

Uso de atributos de código para classificação da facilidade de questões de codificação

Marcos A. P. de Lima
Instituto de Computação -
Universidade Federal do Amazonas
(UFAM)
Manaus, Amazonas, Brasil
marcos.lima@icomp.ufam.edu.br

Leandro S. G. de Carvalho
Instituto de Computação -
Universidade Federal do Amazonas
(UFAM)
Manaus, Amazonas, Brasil
galvao@icomp.ufam.edu.br

Elaine Harada T. Oliveira
Instituto de Computação -
Universidade Federal do Amazonas
(UFAM)
Manaus, Amazonas, Brasil
elaine@icomp.ufam.edu.br

David B. F. Oliveira
Instituto de Computação -
Universidade Federal do Amazonas
(UFAM)
Manaus, Amazonas, Brasil
david@icomp.ufam.edu.br

Filipe Dwan Pereira
Departamento de Ciência da
Computação - Universidade Federal
de Roraima (UFRR)
Manaus, Amazonas, Brasil
filipe.dwan@ufr.br

RESUMO

No ensino de programação, é comum o uso de Ambientes de Correção Automática de Código (ACACs). Esses apresentam uma alta diversidade de exercícios de programação que requerem que o estudante elabore um código como solução. Contudo, um obstáculo é classificar a facilidade ou dificuldade desses exercícios de modo que sejam apresentados conforme o nível de conhecimento do aluno, ou para que durante uma avaliação formulada por sorteio aleatório de questões, os alunos possam receber exercícios com nível de dificuldade semelhantes. Nesse sentido, este trabalho apresenta um método para classificar automaticamente o grau de facilidade de questões de codificação em Python, usando para isso atributos extraídos de códigos de solução. Foram analisadas 354 questões, agrupadas em 07 conjuntos segundo os tópicos abordados, que foram previamente definidos na ementa de uma disciplina de introdução à programação. Essas questões foram aplicadas em *exames presenciais* de turmas ministradas entre os anos de 2017 e 2019. Entre os resultados, verificou-se que a facilidade das questões pode ser estimada por meio de atributos do código de solução dos instrutores, com f1-score de 0,94 e acurácia de 0,92. Além disso, o conjunto de atributos relevantes para classificação variou conforme o tópico de programação abordado pelas questões.

PALAVRAS-CHAVE

facilidade de questões, questões de codificação, atributos de código, classificação, ensemble

1 INTRODUÇÃO

Aprender programação requer muita prática, em grande parte através de questões de codificação [1, 10, 15, 21, 23, 29, 30, 34, 44]. Para um instrutor, a correção manual das soluções elaboradas pelos estudantes, além de desgastante, é por vezes demorada [2, 9, 29, 30]. Nesse contexto, os Ambientes de Correção Automática de Códigos (ACACs) vêm sendo largamente empregados [2, 5, 9, 31, 36, 41, 44]. Tais ambientes têm como função apresentar ao estudante uma tarefa (questão de codificação) a ser resolvida. Cabe ao estudante elaborar uma solução (código) que atenda às especificações de entradas e saídas do problema. Feito isso, o estudante deve submeter sua solução para que o ACAC faça uma verificação quanto à correção funcional por meio de um conjunto de casos de testes previamente definidos [12, 29, 33, 41, 44]. Por fim, o ACAC gera uma mensagem indicando se o estudante obteve ou não sucesso com sua solução.

Tais ambientes vêm sendo empregados em diversos tipos de atividades, como por exemplo em *trabalhos práticos* ou *exames presenciais* [2, 4, 9, 12, 26, 30, 33]. Em *exames presenciais*, geralmente ocorre o sorteio aleatório de um número predefinido de questões, dentre um conjunto de questões previamente selecionadas por um instrutor [17]. Refletindo um pouco sobre essa abordagem, é fácil perceber sua falha, pois o sorteio aleatório não garante equivalência no exame. Enquanto alguns estudantes podem ser beneficiados ao receberem um subconjunto de questões mais *fáceis*, outros podem ser prejudicados ao receberem um subconjunto de questões mais *difíceis* do que as recebidas pelos demais colegas.

O sorteio aleatório, apesar de possivelmente falho, se justifica pela tentativa de inibir ou dificultar comportamentos desonestos por parte dos estudantes [16, 31, 35, 39, 40]. Ocorre que geralmente os exames são realizados em laboratórios de informática com espaço físico restrito, onde os estudantes ficam dispostos lado a lado, facilitando conversas paralelas ou compartilhamento de código entre os estudantes, principalmente em grandes turmas, a exemplo dos cursos de ensino superior.

Além do uso de questões já disponíveis na base do ACAC para compor uma nova atividade, o instrutor pode optar por elaborar novas questões. Estas não dispõem de informações que possam

Fica permitido ao(s) autor(es) ou a terceiros a reprodução ou distribuição, em parte ou no todo, do material extraído dessa obra, de forma verbatim, adaptada ou remixada, bem como a criação ou produção a partir do conteúdo dessa obra, para fins não comerciais, desde que sejam atribuídos os devidos créditos à criação original, sob os termos da licença CC BY-NC 4.0.

EduComp'21, Abril 27–30, 2021, Jataí, Goiás, Brasil (On-line)

© 2021 Copyright mantido pelo(s) autor(es). Direitos de publicação licenciados à Sociedade Brasileira de Computação (SBC).

indicar o quão fácil será sua resolução por um estudante [10, 29]. Logo, para que o sorteio aleatório de questões seja equânime, faz-se necessário um mecanismo capaz de classificar a facilidade de novas questões antes de serem apresentadas aos estudantes, utilizando para isso somente os dados disponíveis no momento de cadastro [15, 44], ou seja, somente as informações disponibilizadas pelo instrutor.

Análise da facilidade ou dificuldade deve estar associada ao tópico do problema, pois problemas de determinados tópicos são inerentemente mais fáceis/difíceis que outros, inviabilizando uma comparação [32]. Para ilustrar, um problema fácil de programação dinâmica (ex. problema da mochila 0-1) é provavelmente desafiador para a maioria dos alunos nos primeiros períodos de uma graduação em ciência da computação, enquanto que um problema difícil de estruturas sequenciais ou condicionais é provavelmente fácil para um aluno no segundo semestre do curso. Assim, treinar um classificador de facilidade com uma base de dados que possua problemas de diversos tópicos é inviável, devendo existir um classificador de facilidade ou dificuldade para cada tópico de programação. Dessa forma, será possível identificar quais são os problemas fáceis e difíceis do tópico de estruturas condicionais, estruturas de repetição e etc.

Nesse sentido, o objetivo deste trabalho é propor um modelo baseado em técnicas de aprendizagem de máquina que classifique questões de codificação, estratificadas por tópico, segundo a facilidade de resolução das mesmas, de forma a ajudar os instrutores a elaborar atividades (listas de exercícios e exames) mais justos. Essa classificação deve ser feita no momento do cadastro da questão na base do ACAC, antes mesmo de seu uso numa atividade. Isso garante um melhor nivelamento do grau de facilidade do exame para todos os estudantes, mesmo que as questões desse exame tenham sido criadas recentemente e ainda não tenham sido utilizadas em outras ocasiões dentro do ACAC.

