UNIVERSIDADE DO ESTADO DA BAHIA

DEPARTAMENTO DE CIÊNCIAS EXATAS E DA TERRA

CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

EDER PEREIRA DOS SANTOS

AI+RTESTING: REMOÇÃO DE REDUNDÂNCIAS DE CONJUNTOS DE CASOS DE TESTE A PARTIR DE DADOS DE EXECUÇÃO DO TESTE DE MUTAÇÃO

SALVADOR

2018

EDER PEREIRA DOS SANTOS

AI+RTESTING: REMOÇÃO DE REDUNDÂNCIAS DE CONJUNTOS DE CASOS DE TESTE A PARTIR DE DADOS DE EXECUÇÃO DO TESTE DE MUTAÇÃO

Monografia apresentada ao curso de Sistemas de Informação do Departamento de Ciências Exatas e da Terra da Universidade do Estado da Bahia- UNEB, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação. Área de Concentração: Engenharia de Software

Orientador: Me. Alexandre Rafael Lenz

SALVADOR

2018

Universidade do Estado da Bahia

Sistema de Biblioteca

Ficha Catalográfica - Produzida pela Biblioteca Edivaldo Machado Boaventura

dos Santos, Eder Pereira.

AI+RTesting: Remoção de Redundâncias de Conjuntos de Casos de Teste a partir de Dados de Execução do Teste de Mutação: / Eder Pereira dos Santos.-- Salvador, 2018.

92 : il.

Orientador: Alexandre Rafael Lenz

Trabalho de Conclusão de Curso (Graduação) - Universidade do Estado da Bahia. Departamento de Ciências Exatas e da Terra, 2018

1. Assunto 1: Qualidade. Assunto 2: Teste de Software. Assunto 3: Teste de Regressão. Assunto 4 Aprendizado de Máquina. Assunto 5: Redução.. I. Universidade do Estado da Bahia. Departamento de Ciências Exatas e da Terra.

CDD: 025.06

EDER PEREIRA DOS SANTOS

AI+RTESTING: REMOÇÃO DE REDUNDÂNCIAS DE CONJUNTOS DE CASOS DE TESTE A PARTIR DE DADOS DE EXECUÇÃO DO TESTE DE MUTAÇÃO

Monografia apresentada ao curso de Sistemas de Informação do Departamento de Ciências Exatas e da Terra da Universidade do Estado da Bahia- UNEB, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação. Área de Concentração: Engenharia de Software

Aprovada em:

BANCA EXAMINADORA

Me. Alexandre Rafael Lenz (Orientador)
Universidade do Estado da Bahia – UNEB

Prof. Dr. Eduardo Manoel de Freitas Jorge
Universidade do Estado da Bahia – UNEB

Prof. Dr. Mônica de Souza Massa
Universidade do Estado da Bahia – UNEB

AGRADECIMENTOS

A Deus, pelo dom da vida e por me acompanhar durante todo esse tempo. Não posso deixar de ressaltar que este trabalho não teria sido possível sem a orientação do Prof. Me. Alexandre Rafael Lenz, que compartilhou seus saberes e metodologia. Expresso minha gratidão a Profª. Ma. Débora Rego, pelo apoio e incentivo em todos os momentos. À todos os colegas de curso pelas palavras de incentivo nos momentos difíceis, em especial a amiga Thamires Asenate Botelho, pelo companheirismo, confiança e troca de conhecimento. Em especial, agradeço aos meus familiares, por respeitar as minhas ausências, falta de atenção e pelas palavras de estímulo.

