação

A Tabela 1 apresenta os resultados dos classificadores aplicados à previsão da dificuldade de questões de programação, segundo os nove indicadores estudados, ordenador por *f1-score*. Observa-se que a taxa de acerto é o indicador de dificuldade com melhores *f1-score* e acurácia. Ou seja, é o indicador com maior precisão de previsão.

Tabela 1. Resultados dos classificadores

Variável dependente	Classificador	F1-Score	Acurácia
Taxa de acerto	SVC	0,920	0,852
Taxa de consultas com erro	SVC	0,751	0,783
Tempo de Implementação	XGBoost	0,732	0,741
Tamanho do Log	XGBoost	0,725	0,746
Eventos de Remoção	LinearSVC	0,690	0,717
Quantidade de Mudanças	XGBoost	0,642	0,654
Testes	XGBoost	0,618	0.662
Consultas	XGBoost	0,608	0,630
Submissões	XGBoost	0,560	0,743

Outros três indicadores também tiveram um bom desempenho na classificação: taxa de consultas com erro, tempo médio de implementação e tamanho do *log* de resolução. Tal resultado aponta que tais indicadores podem ser melhor investigados no futuro, pois exprimem uma parcela do esforço que o exercício demandou dos alunos, oferecendo ao instrutor uma visão complementar à taxa de acerto.

7. Conclusão e Trabalhos Futuros

Este artigo teve como objetivo prever a dificuldade de questões de programação antes de elas serem respondidas pelos estudantes. Para isso, foram utilizadas métricas de complexidade extraídas dos códigos de solução de instrutores para métricas de dificuldade, tais como a taxa de acerto e a taxa de consultas com erro. No estudo, verificou-se que quatro indicadores de dificuldade (taxa de acerto, taxa de consultas com erro, tamanho do *log* de resolução e tempo de implementação) têm melhor precisão de previsão a partir de métricas de dificuldade extraídas das soluções dos instrutores.

Como trabalho futuro, pode-se aliar métricas extraídas do código com métricas do enunciado para atingir previsões mais precisas. Além disso, a proximidade das soluções dos alunos com o modelo de solução do instrutor poderia ser investigada para selecionar soluções alternativas para questões que não têm um modelo de solução.

Agradecimentos

O presente trabalho é decorrente do projeto de Pesquisa e Desenvolvimento (PD) 001/2020, firmado entre a Fundação da Universidade do Amazonas e FAEPI, que conta com financiamento da Samsung, usando recursos da Lei de Informática para a Amazônia Ocidental (Lei Federal nº 8.387/1991), estando sua divulgação de acordo com o previsto no artigo 39.º do Decreto nº 10.521/2020. Elaine Oliveira também recebeu apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) (Processo 308513/2020-7).

Referências

Beckmann, J. F., Birney, D. P., and Goode, N. (2017). Beyond psychometrics: the difference between difficult problem solving and complex problem solving. *Frontiers in psychology*, 8:1739.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA. Association for Computing Machinery.

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., Cechák, J., and Pelánek, R. (2019). Difficulty and complexity of introductory programming problems. In *Educational Data Mining in Computer Science Education (CSEDM)*.

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

Francisco, R. E. and Ambrosio, A. P. (2015). 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*, pages 93–98.

Lima, M. A., Carvalho, L. S., Oliveira, E. H., Oliveira, D. B., and 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.

Liu, P. and Li, Z. (2012). Task complexity: A review and conceptualization framework. *International Journal of Industrial Ergonomics*, 42(6):553–568.

Llana, L., Martin-Martin, E., and Pareja-Flores, C. (2012). Flop, a free laboratory of programming. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, Koli Calling '12, page 93–99, New York, NY, USA.

Lundberg, S., Erion, G., Chen, H., DeGrave, A., Prutkin, J., Nair, B., Katz, R., Himmelfarb, J., Bansal, N., and Lee, S.-I. (2020). From local explanations to global understanding with explainable ai for trees. *Nature machine intelligence*, pages 56–67.

Meisalo, V., Sutinen, E., and Torvinen, S. (2004). Classification of exercises in a virtual programming course. In *34th Annual Frontiers in Education (FIE 2004)*, pages S3D–1.

Pelánek, R., Effenberger, T., and Čechák, J. (2021). Complexity and difficulty of items in learning systems. *International Journal of Artificial Intelligence in Education*, pages 1–37.

Santos, P., Carvalho, L. S. G., Oliveira, E. H. T., and Oliveira, D. B. F. (2019). Classificação de dificuldade de questões de programação com base na inteligibilidade do enunciado. *Simpósio Brasileiro de Informática na Educação*, 30(1):1886–1895.

Sheard, J., Simon, 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 *Proceedings of the 15th Australasian Computing Education Conference*, volume 136 of *ACE '13*, pages 145—-154, AUS.

Van Der Linden, W. J. (2009). Conceptual issues in response-time modeling. *Journal of Educational Measurement*, 46(3):247–272.

Whalley, J. and Kasto, N. (2014). How difficult are novice code writing tasks? a software metrics approach. In *Proceedings of the Sixteenth Australasian Computing Education Conference*, volume 148, pages 105–112.