Desta forma, o modelo de classificação proposto neste trabalho faz uso de atributos extraídos de *códigos de solução* cadastrados pelo instrutor. O código de solução de uma questão consiste num código desenvolvido pelo próprio instrutor contendo uma possível solução para a questão. Tais códigos são utilizados, por exemplo, para auxiliar o instrutor a criar casos de teste que serão usados pelo ACAC na correção automática dos códigos desenvolvidos pelos estudantes.

O modelo proposto foi avaliado considerando que a facilidade real (variável dependente) de uma dada questão pode ser expressa pela *taxa de acerto* dessa questão. A *taxa de acerto* foi definida como a razão entre o número de estudantes que submeteram códigos corretos para todos os casos de testes e o número de estudantes que tentaram submeter, ao menos uma vez, um possível código de solução.

Por meio deste trabalho almejou-se responder às seguintes questões de pesquisa:

QP1: Como os atributos extraídos dos códigos de solução do instrutor podem ser utilizadas para classificação da facilidade das questões?

QP2: A divisão das questões segundo os conceitos de programação abordados resulta numa classificação melhor?

QP3: O mesmo conjunto de atributos pode ser utilizado para classificar qualquer tipo de questão?

Para apresentar o modelo de classificação proposto, este trabalho encontra-se dividido da seguinte forma. A Seção 2 apresenta algumas das abordagens e trabalhos encontrados na literatura. A Seção 3 caracteriza o conjunto de questões utilizadas e a sequência de etapas adotadas na amostragem. A Seção 4 descreve os métodos e técnicas utilizados, e também como foi realizado o processo de classificação de questões. A Seção 5 apresenta os resultados obtidos. A Seção 6 demonstra qual a contribuição deste trabalho para a Educação em Computação e quais são as aplicações práticas desta pesquisa. A Seção 7, apresenta as premissas adotadas e ameaças à validade. Por fim, a Seção 8 apresenta as considerações finais e trabalhos futuros.

2 TRABALHOS RELACIONADOS

Uma das abordagens para classificação da facilidade de questões encontrada na literatura é a classificação manual. Nela, um grupo de instrutores classifica as questões segundo sua opinião [3, 7, 11, 20]. O principal problema desse método é a subjetividade na avaliação por depender da experiência pessoal do instrutor e de um profundo conhecimento do contexto do problema. Sheard et al. [38] reportam que na classificação de um conjunto 252 questões quanto ao nível de dificuldade, somente 43% dos tutores chegaram a um consenso. Outro problema na classificação manual é a escalabilidade do método. Com o aumento na quantidade de questões, a classificação manual começa a demandar muito tempo tanto para a avaliação quanto para a revisão da classificação.

Mesmo que se tenha uma quantidade ideal de avaliadores humanos e tempo hábil para classificar as questões da base, ainda é possível que haja divergência nas opiniões dos avaliadores. Nessa perspectiva, Sheard et al. [38] propuseram a classificação de questões segundo uma escala de dificuldade com 03 níveis. Como esperado, houve divergências entre os avaliadores e, mesmo após debaterem as classificações conflitantes, houve consenso entre menos da metade dos avaliadores. Ademais, a dificuldade estimada pelo instrutor pode não condizer com aquela experimentada pelo estudante, como foi evidenciado em Meisalo et al. [24].

Alguns autores optam por adaptar modelos já existentes para o contexto de questões de programação. Por exemplo, Zaffalon et al. [43] apresenta um estudo analítico de dois modelos que estimam a habilidade de estudantes: Teoria de Resposta ao Item (TRI) e ELO. A TRI estima a probabilidade de um indivíduo acertar um item em função das características do item e da(s) habilidade(s) estimada(s) do indivíduo. O ELO [8] foi originalmente proposto com o objetivo de estimar o nível de habilidade de jogadores de xadrez por meio dos seus históricos de jogo, entretanto pode ser usado na educação ao interpretarmos a tentativa de um estudante responder a uma questão, como uma “partida” entre o estudante e a questão Pelánek [28]. Apesar dos resultados promissores, ambos os modelos não podem ser aplicados para estimar a facilidade de novas questões.

Alguns trabalhos mais recentes fazem uso de atributos extraídos diretamente do enunciado das questões por meio do processamento de linguagem natural. Seguindo essa abordagem, Santos et al. [37] tentaram correlacionar atributos de legibilidade extraídos do enunciado de questões com a dificuldade numa escala de 04 pontos. Por

meio dos dados extraídos de um juiz online, eles confirmaram a noção intuitiva de que questões com enunciados complexos e difíceis de ler normalmente produzem baixa taxa de acerto, enquanto que o contrário, ou seja, questões com enunciado simples e de fácil leitura nem sempre produzem altas taxas de acerto. Isso se deve ao fato de que alguns enunciados podem trazer informações irrelevantes para a resolução do problema. Como essa abordagem é baseada na contagem de sílabas, palavras e sentenças, tanto os enunciados curtos quanto enunciados verbosos interferem na extração de atributos.

Existe ainda a abordagem baseada na análise de códigos de solução. Oliveira et al. [27] dividem esse tipo de análise em dois grupos: análise dinâmica e análise estática. A análise dinâmica utiliza atributos provenientes da execução de códigos de solução elaborados pelos estudantes como, por exemplo, o tempo de execução e a quantidade de casos de testes corretos. Já a análise estática faz uso de atributos extraídos da estrutura de códigos de solução, por exemplo a quantidade de laços de repetição.

A análise estática não somente apresenta menor custo computacional, por descartar a necessidade de execução dos códigos de solução, mas também não depende de um gabarito (casos de teste). Podendo portanto ser aplicada tanto em códigos de solução de estudantes quanto de instrutores. Nesse sentido, Elnaffar [7] utiliza um pequeno conjunto atributos extraídos manualmente de códigos de solução fornecidos por instrutores para criar um índice para a dificuldade encontrada por estudantes durante a solução de questões de codificação. Apesar do pequeno conjunto de atributos, 05 atributos apenas, seu trabalho encontrou forte correlação ($>0,9$) entre os atributos extraídos de código e a dificuldade estimada de questões. Porém, a extração de atributos manualmente não só é sujeita a falhas humanas como também só é viável quando o conjunto de questões é pequeno.

Seguindo a mesma abordagem, Lima et al. [19] apresentaram um método de classificação de questões de codificação segundo sua dificuldade. Todas as questões foram aplicadas em *exames presenciais* de turmas de introdução à computação. Foram extraídas automaticamente 91 atributos de códigos de solução de instrutores. Esse trabalho obteve um *f1-score* de 81% utilizando uma classificação binária (classes “Difícil” e “Não Difícil”). Apesar dos bons resultados, o modelo de classificação apresentado fez uso de uma grande quantidade de classificadores (11 classificadores-base e 01 meta-classificador). Isso dificulta a manutenibilidade do sistema por conta da grande quantidade de hiperparâmetros.

Outra abordagem possível seria combinar análise estática e dinâmica. Nesse sentido, Neves et al. [25] apresentaram uma ferramenta que mapeia soluções de programação, escritas por estudantes em linguagem C, em perfis de aprendizagem, estes representados por 348 métricas de software (atributos). Embora utilize um amplo conjunto de atributos, essa abordagem exige que as questões tenham sido apresentadas a um número razoável de estudantes. Logo, novas questões só poderiam ter sua facilidade/dificuldade estimada após serem apresentadas aos estudantes. Sendo assim, a análise dinâmica está fora do escopo deste trabalho, pois o objetivo aqui é classificar novas questões.