"Antes de julgar a minha vida ou o meu caráter... calce os meus sapatos e percorra o caminho que eu percorri, viva as minhas tristezas, as minhas dúvidas e as minhas alegrias. Percorra os anos que eu percorri, tropece onde eu tropecei e levante-se assim como eu fiz"

(Clarice Lispector)

RESUMO

Um software evolui ao longo do tempo de vida, essa evolução acontece através da adição de novos requisitos ou realização de manutenções. No sentido de evitar a introdução de defeitos em decorrência de alterações no software é utilizado o Teste de Regressão. A atividade de Teste de Regressão consome um tempo considerável do total do desenvolvimento do software, aumentando o seu custo. Um dos fatores determinantes para o custo são as atividades executadas manualmente. Para contornar esse problema e apoiar a atividade de teste de regressão, muitas metodologias têm sido propostas visando reduzir o tempo despendido na atividade. Muitas dessas metodologias utilizam técnicas de aprendizado de máquina, as quais relacionam as informações coletadas durante a atividade de teste para identificação de casos de teste com comportamentos similares. Essas técnicas comumente permitem a geração de regras que podem ser aplicadas para a redução do conjunto de casos de teste. A redução é capaz de diminuir significativamente o tempo do teste de regressão, pois exclui permanentemente os casos de teste redundantes. O objetivo deste trabalho consiste em aplicar a metodologia proposta no projeto de pesquisa AI+RTesting definida pelo professor Alexandre Rafael Lenz para apoiar a redução do conjunto de casos de teste, a partir da implementação de uma ferramenta que automatize esse processo para programas escritos na linguagem Java em nível de método, ou seja, cobrindo o teste de unidade, com intuito de reduzir o tempo gasto na atividade de teste de regressão. A nova ferramenta integra o teste estrutural apoiado com a ferramenta Jabuti e o teste baseada em erros apoiado pela ferramenta Mujava. Junto com a API WEKA, adicionada a nova ferramenta para gerar os agrupamentos que possibilitam a aplicação de algoritmos de classificação para geração de regras, permitindo classificar as classes de equivalência em ordem de prioridade. Os resultados obtidos com os experimentos realizados na fase de validação demonstraram que o método baseado em PG é capaz de remover redundâncias, mostrando maior eficiência em programas mais simples, como observado quando aplicado no método MDC. Em comparação a programas mais complexos, que é o caso do segundo método testado, os resultados apresentados, demonstram que o método baseado em PG não consegue realizar a remoção de forma eficiente, devido a quantidade elevada de redundâncias encontradas. Para atingir resultados mais precisos, sugere-se a definição de uma metodologia que consiga realizar uma redução mais significativa que a apresentada utilizando a progressão geométrica, assim como a utilização de informações coletadas durante a atividade do teste estrutural devido à complementariedade das técnicas de teste de software.

Palavras-chave: Qualidade. Teste de Software. Teste de Regressão. Aprendizado de Máquina. Redução.

ABSTRACT

Software evolves over the lifetime, this evolution happens through the addition of new requirements or maintenance. In order to avoid introducing defects due to changes in the software, the Regression Test is used. The Regression Test activity consumes considerable time in the total development of the software, increasing its cost. One of the determining factors for the cost is the activities performed manually. To circumvent this problem and support the regression test activity, many methodologies have been proposed to reduce time spent on the activity. Many of these methodologies use machine learning techniques, which relate the information collected during the test activity to identify test cases with similar behaviors. These techniques commonly allow the generation of rules that can be applied to reduce the set of test cases. The reduction is able to significantly decrease the time of the regression test, since it permanently excludes the redundant test cases. The objective of this work is to apply the methodology proposed in the research project AI + RTesting defined by professor Alexandre Rafael Lenz to support the reduction of the set of test cases, from the implementation of a tool that automates this process for programs written in the language Java-level method, that is, covering the unit test, in order to reduce the time spent in the regression test activity. The new tool integrates the structural test supported by the Jabuti tool and the error-based test supported by the Mujava tool. Together with the WEKA API, we added the new tool to generate the clusters that allow the application of classification algorithms for generating rules, allowing to classify the equivalence classes in order of priority. The results obtained with the experiments carried out in the validation phase demonstrated that the PG-based method is able to remove redundancies, showing greater efficiency in simpler programs, as observed when applied in the MDC method. In comparison to more complex programs, which is the case of the second method tested, the results presented, demonstrate that the method based on PG can not perform the removal efficiently due to the high amount of redundancies found. In order to achieve more precise results, it is suggested to define a methodology that can accomplish a more significant reduction than that presented using geometric progression, as well as the use of information collected during the structural test activity due to the complementarity of the test techniques software.

