UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MARIANA MENDES E SILVA

USO DE ANÁLISE ESTÁTICA PARA IDENTIFICAR OBJETIVOS DE APRENDIZAGEM EM CÓDIGOS DE INICIANTES EM PROGRAMAÇÃO ORIENTADA A OBJETOS

CAMPINA GRANDE - PB

2019

MARIANA MENDES E SILVA

USO DE ANÁLISE ESTÁTICA PARA IDENTIFICAR OBJETIVOS DE APRENDIZAGEM EM CÓDIGOS DE INICIANTES EM PROGRAMAÇÃO ORIENTADA A OBJETOS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharela em Ciência da Computação.

Orientadora: Professora Dra. Eliane Cristina de Araújo.

CAMPINA GRANDE - PB

2019

S586u Silva, Mariana Mendes e.
Uso de análise estática para identificar os objetivos de aprendizagem em códigos de iniciantes em Programação Orientada a Objetos. / Mariana Mendes e Silva. – 2019.

9 f.

Orientadora: Profa. Dra. Eliane Cristina de Araújo.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Programação Orientada a Objetos. 2. Disciplina Programação Orientada a Objetos - UFCG. 3. Aprendizagem em programação. 4. Ensino de programação. 5. Objetivos de aprendizagem - programação. 6. Iniciantes em programação. 7. Códigos – análise estática. I. Araújo, Eliane Cristina de. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

MARIANA MENDES E SILVA

USO DE ANÁLISE ESTÁTICA PARA IDENTIFICAR OBJETIVOS DE APRENDIZAGEM EM CÓDIGOS DE INICIANTES EM PROGRAMAÇÃO ORIENTADA A OBJETOS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharela em Ciência da Computação.

BANCA EXAMINADORA:

Professora Dra. Eliane Cristina de Araújo
Orientadora – UASC/CEEI/UFCG

Professor Dr. Cláudio de Souza Baptista
Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 25 de novembro de 2019.

CAMPINA GRANDE - PB

Uso de Análise Estática Para Identificar Objetivos de Aprendizagem Em Códigos de Iniciantes Em Programação Orientada a Objeto

Trabalho de Conclusão de Curso

Mariana Mendes
mariana.silva@ccc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil

Eliane Araújo*
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
eliane@computacao.ufcg.edu.br

RESUMO

Durante o processo de aprendizado de qualquer conteúdo é importante, ou pelo menos desejável, que os alunos consigam exercitar os conceitos apresentados em teoria. No contexto de programação não é diferente, é crucial que exercícios sejam realizados constantemente, especialmente quando um novo paradigma está envolvido. No ensino de Programação Orientada a Objeto é comum que sejam realizadas muitas atividades práticas além dos testes avaliativos. Este trabalho pretende verificar se os objetivos de aprendizagem propostos para os estudantes em determinada atividade podem ser identificados através de análise estática dos códigos produzidos e qual seria a relação destes com o desempenho do estudante de forma geral. Usamos dois métodos diferentes para análise estatística dos dados. O primeiro nos mostrou que pode não existir uma relação muito nítida entre exercitar um determinado objetivo de aprendizagem e a nota, além disso aponta também quais objetivos mais pesaram na variabilidade dos dados. Já no segundo método usado, foi possível observar que é possível explicar pouco mais de 60% da variável nota. Observamos que algumas variáveis, representando os objetivos de aprendizagem, se destacaram; a implementação de testes, uso do padrão herança e o uso de interface, ou seja, influenciam consideravelmente no desempenho do aluno.

1 INTRODUÇÃO

A Programação Orientada a Objetos é um paradigma que possui grande importância no contexto do desenvolvimento de software e hoje tem grande visibilidade no meio comercial, devido à capacidade de reuso e fácil manutenção de código. Dentre o meio acadêmico, no mundo inteiro, esse paradigma está presente nas ementas [3], principalmente porque ele permite usar abstrações, fazendo com que seja possível pensar em responsabilidades usando a ideia de encapsulamento.