Diferentemente dos estudos apresentados, que tentam classificar todo o conjunto de questões numa única etapa, neste estudo as questões são divididas. Nossa hipótese é que a divisão do conjunto de questões segundo o tópico pode melhorar a classificação além de

reduzir a quantidade de atributos utilizados. Isso possibilitará uma simplificação no processo de seleção de problemas para composição de atividades (listas de exercícios e provas) com níveis de facilidade/dificuldade balanceados, já que será possível identificar quais são os problemas fáceis e difíceis de cada tópico. Além disso, com o uso apenas de códigos de solução dos instrutores, não será necessário que um grupo razoável de alunos desenvolva soluções para as questões cadastradas, tornando assim o sistema de classificação mais escalável.

3 CARACTERIZANDO A AMOSTRA DE DADOS

3.1 Base de dados

A base de questões¹ utilizada neste trabalho foi obtida do ambiente de correção automática de código CodeBench², utilizado na Universidade Federal do Amazonas. Dentre as diversas disciplinas que utilizam tal ferramenta, selecionamos somente questões resolvidas pelos estudantes da disciplina de Introdução à Programação de Computadores (IPC), entre os anos de 2017 e 2019.

A disciplina de IPC é ministrada para 17 cursos de graduação, nas áreas de engenharia e ciências exatas. Tem carga horária de 60h e sua ementa é dividida em 7 módulos, cobrindo conceitos básicos de programação, conforme listagem a seguir:

- Módulo 01: Variáveis e estrutura sequencial
- Módulo 02: Estrutura condicional simples
- Módulo 03: Estrutura condicional encadeada
- Módulo 04: Estrutura de repetição por condição
- Módulo 05: Vetores e strings
- Módulo 06: Estrutura de repetição por contagem
- Módulo 07: Matrizes

Embora sejam realizadas *listas de exercícios* e *exames presenciais* em cada módulo, para o presente estudo foram consideradas somente as questões aplicadas em *exames presenciais*, pois as *listas de exercícios* podem ser resolvidas em qualquer ambiente, com consulta e com duas semanas de prazo de entrega. Já os *exames presenciais*, são realizados em um ambiente controlado, sob a supervisão de um instrutor e um tutor, inibindo a possibilidade de compartilhamento de código entre os estudantes ou consultas a fontes externas. Além disso, o tempo de resolução dos exames é limitado, exigindo assim foco dos estudantes na tarefa de elaborar os códigos de resposta.

3.2 Processo de amostragem

Inicialmente, foram obtidas do banco de questões do ACAC um total de 653 questões aplicadas em *exames presenciais*. No entanto, o processo de avaliação do método proposto neste trabalho envolve uma análise das tentativas de solução das questões por parte dos alunos, para o cálculo da variável dependente, e por isso é importante que as questões consideradas nos experimentos tenham sido utilizadas por um conjunto razoável de estudantes.

Para obter uma estimativa estável da facilidade/dificuldade de questões de programação são necessárias algumas centenas de tentativas [6]. Portanto, foi aplicado um filtro nas 653 questões, selecionando apenas aquelas com 20 ou mais tentativas de solução.

¹<http://codebench.icomp.ufam.edu.br/dataset/>

²<http://codebench.icomp.ufam.edu.br/>

Esse valor foi escolhido almejando diminuir a distorção da facilidade e ainda assim manter uma boa quantidade de observações na amostra, haja vista que valores maiores reduziriam a quantidade de questões para menos de 50% da base original. Com a aplicação do filtro, foram selecionadas então 354 questões das 653 iniciais.

3.3 Complementando a amostra

Posto que o objetivo desse trabalho é classificar novas questões de programação no momento em que elas são cadastradas no ACAC, não seria possível utilizar atributos extraídos de códigos de solução dos estudantes, por não estarem disponíveis no momento do cadastro da questão. Ocorre que o próprio ACAC possibilita o cadastro de um modelo de solução (código) para cada questão em sua base. O modelo de solução, quando inserido pelo instrutor no momento da elaboração de uma nova questão, precisa passar em todos os casos de teste da questão.

Como o código de solução é um campo opcional do formulário de cadastro de novas questões, nem todas as questões possuem um modelo de solução cadastrado. Das 354 questões selecionadas, 78 não contavam com um código de solução elaborado pelo instrutor. Objetivando não reduzir mais ainda a amostra, e considerando que cada uma dessas 78 questões foram solucionadas por pelo menos 20 alunos, optou-se por usar códigos de alunos que acertaram as questões como alternativas aos códigos de solução dos instrutores. Desta forma, para cada uma das 78 questões sem códigos de solução, foi selecionado o código de solução de um dado aluno como modelo de solução da questão. A escolha de qual solução utilizar foi feita utilizando 03 critérios de desempate, em ordem de prioridade:

- (1) Menor número de submissões para correção automática feitas pelo estudante;
- (2) Menor número de execuções do código feitas pelo estudante;
- (3) Menor quantidade de erros obtidos durante a tentativa de solucionar a questão.

Como a ementa da disciplina já apresenta uma divisão em módulos, abordando os principais tópicos introdutórios de programação, as questões foram distribuídas seguindo o mesmo critério. A Figura 1 apresenta a distribuição das questões por módulo. O Módulo 01 (M01) por ser um módulo introdutório, contém a maior quantidade de questões, 18,6% delas. Em contrapartida o Módulo 06 (M06), que aborda estruturas de repetição por contagem, apresenta o menor conjunto de questões, 9,3% das questões. Por fim, mais da metade das questões selecionadas está concentrada nos três primeiros módulos da disciplina (M01, M02 e M03).

4 METODOLOGIA

4.1 Variável dependente (alvo)

Apesar da *facilidade* encontrada em questões introdutórias de programação não possuir uma definição formal, encontramos na literatura algumas abordagens semelhantes para mensurá-la: o número médio de submissões, o tempo médio de solução e a taxa de sucesso (acerto) [6, 7, 18]. A viabilidade de cada uma dessas abordagens foi avaliada dentro do contexto deste trabalho e dos dados disponíveis.

A contagem do *número de submissões* feitas por cada estudante foi realizada por um trabalho de mineração de dados nos arquivos

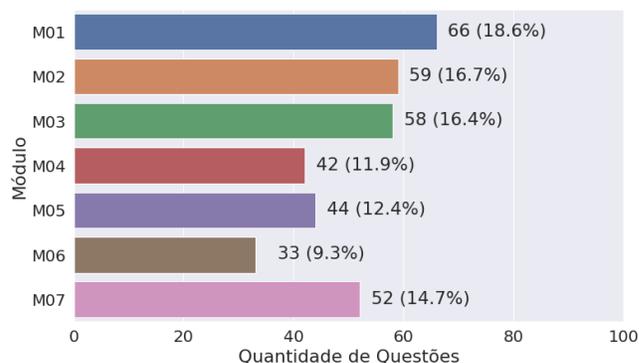


Figura 1: Distribuição de questões por módulo

de registro (*logs*) da base de dados. Esses arquivos são gerados enquanto os estudantes tentam codificar soluções para as questões. Nesses arquivos ficam registradas as submissões para correção automática, as execuções de código feitas pelo estudante e também os erros apontados pelo interpretador Python durante submissões ou execuções. Ao todo foram analisados 17.374 registros de tentativas, correspondentes às 354 questões da amostra.