Keywords: Quality. Software Testing. Regression Test. Machine Learning. Reduction.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AI+RTESTING	*Uma Metodologia para Testes de Regressão Aplicando Técnicas da Inteligência Artificial*
API	*Application Programming Interface*
AM	*Aprendizado de Máquina*
ARFF	*Formato de Arquivo de Atributo-Relação*
CTP	*Casos de Teste da classe de menor Prioridade*
FR	*Fator de Redução*
IA	*Inteligência Artificial*
KAIST	*Instituto Avançado de Ciência e Tecnologia da Coreira*
MDC	*Máximo Divisor Comum*
PG	*Progressão Geométrica*
RF	*Requisitos Funcionais*
RNF	*Requisitos não Funcionais*

SUMÁRIO

1 INTRODUÇÃO

Em virtude da crescente necessidade de sistemas computacionais que apoiem o desenvolvimento e otimização da atividade humana, a Engenharia de Software busca inserir novos métodos e práticas que visam aumentar a qualidade aliada à eficiência e redução do tempo das atividades de desenvolvimento (DELAMARO et al., 2007). No entanto, apesar dos esforços, erros são cometidos ao longo de todo o ciclo de desenvolvimento e manutenção de um software. De acordo com Mateo e Uaola (2012), os tipos de erros mais comuns são: especificação mal concebida, análise mal elaborada, projeto mal construído, codificação mal feita, gerando assim, sistemas com funcionalidades distintas e ou ausentes das especificadas, podendo acarretar em falhas de execução e comprometer a evolução do software.

A evolução do software é necessária, pois para que o mesmo se mantenha útil, é preciso adequar às mudanças das regras de negócio, do ambiente e das expectativas dos usuários, gerando assim novos requisitos e modificações em requisitos existentes. Segundo Sommerville (2010), a maioria das grandes empresas gastam mais na manutenção de sistemas existentes do que no desenvolvimento de novos. Devido ao problema da introdução de novos defeitos, surge o teste de regressão como alternativa. Os testes de regressão são executados para garantir que uma funcionalidade ou parte do software já testado, continue funcionando após o mesmo sofrer manutenção.

Segundo Sommerville (2010), o custo da atividade de teste de regressão gira em torno de 40% do total do desenvolvimento. Um dos fatores determinantes para o custo são as atividades executadas manualmente, como a criação de casos de teste e a rara utilização sistemática de técnicas de teste e manutenção, justificada pela falta de tempo para executar novamente os casos de teste já executados, logo o teste de regressão é deixado de lado.

Para contornar esses problemas e apoiar a atividade de teste de regressão são aplicadas as técnicas: (i) Funcional, que procura identificar as falhas na operacionalidade das funções que foram especificadas; (ii) Estrutural, que avalia o comportamento interno do componente de software; (iii) Baseada em Erros, que verifica se o software está livre de erros típicos cometidos pelos desenvolvedores.

A aplicação de técnicas de teste aliadas ao uso de critérios de teste são fundamentais para guiar a atividade de teste e para definir se a quantidade de testes executados foram suficiente,

provendo uma medida da cobertura dos testes feitos. De acordo com Maldonado (1997) a técnica Baseada em Erros, apoiada pelo critério Análise de Mutantes, se destaca das demais por dispor de métodos que têm como finalidade obter, de maneira sistemática, um conjunto de casos de teste que seja efetivo quanto à meta principal do teste, ou seja, revelar a presença de erros no programa.