Na disciplina de Laboratório de Programação II na Universidade Federal de Campina Grande (UFCG) a linguagem de programação base usada é Java, que dá suporte ao paradigma OO. Nessa disciplina são realizadas provas, atividades de laboratórios e um pequeno projeto ao final do curso. Todas essas práticas fazem parte do cálculo na nota final dos alunos.

As atividades de laboratórios, também chamada de labs, podem ser realizadas dentro do intervalo de mais ou menos uma semana. Dúvidas sobre a implementação podem ser pesquisadas ou sanadas com a ajuda de professores ou monitores durante a semana. Essas atividades são corrigidas por monitores e o seu peso na nota final pode variar, mas é sempre menor do que o da prova e do projeto. Já as provas precisam ser realizadas num tempo limite de 2 horas, não há pesquisa ou consulta e são corrigidas por professores.

Os objetivos de aprendizagem que devem ser exercitados nos labs são aqueles que, possivelmente, serão cobrados na avaliação, portanto é importante que o aluno realize essa tarefa para se preparar. A correção de um lab pode ser demorada, consequentemente o *feedback* sobre o mesmo também. Além da espera pelo monitor, existe o fato de que o professor nem sempre consegue acompanhar, a partir de dados concretos, o desempenho geral e individual de cada estudante, nem mesmo saber, por exemplo qual conteúdo está sendo considerado mais difícil ou quem são as pessoas com entraves.

Este trabalho propõe uma análise automática, a partir de todas atividades práticas realizadas durante a disciplina, que seja capaz de auxiliar monitores e professores e trazer informações relevantes sobre o desempenho individual e geral. A literatura da área mostra que os resultados de uma atividade podem ser elucidadores, revelando quais aspectos devem ser explicitamente ensinados e como alcançar um equilíbrio entre os vários aspectos do ensino de um conceito [5].

Para dar suporte às análises desenvolvemos uma ferramenta automática que identifica se o aluno implementou ou não determinado objetivo de aprendizagem. A mesma técnica foi utilizada para códigos de provas e códigos de atividades de laboratório. Na validação dos resultados obtidos através da ferramenta foram usados dois métodos estatísticos diferentes, na tentativa de gerar mais conhecimentos sobre dados obtidos.

Este trabalho está estruturado da seguinte forma: na próxima seção discutiremos trabalhos relacionados. Na Seção 3, a metodologia usada no estudo, na seguinte os resultados obtidos, separando em subtópicos cada método estatístico utilizado, por fim limitações, conclusões e os trabalhos futuros.

2 TRABALHOS RELACIONADOS

Usar análise automática de código para gerar relatórios ou *feedback* já é uma prática difundida no meio da computação. Nghi Truong em [7] desenvolveu um *framework* para avaliar a qualidade de um código e retornar como resposta como aquele código pode ser

melhorado. Dar um *feedback*, ajudar os estudantes e fazê-los pensar em outras soluções e na qualidade de código é uma tarefa difícil e talvez leve muito tempo.

Em um trabalho mais recente, de Edward [2] realiza uma análise estática para identificar os erros mais comuns dos estudantes na implementação de códigos Java, considerando também tópicos que não necessariamente afetam no comportamento do programa, como indentação e documentação de código. Diferente do primeiro trabalho apresentado, essa análise procurou uma relação entre a presença dos erros e as notas, o resultado não permite afirmar que os tópicos analisados possuem uma influencia considerável no desempenho de um aluno.

Outra forma de usar a análise estática no contexto de Orientação a Objeto, é identificar alguns padrões de projeto, como no trabalho de Antoniol [1], que faz essa identificação usando a coleta de métricas, um ponto interessante é que código e design são mapeados em uma representação intermediária, chamada Abstract Object Language (AOL), para manter a independência da linguagem de programação, a análise mostra uma precisão de 55% na identificação.

3 METODOLOGIA

Na UFCG, a disciplina de LP2 faz parte do início da graduação, sendo a disciplina que introduz conceitos de programação OO. Ao longo do curso os estudantes são expostos a muitos conceitos e padrões específicos, como o uso de coleções, herança, interface, testes de unidade, tipos abstratos, exceções e alguns tópicos sobre qualidade de código.