Durante a análise exploratória do número de submissões, foram identificados muitos *outliers* na amostra. A Figura 2 apresenta a variação do número de submissões num diagrama de caixa. Para melhorar a escala do gráfico e facilitar a visualização aplicou-se a função raiz quadrada nos dados. A análise estatística descritiva do número de submissões revelou que essa distribuição é assimétrica positiva com $\bar{X} = 4,32(\pm 0,074)$, $\bar{X} = 2,00$, $s = 9,693$. Além disso apresenta como quartis: $q_1 = 1,00$, $q_2 = 2,00$ e $q_3 = 4,00$.

Uma análise mais minuciosa revelou que alguns estudantes, mesmo após já terem solucionado uma dada questão, continuavam a submeter códigos para correção automática, muitos deles não funcionais. Isso inviabilizou, pelo menos para o nosso contexto, o uso do *número de submissões* como variável dependente para estimar a facilidade, haja vista que o comportamento arbitrário dos estudantes distorce de forma significativa a amostra e a remoção dos *outliers* levaria a uma redução ainda maior da amostra, já anteriormente distribuída em 07 partições.

Adicionalmente, os *logs* também armazenam todas as edições de código feitas pelos estudantes durante o desenvolvimento de suas soluções. Para cada modificação feita no código-fonte é gerada uma nova entrada no arquivo de *log*, contendo a data, hora, linha, coluna e a edição feita. Além disso, também são registrados alguns eventos relacionados com a área de edição de código: ganho de foco, perda de foco, rolagem (*scroll*), etc. O *tempo de solução* foi calculado somando as diferenças entre os intervalos de tempo dos eventos registrados nesses arquivos de *log*. Além disso, só foram considerados os eventos registrados que ocorreram dentro do intervalo de duração do *exame presencial*, ou seja, modificações posteriores ao término da atividade foram desconsideradas. Foi observado que, durante a resolução de um problema, existiam longos intervalos de tempo em que o estudante não interagiu com o ambiente, mesmo que a área de edição estivesse com o foco. Esses intervalos poderiam representar um momento de dúvida do estudante, um período de distração e até mesmo abandono da questão. Consequentemente, mensurar de

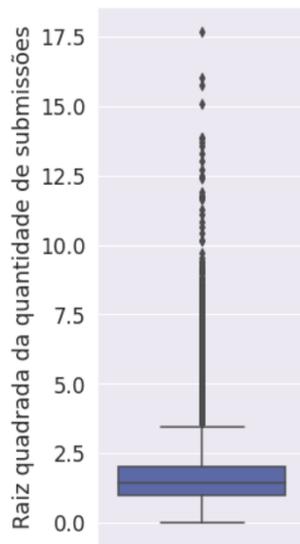


Figura 2: Diagrama de caixa da quantidade de submissões

forma precisa o real *tempo de solução* seria algo muito complexo e incerto.

Diante disso, adotou-se como *variável-dependente* somente a *taxa de acerto*, aqui definida como a razão entre o número de alunos que conseguiram solucionar uma questão e o total de alunos que tentaram resolvê-la. Foram desconsideradas as tentativas em que o código de solução do estudante não era funcional, ou seja, o código não era executável. Foge do escopo deste trabalho tentar distinguir se o estudante realmente tentou solucionar a questão ou se simplesmente desistiu do problema, deixando um código de solução incompleto.

4.2 Variáveis independentes (atributos)

Como *variáveis independentes* foram utilizados um conjunto de atributos extraídos dos códigos de solução elaborados para cada questão. Para o processo automático de extração de atributos foi construído um Extrator³ em Python que analisa toda a estrutura de diretórios da base de dados. Para a extração de atributos dos códigos de solução, o extrator utiliza dois módulos: *Radon*⁴ e *Tokenize*⁵. Ambos são aplicados separadamente em cada um dos códigos de solução dos instrutores.

O módulo *Radon* foi utilizado para extrair os atributos relacionados com métricas de software [13], algumas delas baseadas no tamanho do código (quantidade total de linhas, linhas lógicas, linhas em branco, comentários, etc.) e também na *complexidade ciclomática total do código* [22].

O módulo *Tokenize*, por sua vez, faz parte da biblioteca padrão da linguagem Python, e tem como função a geração de *tokens* a partir da *análise léxica* de códigos em Python. Os *tokens* gerados foram utilizados para criar novos atributos, aqui chamados de “atributos

³<https://github.com/marcosmapl/codebench-extractor>

⁴<https://pypi.org/project/radon/>

⁵<https://docs.python.org/3/library/tokenize.html>

de complexidade estrutural”. Alguns exemplos de “atributos de complexidade estrutural” são as quantidades encontradas de: *keywords*, *imports*, *laços while*, *elif*, operadores aritméticos, identificadores, constantes e funções embutidas (*built-in*). Tanto as estruturas condicionais quanto as de repetição foram contabilizadas de duas formas: individualmente (*if*, *else*, *while*, *for*, etc.) e na totalidade de ocorrências. Os operadores, por sua vez, foram contabilizados de forma isolada e também agrupados em 04 categorias:

- Aritméticos
- Lógicos
- Relacionais
- Bit-a-bit

Além disso, foram derivados alguns atributos relativos a outros componentes da linguagem, como a quantidade de *parênteses* e *dois-pontos*. Por fim, a *quantidade de funções embutidas* na linguagem também foram contabilizadas como um único atributo, exceto pelas funções *input* e *print*, que por estarem relacionadas com os mecanismos de verificação e correção das soluções foram contabilizadas individualmente.

Como última etapa, foi realizado um processo de *engenharia de atributos* utilizando os atributos obtidos até o momento. O objetivo desse processo é criar mais informação que possa ser pertinente aos modelos na etapa de classificação. Por meio dos atributos discretos quantitativos foram criados novos atributos booleanos (verdadeiro ou falso) indicando a ocorrência ou não de algumas estruturas no código, como estruturas condicionais e repetição. Esse tipo de informação pode ser útil em modelos baseados em árvores de decisão, de modo que o valor booleano possa ser utilizado como critério de divisão num nó. Do mesmo modo, os atributos baseados em tamanho do código foram combinados para gerar novos atributos como, por exemplo, a média de identificadores por linha de código e a média de caracteres por identificador (nomes definidos pelo estudante).

Ao final de todas essas etapas foi obtido um conjunto de 91 atributos a serem utilizados como *variáveis independentes*.

4.3 Remoção de atributos com baixa variância

A análise estatística dos atributos extraídos revelou que alguns possuíam baixa variância e média próximas de zero. Isso se explica pelo fato da disciplina ser dividida em sete módulos organizados de modo a facilitar a evolução gradual do estudante. É de se esperar que nos códigos elaborados nos primeiros módulos não apareçam construções mais avançadas (operadores bit-a-bit, expressões lambdas, módulos externos, etc.). Tal fato, além de dificultar a generalização, ainda poderia atrapalhar a seleção de atributos para os modelos de classificação. Em função disso, em cada conjunto de dados foram removidos os atributos com variância inferior a 10^{-5} . A quantidade de atributos restantes para cada conjunto encontra-se descrita na Tabela 2.