Um problema apresentado por essa abordagem é a questão do tempo de execução dos testes. A alta cardinalidade dos conjuntos de mutantes e de um conjunto de casos de teste, necessários para realização do Teste de Mutação em um programa, faz com que esse critério apresente alto custo computacional para sua realização. Mesmo para pequenos programas o tempo de execução dos testes pode ser muito grande (OLIVEIRA, 2013).

Nota-se que, para o teste de regressão, a aplicação da técnica baseada em erros é ainda mais difícil, visto que o teste de regressão geralmente dispõem de menos recursos e tempo do que a primeira execução dos testes. Os requisitos de tempo e recursos devem ser considerados e, nesses casos, há a necessidade de se aplicar métodos capazes de reduzir a quantidade de casos de teste a serem utilizados para execução dos mutantes, visto que a reexecução de um grande conjunto de casos teste é no mínimo indesejável.

O Aprendizado de Máquina é amplamente utilizado na resolução de diversos problemas relacionados ao custo de execução de teste de software. Uma das possibilidades consiste em utilizar algoritmos de agrupamento, encontrando padrões entre os casos de teste de um determinado conjunto. Por exemplo M.Kartheek e Rajshekhar (2013) propuseram reduzir o tempo gasto na execução dos testes a partir da redução do número de casos de teste redundantes através de técnicas de aprendizado de máquina.

A partir dos resultados apresentados por Santos (2016) e Lenz (2010) nota-se que as informações geradas a partir da aplicação do critério Análise de Mutantes podem ser utilizados por técnicas de aprendizado de máquina para redução de redundâncias no conjunto de casos de teste. Essa redução visa eliminar permanentemente os casos de teste redundantes do conjunto de casos de teste, proporcionando uma redução do esforço gasto nesta atividade. Entretanto, é necessário avaliar a eficácia do conjunto de casos de teste reduzido com relação ao escore de mutação. Ainda conforme os resultados apresentados por Santos (2016) e Lenz (2010), pode-se observar que esse processo, quando executado manualmente, é complexo, lento e propenso a erros. Demonstrando a necessidade de avaliar os possíveis resultados a partir da automatização

do processo com a implementação de uma ferramenta.

Como justificativa para esta pesquisa, destaca-se a eficácia apresentada pelo critério Análise de Mutantes em obter, de maneira sistemática, um conjunto de casos de teste que seja efetivo para revelar defeitos. A importância de uma abordagem de redução do conjunto de casos de teste, uma vez que o custo da execução dos casos de teste sobre os mutantes pode ser reduzido quando são eliminadas as redundâncias do conjunto de casos de teste. O desenvolvimento de pesquisas que buscam diminuir o custo de aplicação do teste de mutação como em Mateo e Uaola (2012), que utilizou a técnica MUSIC (*MUtant Schema Improved with extra Code*), demonstrado a capacidade da técnica de identificar situações em que um mutante não deve ser executado, reduzindo assim o número total de execuções necessárias na análise de mutação. E da mesma maneira pesquisas que comprovaram a eficiência do teste de mutação mas salientam a necessidade de redução do custo da sua aplicação, como por exemplo a pesquisa desenvolvida por Jia e Harman (2011) que forneceu um levantamento detalhado de tendências na aplicação de técnicas e ferramentas de Teste de Mutação.

O objetivo da presente pesquisa é o desenvolvimento de uma ferramenta para apoiar o teste de regressão na redução do conjunto de casos de teste, a partir da aplicação de técnicas de aprendizado de máquina, para encontrar similaridade entre casos de teste, utilizando informações do teste de mutação, para remover redundâncias, com o intuito de reduzir o tempo e esforço de execução. Primeiramente são aplicados algoritmos de agrupamento, para encontrar similaridade entre casos de teste. Em seguida os agrupamentos gerados servem de base para geração de regras utilizando um algoritmo de classificação. Primeiramente são aplicados algoritmos de agrupamento, para encontrar similaridade entre casos de teste. Em seguida os agrupamentos gerados servem de base para geração de regras utilizando um algoritmo de classificação. Por fim, é aplicada uma metodologia baseada em progressão geométrica para remover permanentemente os casos de teste redundantes, de acordo com o Fator de Redução (FR) informado.