Como já visto anteriormente, a nota final da disciplina é composta por três tipos de atividades diferentes, provas, atividades de laboratório e projetos. Elas têm um número variado de funcionalidades a serem implementadas, mas todas são fornecidas pelos professores e envolvem conceitos vistos em teoria. Contudo, não iremos incluir os projetos nos dados, pois trata-se de uma atividade em grupo e aqui queremos analisar o desempenho individual e o geral.

O último estágio da disciplina é o que possui uma maior diversidade de conceitos que podem ser exercitados na prática, sabendo disso, as notas, os laboratórios e as provas utilizadas para análise fazem parte dessa etapa da disciplina. No total, 111 códigos de provas e 112 códigos estavam aptos para serem usados na análise, referentes aos períodos 2017.2 e 2018.1.

Para extração de dados foi desenvolvida uma ferramenta em java utilizando a biblioteca *javaparser* que é usada para análise e geração de código [6], mas nesse caso, utilizamos apenas a sua funcionalidade de análise. Com essa biblioteca conseguimos gerar para cada código uma unidade de compilação e percorrer todos os arquivos '.java'.

Para cada arquivo guardamos a informação de nome, assinatura dos métodos presentes, se é abstrata ou concreta, se é uma interface e seus atributos. Com essas informações vamos mapear os objetivos de aprendizagem a esses elementos encontrados, por exemplo, se queremos identificar se o aluno exercita o conceito classes abstratas, buscaremos nas classes coletadas alguma que seja abstrata. Os objetivos de aprendizagem que serão mapeados foram escolhidos manualmente a partir das diretrizes de correção fornecidas pelos professores, isso é válido para as provas e para os *labs*. Objetivos de aprendizagem escolhidos e como se deu a estratégia para mapeá-los:

- `Controller`: Indicará se o aluno fez uso do padrão através da busca textual, geralmente os nomes das classes que exercem a função de Controller incluem Controller, Controlador ou Sistema, então para cada nome de classe encontrado, verificamos se ele inclui alguma das palavras chaves.
- `Facade`: Uso do padrão *Facade*. Também feito a partir da busca textual usando as palavras *Facade* ou Fachada. Usando a mesma estratégia para identificar uso do *Controller*.
- `hashMap`: Indica se o aluno usou *hashMap* em alguma classe do código. A estratégia usada para buscar o uso do *hashMap* usa o javaparser para guardar os atributos de uma classe, e a busca textual para identificar se o tipo daquele atributo inclui as palavras *Map* ou *hashMap*.
- `Testes`: Indica se testes foram implementados para o código. A verificação de existência de testes é feita tanto pela presença da anotação '@Test' no método, como também pela presença de *assert* na implementação do corpo do método de teste usando a busca textual.
- `Documentacao`: Verifica se existe pelo menos 3 métodos e/ou classes com documentação.
- `Heranca`: Identifica se alguma classe herda de outra, ou seja, se o aluno usou *extends* em alguma classe. Feita unicamente usando o javaparser.
- `Interface`: Identifica a criação de interfaces.
- `Classes Abstratas`: Identifica o uso de classes abstratas no código.
- `Equals`: Indica se o aluno implementou o método *equals* de alguma classe. A estratégia utilizada envolve o *javaparser*, que guarda os métodos e a busca textual para identificar o nome do método.
- `HashCode`: Indica se o aluno implementou o método *hashCode* de alguma classe. Mesma estratégia usada para identificar *equals*.
- `Exception` : Indica se o código contém métodos que lançam exceções.

Para cada atividade de laboratório ou prova como entrada, a ferramenta retorna uma lista, que indica se o conceito foi encontrado no código (1) ou se o conceito não foi encontrado no código (0). A tabela 1 mostra um exemplo de saída para um código submetido.