4.4 Discretização da taxa de acerto

O domínio de valores da *taxa de acerto* é contínuo, compreendendo valores no intervalo de 0 a 1. Para que essa variável possa ser utilizada para classificação é necessário discretizá-la. Como referência para a discretização foi utilizado o “índice de facilidade”, adotado pelo Instituto Nacional de Estudos e Pesquisas Educacionais Anísio

Teixeira (INEP) no Enade [14]. O índice é dividido em cinco níveis de facilidade de acordo com a “taxa de acerto”. Neste trabalho, esse índice foi simplificado para apenas dois níveis, pois a adoção de mais níveis reduziria a quantidade de amostras das classes minoritárias em partições estratificadas num treino por validação cruzada. Além disso, o uso de dois níveis nos permitiu comparar nossa abordagem com o trabalho realizado por Lima et al. [19]. O mapeamento realizado está disposto na Tabela 1.

Tabela 1: Discretização da taxa de acerto.

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

Conforme novos conceitos são ensinados aos estudantes, a *complexidade* das questões aumenta, exigindo o uso de estruturas mais complexas nos códigos de solução. Isso pode ser observado na Tabela 2 onde a quantidade de atributos extraídos dos códigos de solução aumenta com o avanço nos módulos da disciplina. Além disso, mais da metade das questões está concentrada nos três módulos iniciais da disciplina (M01, M02 e M03). Observando a distribuição de questões em cada classe dentro dos módulos, a tabela também mostra que quase todos os módulos são desbalanceados, exceto pelo *Módulo 05* (M05). Por fim, o *Módulo 04* (M04) é o único com mais questões da classe “Não Fácil”, em todos os demais a classe “Fácil” é majoritária.

Tabela 2: Distribuição de Questões para Classificação Binária.

| Módulo | Atributos | Questões | Classes | |
|--------|-----------|----------|---------|-----------|
| | | | Fácil | Não Fácil |
| M01 | 56 | 66 | 84,85% | 15,15% |
| M02 | 65 | 59 | 86,44% | 13,53% |
| M03 | 72 | 58 | 72,41% | 27,59% |
| M04 | 73 | 42 | 23,81% | 76,19% |
| M05 | 74 | 44 | 56,82% | 43,18% |
| M06 | 83 | 33 | 66,67% | 33,33% |
| M07 | 81 | 52 | 69,23% | 30,77% |

4.5 Escolhendo uma métrica de desempenho para avaliação dos modelos

Antes de iniciar o processo de classificação, é necessário escolher uma métrica para avaliar e comparar o desempenho dos modelos. Embora existam diversas métricas para avaliar modelos de classificação, foram analisadas 05 métricas e duas delas foram tomadas como principais.

A *acurácia*, apesar de ter sido computada nos resultados, não foi utilizada como parâmetro de desempenho dos modelos justamente

por termos um conjunto de dados desbalanceados. Utilizar a *acurácia* em bases desbalanceadas pode levar a resultados enganosos, haja vista que nesses cenários, até mesmo modelos enviesados que chutem todas as amostras como sendo da classes majoritária, terão uma *acurácia* elevada.

Quanto a *precisão*, por ser uma métrica que avalia o quão assertivo é um modelo, serve como indicador do quão bom um modelo é em identificar as observações para classe positiva. Em geral, seu uso é recomendado quando *Falsos Positivos* são mais prejudiciais que *Falsos Negativos*. Entretanto, na classificação da facilidade de questões de programação é importante identificar corretamente as observações para ambas as classes. Classificar erroneamente uma questão “Fácil” como sendo “Não Fácil” pode beneficiar alguns estudantes com um sorteio de questões mais fáceis do que os demais, do mesmo modo classificar questões “Não Fáceis” como sendo “Fáceis” prejudica os estudantes por gerar atividades mais difíceis do que o desejado. Por isso a *precisão* também foi descartada como métrica principal.

Outra métrica avaliada foi a *revocação*. Por ser uma métrica que avalia a completude dos modelos, ela serve como indicador da frequência com que o modelo identifica exemplos de uma das classes. Geralmente, a *revocação* é utilizada em situações em que *Falsos Negativos* são mais prejudiciais que *Falsos Positivos*. Não sendo ideal como métrica principal pelo mesmo que motivo da *precisão*.

Por fim, temos o *f1-score* que consiste na média harmônica entre *precisão* e *revocação*. Essa combinação permite avaliar tanto a *precisão* quanto a *revocação* do modelo num único valor. Por esse motivo o *f1-score* foi escolhido como métrica principal para avaliar o desempenho do modelo. Além do *f1-score*, foi utilizado como critério de desempate o *log loss* por ser uma métrica baseada na probabilidade estimada pelos modelos e também punir previsões incorretas muito confiantes.

4.6 Processo de classificação

Para a classificação das questões foi utilizada a *técnica de ensemble Stacking* [42] em duas etapas. Esta técnica permite a combinação de diversos algoritmos de aprendizagem de máquina, aproveitando dos pontos fortes de cada um para uma predição final mais eficiente.

Primeiramente, para cada um dos 07 conjuntos de questões foram treinados 04 “classificadores-base”. Um “classificador-base” é um modelo treinado isoladamente que será combinado num *ensemble*. O processo de treino dos “classificadores-base” foi realizado por meio de “validação cruzada” (*cross-validation*). Devido à pouca quantidade de questões dentro de cada conjunto, foram utilizadas 03 partições, escolhidas de forma estratificadas. As partições dos “classificadores-base” foram escolhidas usando o número 42 como *semente de aleatoriedade*.

Durante o processo de “validação cruzada”, duas partições eram utilizadas para treino do “classificador-base” e uma partição para teste. As predições estimadas (classe à qual a observação pertence) pelos “classificadores-base” para a partição de teste eram salvas com intuito de serem utilizadas na segunda etapa de classificação. O mesmo foi feito para as probabilidades estimadas (distribuição de probabilidade no conjunto de classes) pelos “classificadores-base” probabilísticos.

Concomitante ao treino de cada “classificador-base”, foi feito um processo de “seleção de atributos” objetivando remover atributos que adicionem ruído e melhorar o custo computacional ao reduzir a quantidade de dados. Para a “seleção de atributos” foram utilizadas duas abordagens em que os atributos recebem pontos (scores) para que posteriormente sejam selecionados os “k” melhores atributos para cada “classificador-base”. A primeira abordagem faz uso da função de uma função para pontuar os atributos. Como funções de pontuação foram utilizadas $f_classif$ e $mutual_f_classif$, dentre elas a $f_classif$ obteve melhores resultados. A segunda abordagem faz uso de um outro classificador criado apenas para pontuar os atributos. Nas duas abordagens, o parâmetro “k” foi testado para o intervalo de 1 até n, onde n representa a quantidade de atributos do conjunto de dados que o “classificador-base” fará uso. Por fim, selecionamos a abordagem com o melhor $f1$ -score para cada “classificador-base” treinado.

Como “classificadores-base” foram utilizados os modelos:

- K-Nearest Neighbor (knn);
- Random Forest (rf);
- Support Vector Machine (svm); e
- XGBoost (xgb).

Em seguida, as previsões e probabilidades são utilizadas como atributos para o treino de 02 “meta-classificadores” para cada um dos 07 conjuntos de questões, ou seja, 14 “meta-classificadores” ao todo. O primeiro “meta-classificador” é treinado somente com as previsões estimadas, enquanto que o segundo “meta-classificador” é treinado somente com as probabilidades estimadas. Os “meta-classificadores” foram treinados utilizando o mesmo processo dos “classificadores-base”, exceto que para os “meta-classificadores” o número 43 foi escolhido como *semente de aleatoriedade*. Isso garantiu que os “meta-classificadores” não tivessem as mesmas partições das que foram selecionadas para os “classificadores-base”, não permitindo assim o vazamento de informação entre as etapas.