O presente trabalho está organizado em sete capítulos. O capítulo 2 explora as técnicas de Aprendizado de Máquina. O capítulo 3 explora os conceitos relacionados à atividade de teste, detalhando os aspectos das técnicas de teste de software, destacando o critério análise de mutantes. O capítulo 4 apresenta as metodologias para o teste de regressão. O capítulo 5 apresenta o método de redução de conjunto de casos de teste proposto. O capítulo 6 apresenta os resultados dos experimentos e validação. Por fim, o capítulo 7 apresenta a conclusão e os

trabalhos futuros.

2 APRENDIZADO DE MÁQUINA

O Aprendizado de Máquina (AM) é uma subárea da Inteligência Artificial (IA) dedicada ao desenvolvimento de métodos e técnicas que permitam ao computador aprender, isto é, que permitam ao computador aperfeiçoar seu desempenho em alguma tarefa.

No estudo da IA, existe uma grande variedade de tipos de aprendizado, os quais dividem-se em três grupos básicos, segundo (NORVIG; RUSSEL, 2013):

(i) No Aprendizado Supervisionado, o algoritmo de aprendizado (indutor) recebe um conjunto de exemplos de treinamento para os quais os rótulos da classe associada são conhecidos. Cada exemplo (instância ou padrão) é descrito por um vetor de valores (atributos) e pelo rótulo da classe associada;

(ii) No Aprendizado Não-Supervisionado o indutor analisa os exemplos fornecidos e tenta determinar se alguns deles podem ser agrupados de alguma maneira, formando agrupamentos ou*clusters*. Após a determinação dos agrupamentos, em geral, é necessário uma análise para determinar o que cada agrupamento significa no contexto do problema que está sendo analisado;

(iii) O Aprendizado por reforço é baseado em dados de um ambiente completamente observável. Sua meta é aprender o quanto a política é boa, ou seja, descobrir a sua utilidade. Para que seja possível realizar tais tarefas, o agente realiza uma série de testes no seu ambiente. Em cada etapa, ele aprende o próximo passo, somente o estado inicial e final são conhecidos, o que caracteriza o sistema de recompensa.

2.1 AGRUPAMENTO

De acordo com Jain et al. (2000) agrupamento é definido como uma técnica que utiliza o aprendizado não-supervisionado, para organização de uma coleção de dados em grupos com base na similaridade. Dessa forma busca-se particionar um conjunto de dados em grupos, submetendo-os a alguma regra que identifique padrões com base em características dos próprios dados.

Na realização dessa ação emprega-se um mecanismo denominado análise de *cluster* ou clusterização, que busca semelhanças entre padrões agrupando os padrões similares em categorias ou grupos, buscando extrair informação relevante de dados não rotulados. O mecanismo

relevante que um procedimento de agrupamento implementa é o de comparar dados entre si e agrupá-los convenientemente em grupos (JAIN et al., 2000).

A definição da medida de similaridade na maioria das situações é determinante para a indução de um agrupamento que seja significativo e que retrata a real natureza da organização dos dados, identificando assim os grupos de dados similares. Não necessita que os registros sejam previamente categorizados, diferentemente do método de classificação. E não tem a pretensão de classificar, estimar ou predizer o valor de uma variável.

Na literatura podem ser identificados vários algoritmos de agrupamento, apoiados pelos mais variados formalismos matemáticos e estatísticos. Santos (2016), realizou uma avaliação criteriosa para escolha dos algoritmos aplicados na sua pesquisa e observou a necessidade de realização de testes automatizados nos algoritmos listados abaixo.

De a cordo com Berkhin (2002), o algoritmo hierárquico cria uma hierarquia de relacionamentos entre os elementos ou decomposição hierárquica dos objetos usando algum critério que defina níveis de similaridade que mudam o agrupamento. A partir de uma matriz inicial de dados obtém-se uma matriz simétrica de similaridades e inicia-se a detecção de pares de casos em função do coeficiente de similaridade escolhido.