A tabela 1 é um exemplo de resposta da ferramenta. Indica que o aluno usou herança, interface, *hashMap* e Fachada, mas não implementou *equals*, *hashCode* ou testes, não usou classes abstratas, não usou *controller* e não fez a documentação do código.

A partir dessas respostas, realizamos duas análises, na primeira tentamos localizar grupos que se formaram a partir das semelhanças e diferenças do que foi exercitado no código ou não, e depois tentamos relacionar com as notas. Já na segunda forma de análise tentamos explicar as notas também usando os mesmos dados da análise anterior, as variáveis que indicam a presença ou não de um objetivo de aprendizagem.

4 RESULTADOS

Para explicar os resultados obtidos a partir dos 222 códigos válidos, unindo provas e laboratórios usamos duas técnicas separadamente,

Tabela 1: Modelo de Saída

Objetivo de Aprendizagem	Resultado
controller	0
facade	1
hashMap	1
testes	1
documentação	0
herança	1
interface	1
classes abstratas	0
equals	0
hashCode	0
exception	1

Tabela 2: Principais Componentes Usando Todos os Códigos

Objetivo de Aprendizagem	PC1	PC2
facade	0.3014265	-0.2046022
controller	-0.0237928	-0.0236780
herança	0.3370666	-0.2950991
interface	-0.2644563	0.0646584
classes_abstratas	0.3207531	-0.3093307
exception	-0.1012679	0.0876481
hashCode	-0.3779977	-0.5803891
equals	-0.3779977	-0.5803891
documentação	0.3956926	-0.2809191
testes	0.1266961	-0.0304288
hashMap	-0.3917884	0.0969591

o PCA e a Regressão Linear Múltipla. Para cada um dos métodos vamos fazer uma análise geral, contando com todos os códigos e depois analisar separadamente, códigos de prova e códigos de laboratórios. A decisão de também analisar as práticas separadamente aconteceu devido às diferenças de correções e execução.

Como já visto antes, os labs tem um tempo maior para serem desenvolvidos, além das consultas que podem ser realizadas, ou seja, mesmo que o aluno não saiba ou não entenda como exercitar aquele objetivo de aprendizado, possivelmente irá pedir assistência do monitor ou professor, então é muito provável que esse indivíduo consiga implementar mais requisitos numa atividade de laboratório do que numa prova. Ainda nesse contexto, mesmo que o aluno implemente de fato todos os objetivos estabelecidos, seu desempenho pode não ser um dos melhores, pois não conseguimos mensurar a qualidade da implementação ou coesão na abordagem usada neste trabalho.

Por outro lado, as provas têm um tempo bem menor de duração com apenas 2 horas, mas abordam os mesmo objetivos que o lab anterior a elas. Durante essa atividade os alunos não podem fazer consultas online, nem tirar dúvidas sobre os assuntos que estão sendo cobrados. Sendo assim, possivelmente se um aluno não possui conhecimento suficiente para abordar algum objetivo de aprendizagem, ele não vai implementar, pois não há como tirar dúvidas.

4.1 PCA

O método PCA procura reduzir a dimensionalidade dos dados, para que se possa visualizá-los de forma clara. Ele é usado quando o dado é composto por um grande número de variáveis e, consequentemente, a análise e visualização da relação entre elas é mais difícil. O objetivo do método PCA é identificar componentes que possam explicar os dados agrupando-os de acordo com suas semelhanças e mostrar como os mesmos são influenciados pelas variáveis em questão.

Na tabela abaixo observamos os dois principais componentes resultantes da união de todos os códigos e como cada variável influenciou no PC1 e no PC2.

A partir da tabela 2, é possível observar na primeira coluna o nome das variáveis e na segunda a influencia de cada uma delas sobre o PC1 (*Principal Component 1*). Na terceira coluna é possível observar a influencia de cada variável sobre o PC2 (*Principal Component 2*). Sabendo disso e analisando o PC1, vemos que cada uma das variáveis influencia menos de 50% na explicação do componente, sendo o maior coeficiente 0.3956926, atrelado a *documentação*. Já no PC2, temos o valor máximo de -0.580389 para as variáveis *hashCode* e *equals*, no entanto todas as outras possuem um valor pouco significativo para o grupo. Isso quer dizer que essas duas variáveis tem um peso maior sobre os dados e muitos códigos se diferenciam um dos outros por possuírem ou não os métodos de *equals* e *hashCode*.