Para “meta-classificadores” foram utilizados os modelos:

- K-Nearest Neighbor (knn);
- Random Forest (rf); e
- XGBoost (xgb).

Os scripts (jupyter notebooks) e dados utilizados encontram-se organizados num repositório do GitHub no endereço: <https://github.com/marcosmapl/dificuldade-questoes>

5 ANÁLISE DOS RESULTADOS

5.1 Resultado dos Meta-classificadores

Os resultados de cada “meta-classificador” estão dispostos na Tabela 3. Dentre os “meta-classificadores”, o “Meta 02” do Módulo 05 – Vetores e strings, um XGBoost, foi o que obteve melhor resultado dentre os conjuntos de dados isolados. Quando os “meta-classificadores” de um mesmo conjunto de dados obtinham o mesmo desempenho ($f1$ -score) o menor valor de \log loss era utilizado como critério de desempate.

Os resultados dos “classificadores-base” quando comparados com seus respectivos “meta-classificadores” tiveram resultados melhores, exceto para os módulos 03, 04 e 06, onde os “meta-classificadores” obtiveram o mesmo resultado que o melhor “classificador-base”. Logo, o processo de classificação utilizando pode ser simplificado

Tabela 3: Resultados dos meta-classificadores por módulo

| Módulo | Meta-Classificador | F1 | Log Loss | Acurácia |
|--------|--------------------|------|----------|----------|
| M01 | Meta 01 (rf) | 0,91 | 0,18 | 0,95 |
| | Meta 02 (xgb) | 0,86 | 0,26 | 0,94 |
| M02 | Meta 01 (rf) | 0,73 | 1,36 | 0,88 |
| | Meta 02 (xgb) | 0,81 | 0,26 | 0,93 |
| M03 | Meta 01 (xgb) | 0,87 | 0,70 | 0,90 |
| | Meta 02 (xgb) | 0,87 | 0,51 | 0,90 |
| M04 | Meta 01 (xgb) | 0,84 | 0,37 | 0,88 |
| | Meta 02 (xgb) | 0,84 | 0,43 | 0,88 |
| M05 | Meta 01 (xgb) | 0,98 | 0,22 | 0,98 |
| | Meta 02 (xgb) | 0,95 | 0,20 | 0,98 |
| M06 | Meta 01 (knn) | 0,84 | 1,31 | 0,85 |
| | Meta 02 (xgb) | 0,84 | 0,66 | 0,85 |
| M07 | Meta 01 (xgb) | 0,91 | 0,68 | 0,92 |
| | Meta 02 (xgb) | 0,83 | 0,68 | 0,85 |

utilizando somente um “classificador-base” para cada um dos módulos em que não houve melhora no desempenho com o uso de *ensemble*.

O resultado final do processo de classificação foi obtido pela junção das previsões dos melhores “meta-classificadores” de cada módulo. Analisando a matriz de confusão (Figura 3) temos, no geral, uma *acurácia* de 0,92 e *f1-score* de 0,94. A *precisão* por sua vez é alta tanto para classe “Fácil” quanto para “Não Fácil”, 0,93 e 0,88 respectivamente. Semelhantemente, a *revocação* também foi alta, 0,86 para a classe “Não Fácil” e 0,95 para a classe “Fácil”, mesmo com a relação de compromisso (*trade-off*) existente entre *precisão* e *revocação*. Ademais, o modelo obteve um boa completude para ambas as classes.

Nossa abordagem demonstrou a viabilidade da classificação da facilidade de questões de codificação, pelo uso de atributos extraídos de códigos de solução elaborados por instrutores (QP1), sendo capaz de superar os resultados de Lima et al. [19]. Logo, a divisão do conjunto de questões segundo os tópicos abordados (conceitos de programação), além de melhorar o desempenho dos classificadores (QP2) também reduziu a quantidade de atributos utilizados pelos modelos, em todos os conjuntos de questões.

5.2 Análise do conjunto de atributos

Durante a seleção de atributos constatou-se que o conjunto de atributos selecionado variava não somente de modelo para modelo, como também de acordo com o conjunto de questões. Esse fato em conjunto com a distribuição de atributos por módulos da Tabela 2 evidencia que o mesmo conjunto de atributos não deve ser utilizado na classificação de qualquer tipo de questão (QP3).

6 CONTRIBUIÇÃO E APLICAÇÕES DO CLASSIFICADOR

A principal contribuição deste trabalho para a Educação em Computação foi apresentar um modelo automático para classificação de questões de codificação, quanto à sua facilidade, pelo uso de atributos extraídos de códigos de solução de instrutores. Além de

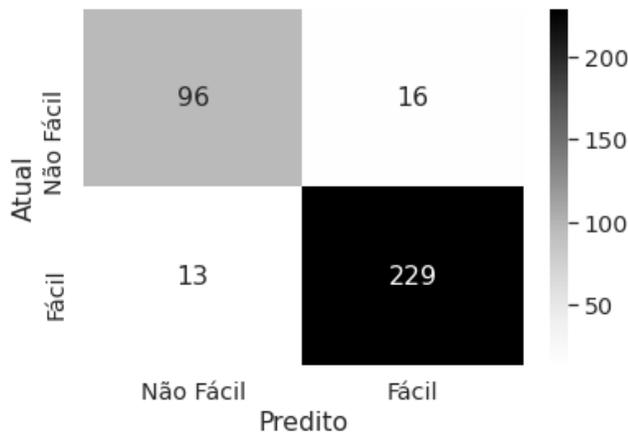


Figura 3: Resultado final da classificação de questões

auxiliar o trabalho do instrutor durante a elaboração e balanceamento de listas de exercícios e exames, o modelo apresentado possibilita também uma aprendizagem não-assistida para o estudante. Após a classificação de toda a base, o próprio aluno poderá selecionar quais questões deseja resolver, de acordo com a facilidade e o tópico desejado.

Outra aplicação vislumbrada seria o uso do modelo de classificação em sistemas de recomendação automática de questões. Inicialmente o ambiente sugere somente questões fáceis, e conforme o estudante for obtendo sucesso em solucioná-las, poderá receber como sugestão questões mais desafiadoras. Logo, o presente trabalho é um passo em direção ao uso de modelos híbridos IA-humanos na educação, onde alunos e professores são assistidos por um modelo preditivo.

Além disso, o modelo de classificação pode ser aplicado como ferramenta para o instrutor na gestão da própria base de questões. Uma vez classificadas as questões da base, o instrutor irá dispor de um “feedback” quanto a dificuldade esperada para cada questão. De posse dessa informação ele poderá refinar as questões conforme a necessidade da turma.

7 PREMISSAS E AMEAÇAS À VALIDADE

Assim como Elnaffar [7], os atributos extraídos são derivados da solução de amostra fornecida pelo instrutor. Embora seja uma abordagem válida e viável, algumas das soluções fornecidas pelos instrutores podem divergir das soluções elaboradas pelos estudantes. Em outras palavras, o código do instrutor pode ser mais conciso e enxuto em questões de disciplinas mais avançadas, não sendo portanto o mais adequado para todos os contextos.

Embora o uso de somente uma métrica com variável-alvo possa ser problemático, por não expressar todos os aspectos da facilidade encontrada pelo estudante, a métrica escolhida (taxa de acerto) foi a alternativa mais viável e que trouxe bons resultados.