No algoritmo hierárquico é possível a utilização de dois métodos: aglomerativo e divisivo. O método aglomerativa, opera criando conjuntos a partir de elementos isolados, onde os dados são inicialmente distribuídos de modo que cada exemplo represente um *cluster* e, então, esses *clusters* são recursivamente agrupados considerando alguma medida de similaridade, até que todos os exemplos pertençam a apenas um *cluster*. E o método divisivo começa com um grande conjunto e vai quebrando-o em partes até chegar a elementos isolados, o processo inicia-se com apenas um agrupamento contendo todos os dados e segue dividindo recursivamente segundo alguma métrica até que alcance algum critério de parada, frequentemente o número de *clusters* desejados (BERKHIN, 2002).

Dempster (1977) definem que o algoritmo EM (*Expectation Maximization*) é uma ferramenta computacional utilizada para o cálculo do estimador de máxima verossimilhança (EMV) de forma iterativa, e é principalmente utilizado em problemas envolvendo dados incompletos. A ideia central é otimizar os parâmetros de uma função de distribuição de probabilidades, de forma que esta represente os dados da maneira mais verossímil possível.

De acordo com Bilmes (1998) o algoritmo EM é aplicado principalmente em duas situações, quais sejam, quando há valores faltantes e quando a função de verossimilhança é de ser obtida por outros métodos analíticos, sendo esta última muito utilizada na área de reconhecimento de padrões.

Por tratar-se de uma função complexa, o algoritmo EM busca iterativamente o ponto de máximo da função aplicando dois passos; O passo E descreve o melhor modelo de dados completo possível para os dados incompleto, dada todas as informações atuais. O passo M usa esse novo modelo completo para escolher as mais altas estimativas de verossimilhança dos parâmetros da distribuição para dados incompletos. A melhoria das estimativas dos parâmetros a partir do passo M conduz a um melhor modelo completo no passo E (OSOBA, 2013).

De acordo com MacQueen (1967) o algoritmo *K-Means*, é um método de particionamento não hierárquico, que fragmenta as observações dos dados em k *clusters* mutuamente exclusivos ou seja o método *K-Means* possui um algoritmo de aprendizagem que organiza n objetos em k partições onde cada uma representa um grupo. A distância entre um ponto Pi e um conjunto de *clusters*, dada por d(Pi,X), é definida como sendo a distância do ponto ao centro mais próximo dele.

Duda e Hart P.and Stork (2001) afirmam que o algoritmo de *K-Means* essencialmente define um conjunto de K centróides para os clusters. Cada amostra do conjunto de dados é então associada ao aglomerado com centróide mais próximo. Em seguida os centróides são recalculados e a associação das amostras a um aglomerado é refeita. Esse processo é repetido até que os centróides não sejam mais alterados.

O algoritmo funciona enquanto existir elementos na base que não façam parte de algum *cluester*, faz-se a comparação entre os elementos (nova informação comparada aos centróides dos *clusters* existentes) para poder incluir os novos.

O algoritmo *K-Means*, escolhe k distintos valores para centros dos grupos (possíveis e forma aleatória), onde k é o número de *clusters* (grupos) desejado e informado a priori, associando cada ponto ao centro mais próximo. E recalcula o centro de cada grupo, repetindo o processo até nenhum elemento mudar de grupo.

2.2 CLASSIFICAÇÃO

A classificação é um processo de mineração de dados, voltada à atribuição de uma das classes pré-definidas pelo analista a novos fatos ou objetos submetidos à classificação. Uma categoria de algoritmos muito utilizada para este fim compreende as árvores de decisão. Uma árvore de decisão é uma maneira simples de representar o conhecimento extraído de uma base de dados genérica (NORVIG; RUSSEL, 2013).