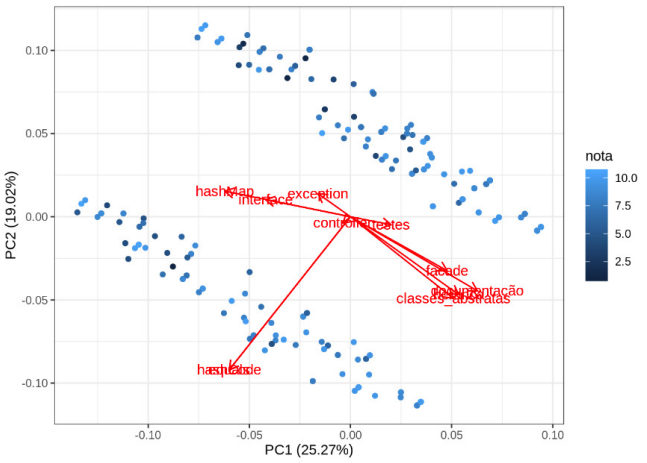

Figura 1: Distribuição dos resultados de cada código de acordo com a influencia das variáveis em cada componente.

No gráfico conseguimos ver o quanto cada componente explica os dados na sua totalidade. PC1 explica 25.27% e o PC2, 19.0225.27%, ou seja, ambos os componentes explicam menos do que a metade da variação dados. Visualmente é possível identificar que os pontos se distribuem em dois grupos bem definidos, contudo não é possível, com as análises atuais, apontar quais fatores influenciaram nessa distribuição. Ainda, analisando o outro canal da nossa visualização, é notório que as notas não seguem nenhum padrão, nem se relacionam diretamente com as variáveis ou com os grupos formados.

Agora, separadamente e usando apenas os resultados obtidos a partir dos códigos das provas, iremos observar uma tabela semelhante a tabela 1.

Tabela 3: Principais Componentes Usando Provas

Objetivo de Aprendizagem	PC1	PC2
facade	0.1534928	0.3898197
controller	0.1271979	0.4038966
herança	0.1965294	0.5253283
interface	0.2151762	-0.3356680
classes_abstratas	0.0745595	0.4405093
exception	-0.0636455	-0.1145711
hashCode	0.6512211	-0.1357058
equals	0.6512211	-0.1357058
documentação	-0.0810502	0.0406650
testes	0.0360140	-0.2178226
hashMap	0.0983978	0.0561947

Assim como na análise para todos os códigos, os componentes encontrados usando só as provas também são pouco influenciados pelas variáveis utilizadas. No entanto, há um peso maior para as variáveis *hashCode* e *equals* dentro do PC1, chegando a 65.122%, enquanto o restante possui um coeficiente abaixo de 25%. Já no segundo componente, temos uma distribuição maior entre as influências das variáveis, sendo *herança* a que possui mais peso, seguida de *classes_abstratas* e *controller*. Mas antes de concluir algo sobre os dados, precisamos saber o quanto cada componente explica a variabilidade.

No gráfico abaixo temos a distribuição das provas pelos componentes PC1 e PC2.

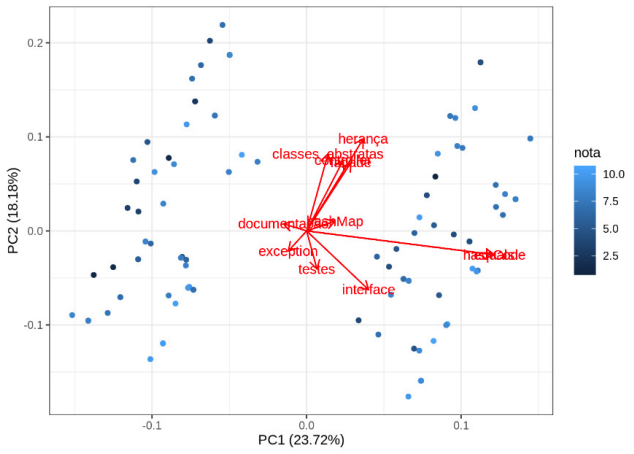

Figura 2: Distribuição dos resultados de cada código de acordo com a influência das variáveis em cada componente.