Apesar do filtro utilizado não ser tão alto, utilizamos um conjunto com número elevado de questões quando comparado com os trabalhos encontrados. Outro fator importante é que a variável-alvo (taxa de acerto) foi calculada de um total de 17.374 submissões.

8 CONCLUSÃO E TRABALHOS FUTUROS

O presente trabalho utilizou um total de 354 questões introdutórias de programação, divididas em 07 conjuntos segundo os módulos da disciplina ministrada, para serem classificadas automaticamente segundo sua facilidade, em duas classes: “Fácil” e “Não Fácil”. Para a classificação, foram utilizados como atributos, métricas extraídas diretamente de códigos de solução elaborados por instrutores. Posteriormente, os atributos foram utilizados no processo de treino de 28 classificadores-base (04 para cada módulo). Em seguida, as predições e probabilidades estimadas pelos classificadores-base serviram como dados de treino para 14 meta-classificadores (02 para cada módulo), sendo que para cada módulo um dos meta-classificadores treinou somente com predições e o outro, com probabilidades estimadas. Como resultado final, combinamos as predições dos melhores meta-classificadores de cada módulo, obtendo assim 0,94 de *f1-score* e 0,92 de *acurácia*.

Como continuidade de nossa pesquisa, pretende-se:

- Investigar a possibilidade de conjugar mais métricas extraídas do código de solução com métricas de inteligibilidade textual provenientes do enunciado das questões, obtendo assim uma cobertura maior da facilidade.
- Expandir as classes utilizadas na classificação de modo que tenhamos uma melhor separação dos níveis de facilidade, permitindo assim a rotulação das questões por grau de facilidade.
- Avaliar a possibilidade de expressar a facilidade através de valores contínuos, permitindo assim a implementação de um mecanismo de ordenação (ranking) de questões por meio de modelos de regressão.
- Avaliar o desempenho de classificadores utilizando o conjunto de dados como um todo, sem a divisão por módulos. Nossa hipótese é que, embora as questões estejam divididas em módulos segundo algum conceito de programação, é plausível acreditar que existam questões que envolvam mais de um conceito.