Rothermel et al. (2005) argumenta que o algoritmo J48 surgiu da necessidade de recodificar o algoritmo C4.5, que, originalmente, é escrito na linguagem C, para a linguagem Java. O algoritmo tem o objetivo de gerar uma árvore de decisão baseada em um conjunto de dados de treinamento, sendo este modelo usado para classificar as instâncias no conjunto de teste. Um dos aspectos para a grande utilização do algoritmo J48 pelos especialistas em mineração de dados é que o mesmo se mostra adequado para os procedimentos, envolvendo as variáveis (dados) qualitativas contínuas e discretas presentes nas bases de dados.

O algoritmo J48 apresenta o melhor resultado na montagem de árvores de decisão, a partir de um conjunto de dados de treinamento, permitindo a criação de modelos de decisão em árvores. A árvore de decisão é construída pela análise dos dados de treino e o modelo utilizado para classificar os dados ainda não classificados.

Para a montagem da árvore, o algoritmo J48 utiliza a abordagem de dividir para conquistar, onde um problema complexo é decomposto em problemas mais simples, aplicando recursivamente a mesma estratégia a cada subproblema, dividindo o espaço definido pelos atributos em subespaços, associando-se a eles uma classe (ROTHERMEL et al., 2005).

O algoritmo original (C4.5) exige que todos os valores de atributos sejam de domínio relativamente pequeno para que o desempenho computacional seja aceitável. No entanto, o algoritmo implementado pela ferramenta *Waikato Environment for KnowledgeAnalysis* (WEKA) é o J48, uma versão do tradicional algoritmo C4.5 que consegue, através da discretização de valores numéricos e de heurísticas, processar atributos de qualquer tipo na entrada.

2.3 FERRAMENTA WEKA

A ferramenta WEKA compreende um conjunto de algoritmos de aprendizado de máquina para tarefas de mineração de dados. Os algoritmos podem ser aplicados diretamente a

um conjunto de dados ou chamados a partir do próprio código Java, utilizando a API WEKA (JEYAMALA; SUBASHINI, 2014). A ferramenta WEKA foi concebida com propósito de que seus métodos sejam executados em conjuntos de dados da forma mais flexível possível.

De acordo com Jeyamala e Subashini (2014) a ferramenta WEKA contém algoritmos para pré-processamento de dados, classificação, regressão, agrupamento, regras de associação e visualização. A ferramenta também é adequada para o desenvolvimento de novos esquemas de aprendizado de máquina.

Para a aplicação de técnicas de mineração de dados é necessário que os dados a serem utilizados estejam de forma organizada. Estes arquivos podem estar em alguma estrutura de dados, planilha ou banco de dados. O formato de arquivo ARFF foi definido para organização dos dados utilizados pela ferramenta WEKA. O arquivo deve ser configurado com uma série de informações, dentre elas: domínio dos atributos, valores que os atributos podem representar e atributo classe.

3 A ATIVIDADE DE TESTE DE SOFTWARE

O teste de software é uma atividade que está inserida no processo de desenvolvimento de software. Segundo Sommerville (2010) o teste tem como objetivo mostrar que um programa faz o que é proposto a fazer e revelar os defeitos do programa antes do uso.

A execução da atividade de teste pode ser realizada de diversas formas, entretanto, independente de como a mesma é executada, o propósito central é gerar indícios de que o programa que está sendo testado não está correto, ou seja, o objetivo principal do teste é identificar o número máximo de defeitos dispondo do mínimo de esforço (PRESSMAN, 2011).

O teste é parte de um amplo processo de verificação e validação, que deve responder a dois questionamentos. Validação: estamos construindo o produto certo? Verificação: estamos produzindo o produto da maneira correta? Os processos de verificação e validação objetivam verificar se o software em construção satisfaz suas especificações (SOMMERVILLE, 2010).

Pressman (2011) define que ao longo do desenvolvimento o teste pode dar-se em três níveis de granularidade, teste de unidade, teste de integração ou teste de sistema.