O PC1 consegue explicar 23.72% e o PC2 18.18%. As porcentagens são sutilmente menores que as anteriores e a distribuição forma novamente dois grupos, as notas se mostraram sem nenhum padrão visual, dessa forma, usando o PCA ainda não conseguimos inferir pontos relevantes sobre o exercício dos conceitos e o desempenho do individuo na avaliação. O resultados são semelhantes até aqui, visto que não houve nenhuma variação brusca no quanto os componentes explicam.

Por último, vamos analisar apenas os resultados dos laboratórios separadamente.

Tabela 4: Principais Componentes Usando Atividades de Laboratório

Objetivo de Aprendizagem	PC1	PC2
facade	-0.0381583	0.0052589
controller	-0.0227900	-0.5353334
herança	0.0287880	0.0328046
interface	0.0153635	-0.2918807
classes_abstratas	0.1265168	0.6501957
exception	0.0129205	-0.0303882
hashCode	0.6792908	-0.1277453
equals	0.6792908	-0.1277453
documentação	0.1781619	0.4047312
testes	-0.0355314	0.0314401
hashMap	0.1578294	0.0775148

Levando em consideração as duas últimas análises, temos que o valor máximo da influência de uma variável na variabilidade dos dados foi encontrado quando usamos apenas os resultados das atividades de laboratórios. Existe um peso considerável na implementação dos métodos *equals* e *hashCode*, eles estão ligados a um coeficiente de 0.679, o maior dentro do PC1, enquanto as outras variáveis influenciam no máximo 17.8% na variabilidade. Sobre o PC2, mais uma vez, *classes_abstratas* se destacam, dessa vez com um número maior do que nas amostra de provas, com 0.650, seguida de *documentação*.

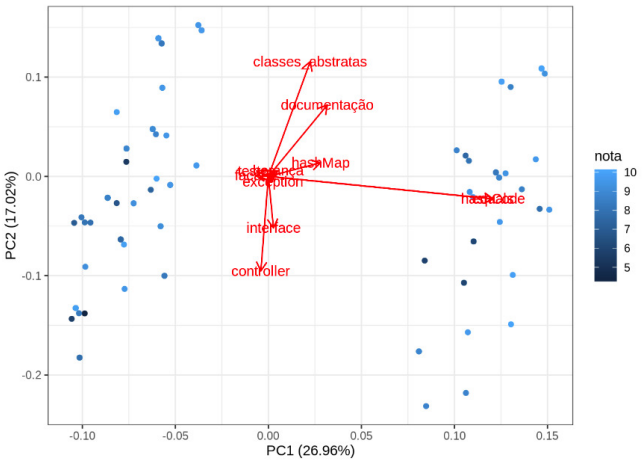

Figura 3: Distribuição dos resultados de cada código de acordo com a influencia das variáveis em cada componente.

Nessa última representação do PCA temos o PC1 explicando 26.97% e o PC2 17.02% da variabilidade dos dados. A distribuição

forma visualmente dois grupos, contudo, diferentemente das outras distribuições, usando apenas os laboratórios é possível observar algum padrão, em ambos os grupos as notas parecem estar maiores na parte superior do gráfico, além disso, a concentração desses pontos mais claros do lado direito parece ser maior do que no lado esquerdo. Então, possivelmente, no caso dessas atividades, quando o aluno implementa métodos de *equals* e *hashCode*, variáveis que ficaram mais a direita no gráfico, a nota tende a ser maior, bem como fazer uso de classes abstratas e fazer documentação parcial do código, também parece elevar a nota.