AGRADECIMENTOS

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

REFERÊNCIAS

- [1] Brett A. Becker and Keith Quille. 2019. 50 Years of CS1 at SIGCSE: A Review of the Evolution of Introductory Programming Education Research. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) (SIGCSE '19). Association for Computing Machinery, New York, NY, USA, 338–344. <https://doi.org/10.1145/3287324.3287432>
- [2] Jean Luca Bez, Carlos E. Ferreira, and Neilor Tonin. 2013/08. URI Online Judge Academic: A Tool for Professors. In *Proceedings of the 2013 International Conference on Advanced ICT and Education*. Atlantis Press, Hainan, China, 744–747. <https://doi.org/10.2991/icaicte.2013.153>
- [3] Paul Denny, Diana Cukierman, and Jonathan Bhaskar. 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*. Association for Computing Machinery, New York, NY, USA, 13–22.
- [4] Leandro Galvão e David Fernandes e Bruno Gadelha. 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 (2016), 140. <https://doi.org/10.5753/cbie.sbie.2016.140>
- [5] Samuel Fonseca e Elaine Oliveira e Filipe Pereira e David Fernandes e Leandro Carvalho. 2019. Adaptação de um método preditivo para inferir o desempenho de alunos de programação. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação - SBIE)* 30, 1 (2019), 1651. <https://doi.org/10.5753/cbie.sbie.2019.1651>
- [6] Tomáš Effenberger, Jaroslav Čechák, and Radek Pelánek. 2019. Measuring Difficulty of Introductory Programming Tasks. In *Proceedings of the Sixth (2019) ACM Conference on Learning @ Scale (Chicago, IL, USA) (L@S '19)*. Association for Computing Machinery, New York, NY, USA, Article 28, 4 pages. <https://doi.org/10.1145/3330430.3333641>
- [7] Said Elnaffar. 2016. Using software metrics to predict the difficulty of code writing questions. In *2016 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, Abu Dhabi, UAE, 513–518.
- [8] A.E. Elo. 1978. *The Rating of Chessplayers, Past and Present*. Arco Pub. <https://books.google.com.br/books?id=8pMnAQAAAMAAJ>
- [9] Daniel Filho, Elaine Oliveira, Leandro Carvalho, Marcela Pessoa, Filipe Pereira, and David Oliveira. 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 (Online)*. SBC, Porto Alegre, RS, Brasil, 491–500. <https://doi.org/10.5753/cbie.sbie.2020.491>
- [10] Samuel Fonseca, Filipe Pereira, Elaine Teixeira de Oliveira, David Oliveira, Leandro Carvalho, and Alexandra Cristea. 2020. Automatic Subject-based Contextualisation of Programming Assignment Lists. EDM, EDM.
- [11] Rodrigo Elias Francisco and Ana Paula Ambrosio. 2015. Mining an Online Judge System to Support Introductory Computer Programming Teaching. In *8th International Conference on Education Data Mining (EDM2015)*. EDM, Madrid, Spain.
- [12] Rodrigo Elias Francisco, Ana Paula Laboissière Ambrósio, Cleon Xavier Pereira Junior, and Márcia Aparecida Fernandes. 2018. Juiz online no ensino de CS1-lições aprendidas e proposta de uma ferramenta. *Revista Brasileira de Informática na Educação* 26, 03 (2018), 163.
- [13] Maurice H. Halstead. 1977. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA.
- [14] INEP. 2017. Relatório Síntese de Área – Ciência da Computação. Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira.
- [15] Chowdhury Md Intisar and Yutaka Watanobe. 2018. Cluster Analysis to Estimate the Difficulty of Programming Problems. In *Proceedings of the 3rd International Conference on Applications in Information Technology (Aizu-Wakamatsu, Japan) (ICAIT'2018)*. Association for Computing Machinery, New York, NY, USA, 23–28. <https://doi.org/10.1145/3274856.3274862>
- [16] Mike Joy and Michael Luck. 1999. Plagiarism in programming assignments. *IEEE Transactions on education* 42, 2 (1999), 129–133.
- [17] Hermínio Júnior, Filipe Pereira, Elaine Oliveira, David Oliveira, and Leandro Carvalho. 2020. Recomendação Automática de Problemas em Juizes Online Usando Processamento de Linguagem Natural e Análise Dirigida aos Dados. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação (Online)*. SBC, Porto Alegre, RS, Brasil, 1152–1161. <https://doi.org/10.5753/cbie.sbie.2020.1152>
- [18] Nadia Kasto and Jacqueline Whalley. 2013. Measuring the Difficulty of Code Comprehension Tasks Using Software Metrics. In *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136 (Adelaide, Australia) (ACE '13)*. Australian Computer Society, Inc., AUS, 59–65.
- [19] Marcos Lima, Leandro Carvalho, Elaine Oliveira, David Oliveira, and Filipe Pereira. 2020. Classificação de dificuldade de questões de programação com base em métricas de código. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação (Online)*. SBC, Porto Alegre, RS, Brasil, 1323–1332. <https://doi.org/10.5753/cbie.sbie.2020.1323>
- [20] Luis Llana, Enrique Martín-Martín, and Cristóbal Pareja-Flores. 2012. FLOP, a Free Laboratory of Programming. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research (Koli, Finland) (Koli Calling '12)*. Association for Computing Machinery, New York, NY, USA, 93–99. <https://doi.org/10.1145/2401796.2401807>
- [21] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Gianakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review (*ITICSE 2018 Companion*). Association for Computing Machinery, New York, NY, USA, 55–106. <https://doi.org/10.1145/3293881.3295779>
- [22] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320.
- [23] Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. 2018. A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education* 62, 2 (2018), 77–90.
- [24] Veijo Meisalo, E Sutinen, and S Torvinen. 2004. Classification of exercises in a virtual programming course. In *34th Annual Frontiers in Education (FIE 2004)*, Vol. 3. IEEE, Savannah, GA, USA, S3D–1. <https://doi.org/10.1109/FIE.2004.1408764>
- [25] Adler Neves, Marcia Oliveira, Helen França, Mônica Lopes, Leonardo Reblin, and Elias Oliveira. 2017. PCódigo 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 (2017), 339. <https://doi.org/10.5753/cbie.wcbie.2017.339>
- [26] Joseph Oliveira, Felipe Salem, Elaine Oliveira, David Oliveira, Leandro Carvalho, and Filipe Pereira. 2020. Os estudantes leem as mensagens de feedback estendido exibidas em juizes online?. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação (Online)*. SBC, Porto Alegre, RS, Brasil, 1723–1732. <https://doi.org/10.5753/cbie.sbie.2020.1723>
- [27] Márcia Oliveira, Adler Neves, Mônica Lopes, and Andreangelo Patuzzo. 2018. Análise de Aprendizagem a partir de Códigos-Fontes e uma Proposta de Seleção Automática de Métricas de Avaliação. *Anais dos Workshops do Congresso Brasileiro de Informática na Educação* 7, 1 (2018), 369. <https://doi.org/10.5753/cbie.wcbie.2018.369>
- [28] Radek Pelánek. 2016. Applications of the Elo rating system in adaptive educational systems. *Computers & Education* 98 (2016), 169 – 179. <https://doi.org/10.1016/j.compedu.2016.03.017>
- [29] Filipe Pereira, Linnik Souza, Elaine Oliveira, David Oliveira, and Leandro Carvalho. 2020. Predição de desempenho em ambientes computacionais para turmas de programação: um Mapeamento Sistemático da Literatura. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação (Online)*. SBC, Porto Alegre, RS, Brasil, 1673–1682. <https://doi.org/10.5753/cbie.sbie.2020.1673>
- [30] Filipe Pereira, Elaine Teixeira de Oliveira, David Oliveira, Alexandra Cristea, Leandro Carvalho, Samuel Fonseca, Armando Toda, and Seiji Isotani. 2020. Using learning analytics in the Amazonas: understanding students' behaviour in introductory programming. *British Journal of Educational Technology* 51 (05 2020), 955–972. <https://doi.org/10.1111/bjet.12953>
- [31] Filipe D. Pereira, Elaine Oliveira, Alexandra Cristea, David Fernandes, Luciano Silva, Gene Aguiar, Ahmed Alamri, and Mohammad Alshehri. 2019. Early Dropout Prediction for Programming Courses Supported by Online Judges. In *Artificial Intelligence in Education*, Seiji Isotani, Eva Millán, Amy Ogan, Peter Hastings, Bruce McLaren, and Rose Luckin (Eds.). Springer International Publishing, Cham, 67–72.
- [32] Filipe Dwan Pereira, Francisco Pires, Samuel Fonseca, Elaine Oliveira, Leandro Carvalho, David Oliveira, and Alexandra Cristea. 2021. Towards a Human-AI hybrid system for categorising programming problems (*SIGCSE '21*). Association for Computing Machinery, New York, NY, USA, 7. <https://doi.org/10.1145/3408877.3432422>
- [33] Filipe D. Pereira, Armando Toda, Elaine H. T. Oliveira, Alexandra I. Cristea, Seiji Isotani, Dion Laranjeira, Adriano Almeida, and Jonas Mendonça. 2020. Can We Use Gamification to Predict Students' Performance? A Case Study Supported by an Online Judge. In *Intelligent Tutoring Systems*, Vivekanandan Kumar and Christos Troussas (Eds.). Springer International Publishing, Cham, 259–269.
- [34] Anthony V. Robins. 2019. *Novice Programmers and Introductory Programming*. Cambridge University Press, 327–376. <https://doi.org/10.1017/9781108654555.013>
- [35] Francisco Rosales, Antonio García, Santiago Rodríguez, José L Pedraza, Rafael Méndez, and Manuel M Nieto. 2008. Detection of plagiarism in programming assignments. *IEEE Transactions on Education* 51, 2 (2008), 174–183.
- [36] Ingrid Santos, David Oliveira, Leandro Carvalho, Filipe Pereira, and Elaine Oliveira. 2020. Tempos de Transição em Estados de Corretude e Erro como Indicadores de Desempenho em Juizes Online. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação (Online)*. SBC, Porto Alegre, RS, Brasil, 1283–1292. <https://doi.org/10.5753/cbie.sbie.2020.1283>
- [37] Pedro Santos, Leandro Silva Galvão Carvalho, Elaine Oliveira, and David Fernandes. 2019. Classificação de dificuldade de questões de programação com base na inteligibilidade do enunciado. In *Simpósio Brasileiro de Informática na Educação (SBIE)*, Vol. 30. SBC, Porto Alegre, RS, Brasil, 1886–1895.
- [38] Judy Sheard, Simon, Angela Carbone, Donald Chinn, Mikko-Jussi Laakso, Tony Clear, Michael Raadt, Daryl D'Souza, James Harland, Raymond Lister, Anne Philpott, and Geoff Warburton. 2011. Exploring Programming Assessment Instruments: A Classification Scheme for Examination Questions. In *7th Intern. Workshop on Computing Education Research (ICER)*. Association for Computing Machinery, Providence, USA, 33–38.
- [39] Narjes Tahaei and David C. Noelle. 2018. Automated Plagiarism Detection for Computer Programming Exercises Based on Patterns of Resubmission. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (Espoo, Finland) (ICER '18)*. Association for Computing Machinery, New York, NY, USA, 178–186. <https://doi.org/10.1145/3230977.3231006>
- [40] Farhan Ullah, Junfeng Wang, Muhammad Farhan, Sohail Jabbar, Zhiming Wu, and Shehzad Khalid. 2020. Plagiarism detection in students' programming assignments based on semantics: multimedia e-learning based smart assessment methodology. *Multimedia tools and applications* 79, 13 (2020), 8581–8598.

- [41] Szymon Wasik, Maciej Antczak, Jan Badura, Artur Laskowski, and Tomasz Sternal. 2018. A survey on online judge systems and their applications. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–34.
- [42] David Wolpert. 1992. Stacked Generalization. *Neural Networks* 5 (12 1992), 241–259. [https://doi.org/10.1016/S0893-6080\(05\)80023-1](https://doi.org/10.1016/S0893-6080(05)80023-1)
- [43] Fabiana Zaffalon, André Vargas, Ricardo Souza, Rafael Penna, Jean Bez, Neilor Tonin, and Sílvia Botelho. 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 (2019), 459. <https://doi.org/10.5753/cbie.sbie.2019.459>
- [44] Wayne Xin Zhao, Wenhui Zhang, Yulan He, Xing Xie, and Ji-Rong Wen. 2018. Automatically learning topics and difficulty levels of problems in online judge systems. *ACM Transactions on Information Systems (TOIS)* 36, 3 (2018), 1–33.