4.2 Regressão Múltipla

Regressão múltipla é uma coleção de técnicas estatísticas para construir modelos que descrevem de maneira razoável relações entre várias variáveis explicativas de um determinado processo. A diferença entre a regressão linear simples e a múltipla é que na múltipla são tratadas duas ou mais variáveis explicativas.

Nossas variáveis explicativas serão justamente as que foram obtidas a partir da ferramenta, ligadas a um objetivo de aprendizagem. Usando essa técnica, vamos exibir os modelos encontrados para descrever a relação entre nossas variáveis explicativas e a nossa variável resposta, a nota do aluno. Seguiremos a mesma ordem usada no método anterior, apresentando primeiro a análise usando todos os códigos.

Modelo Usando Todos os Códigos

*nota = 5.167 - (0.2757 * facade) - (0.3002 * controller) - (0.0932 * exception) + (1.0051 * herança) - (0.1622 * hashMap) + (2.6591 * testes) + (0.1316 * equals) + (0.1316 * hashCode) + (0.6179 * interface) + (0.6083 * classe_abstrata)*

O modelo apresentado consegue explicar 47.82% (R^2 = 0.4782) da nota. Mesmo não sendo uma porcentagem acima de 50%, o que pode tornar o resultado pouco relevante, podemos observar que alguma variáveis se sobressaem, como por exemplo Testes e Herança, contribuindo mais do que as outras para o modelo.

Modelo Usando Código das Provas

*nota = 4.015 - (0.60508 * facade) - (0.03655 * controller) + (0.28476 * exception) + (0.05499 * Herança) + (0.39235 * hashMap) + (3.02713 * testes) + (0.08197 * equals) + (0.08197 * hashCode) + (1.45936 * interface) + (1.02445 * classe_abstrata)*

Esse segundo modelo, usando apenas os códigos das provas, nos mostra um coeficiente um pouco maior, pois consegue explicar 60.75% da nota. Também nele, assim como no primeiro, observamos que Teste é a variável que mais afeta o modelo, em seguida temos Interface e classe_abstrata com valores relevantes.

Modelo Usando Código dos Laboratórios

*nota = 5.04101 - (0.18287 * facade) - (0.19136 * controller) - (0.10531 * exception) + (2.33319 * herança) + (0.29382 * hashMap) + (1.64019 * testes) + (0.14771 * equals) + (0.14771 * hashCode) - (0.04985 * interface) + (0.35728 * classe_abstrata)*

Usando os resultados dos laboratórios temos um modelo que explica apenas 29.99%, que foi a menor porcentagem dos três modelos até agora. Ainda assim vemos que testes ainda continua como uma das variáveis que mais influencia o modelo, logo depois de Herança.

5 LIMITAÇÕES

Apesar de alguns resultados promissores ainda temos algumas limitações relacionadas a nossa análise.

Forma de identificar alguns objetivos de aprendizagem: alguns padrões da programação Orientada a Objeto tais como *Controller* e *Facade* talvez precisem de outras formas para serem identificados, pois como foi descrito antes, esses dois objetivos são localizados apenas através da busca textual, supondo que os alunos seguem o padrão de nomenclatura citado anteriormente, portanto, o fator pode atrapalhar nos resultados pois o aluno pode ter implementado os padrões, mas fugindo da nomenclatura, o que é totalmente aceitável.

Além dos padrões apresentados, a implementação dos métodos *equals* e *hashCode* também possui uma limitação. Esses dois métodos são utilizados para que seja possível diferenciar instâncias de uma mesma classe de acordo com um ou mais atributos, e é importante que o aluno exercite e saiba como e quando utilizar tais métodos. Contudo, existe uma solução alternativa para essa identificação, como por exemplo criar um atributo na classe que seja um identificador único, garantindo que cada instância terá um valor diferente, essa solução terá quase o mesmo efeito que usar *equals* e *hashCode*, então a nota final pode não ser afetada pela ausência dos mesmos.

No caso do uso de *hashMap*, a única limitação existente é de que a análise não consegue identificar se a coleção está sendo usada da forma adequada e armazenando as entidades corretamente, ou seja, o desempenho pode não ser afetado pelo uso ou não da coleção. Já no caso dos testes, não é possível identificar se ao executá-los estarão funcionando ou não, além de que o teste pode ser muito simples e raso, ainda assim a ferramenta vai apontar que esse objetivo de aprendizagem foi alcançado.

Forma de identificar alguns objetivos de aprendizagem: a nota final das atividades e das provas varia de 0 a 10, mas é formada por componentes menores e com pesos diferentes. Isso pode afetar nos resultados, pois na análise realizada consideramos que todos os objetivos possuem pesos iguais.

Aspectos mais complexos da nota: ainda sobre a de divisão de notas, temos que parte da avaliação leva em consideração qualidade de código, corretude das funcionalidades e estratégias utilizadas para resolução dos problemas, pontos que não são identificados na análise estática realizada e que, por serem complexos, precisam de correção mais minuciosa. Então, é possível que essa porcentagem mais complexa da nota influencie negativamente nos resultados, pois não estamos considerando-a.

Diferenças entre as atividades de laboratório e provas: como citado anteriormente, as duas práticas possuem diferenças que podem afetar diretamente os resultados, como não exercitar o objetivo de aprendizagem por não saber durante uma prova ou exercitar de forma equivocada nas atividades de laboratório.

6 CONCLUSÕES

A análise estática proposta é, possivelmente, um mecanismo promissor para identificação de objetivos de aprendizagem nos códigos de alunos iniciantes em POO. Analisamos os resultados da ferramenta de duas formas para tentar achar a mais adequada ao tipo do dado. No contexto desse trabalho, a Regressão Múltipla pareceu se encaixar melhor nos objetivos estabelecidos, o que não exclui a possibilidade de tentar buscar métodos alternativos.

Então, podemos dizer que é possível realizar uma análise automática para ter uma visão geral do aprendizado do aluno. Visualizar objetivos de aprendizagem menos exercitados e mais exercitados, como está o desempenho geral de uma turma e de cada aluno, além de como a implementação dos objetivos influencia na nota final, gerando um relatório completo para os professores. Além disso, a análise automática pode beneficiar os monitores da disciplina, pois poderão gerar um *feedback* rápido sobre as atividades de laboratório, assim, consequentemente o aluno poderá receber *feedback* contínuo sem nenhum custo para o monitor.

7 TRABALHOS FUTUROS

Como trabalhos futuros, pretendemos fazer uso de novas estratégias para mitigar as limitações observadas. Para fazer a identificação de padrões, usaremos um métodos mais concretos, podendo usar até ferramentas externas como o *JStereocode* [4], que identifica automaticamente estereótipos de uma classe. Também para reduzir as limitações, buscar um meio de reduzir o impacto da porcentagem da nota que diz respeito aos aspectos não funcionais do código, como qualidade de código e estratégias usadas.

REFERÊNCIAS

[1] Giuliano Antoniol, Roberto Fiutem, and Luca Cristoforetti. 1998. Using metrics to identify design patterns in object-oriented software. In *Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No. 98TB100262)*. IEEE, 23–34.

[2] Stephen H Edwards, Nischel Kandru, and Mukund Rajagopal. 2017. Investigating static analysis errors in student Java programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 65–73.

[3] ACM Joint Task Force. 2013. Association for Computing Machinery (ACM) IEEE Computer Society; 2013. *Computer Science Curricula* (2013).

[4] Laura Moreno and Andrian Marcus. 2012. Jstereocode: automatically identifying method and class stereotypes in java code. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 358–361.

[5] Noa Ragonis and Mordechai Ben-Ari. 2005. On understanding the statics and dynamics of object-oriented programs. In *ACM SIGCSE Bulletin*, Vol. 37. ACM, 226–230.

[6] Federico Tomassetti, N Smith, C Maximilien, and S Kirsch. 2017. JavaParser.

[7] Nghi Truong, Paul Roe, and Peter Bancroft. 2004. Static analysis of students' Java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*. Australian Computer Society, Inc., 317–325.