O teste de unidade examina um elemento de software separadamente ou uma pequena coleção de componentes, com o propósito de explorar a menor unidade do projeto, procurando provocar falhas, o universo alvo desse tipo de teste são os métodos, as funções, as sub-rotinas, ou mesmo pequenos trechos de código.

Em seguida, tem-se o teste de integração que procura avaliar a comunicação entre os elementos ou componentes por meio de suas interfaces. O propósito do teste de integração é verificar a ligação entre os elementos de software que formam módulos de um sistema. Com ele é possível descobrir erros de interface entre os componentes do sistema.

Culminando no teste de sistema, que considera a coleção de todos os componentes que constituem um software sendo todo o domínio normalmente considerado em relação à sua especificação, avaliando o software em busca de falhas por meio da utilização do mesmo, tendo como objetivo verificar se o produto satisfaz seus requisitos. O teste de sistema engloba o teste de funcionalidade, o teste de desempenho, o teste de carga, o teste de segurança e situações anormais do sistema.

Os testes descritos utilizam valores de entrada, restrições para a sua execução e um resultado ou comportamento esperado. Essas variáveis e condições descrevem uma categoria particular a ser testada, conhecida como caso de teste.

É importante a escolha de casos de teste efetivos, devido ao custo e demora que os mesmos podem provocar no processo de teste. Criar casos de teste para um programa é uma atividade complexa que, se não for bem conduzida, pode gerar um conjunto muito grande de casos de teste que, muitas vezes, possuem redundâncias e não são efetivos em seu objetivo que é revelar a maior quantidade de defeitos dispondo do mínimo de esforço. Sendo assim, é relevante utilizar técnicas e estabelecer critérios de teste que avaliem os casos de teste de forma a aumentar as possibilidades de revelar defeitos, reduzindo o esforço da atividade e elevando o nível de confiança na correção do produto (DELAMARO et al., 2007).

3.1 TÉCNICAS DE TESTE DE SOFTWARE

A aplicação das técnicas de teste permite gerar, selecionar e avaliar casos de teste para utilização no processo de teste. As três principais técnicas de teste de software são: Técnica Funcional, Técnica Estrutural e Técnica Baseada em Erros. De acordo com Barbosa et al. (2000), a diferença entre as três técnicas é a origem da informação usada para avaliar ou para construir conjuntos de casos de teste. A aplicação dessas técnicas, através dos critérios que compõem cada uma, norteia a condução do teste de software.

Segundo Pressman (2011) , a técnica de Teste Estrutural também conhecida como Caixa Branca avalia o comportamento interno do componente de software. Essa técnica é aplicada diretamente sobre o código fonte do componente de software e baseia-se no conhecimento da estrutura interna da implementação para determinar quais partes do programa devem ser executadas, os aspectos de implementação são fundamentais na escolha dos casos de teste. Myers (1979) classifica o teste Estrutural em critérios baseados em fluxo de controle e critérios baseados em fluxo de dados. Em geral, a maioria dos critérios dessa técnica utiliza uma representação de programa conhecida como grafo de fluxo de controle ou grafo de programa.

No Teste Funcional, a seleção dos casos de teste é fundamentada nas propriedades descritas pela especificação do sistema em teste ou pelo ambiente operacional. Segundo Pressman (2011) , o teste funcional procura, entre outras coisas, mostrar que os requisitos funcionais do software são satisfeitos, que a entrada é adequadamente aceita, que a saída esperada é produzida

e que a integridade das informações externas é mantida. Alguns possíveis critérios de teste funcional que podem ser adotados são: particionamento em classes de equivalência, análise do valor limite e grafo de causa-efeito.

O Teste Baseada em Erros, segundo Delamaro et al. (2004) tem ênfase nos erros mais frequentes que o programador ou projetista pode cometer durante o desenvolvimento e nas abordagens que podem ser usadas para detectar a sua ocorrência. De acordo com Demillo et al. (1978) , a técnica de Teste Baseada em Erros pode ser aplicada através dos critérios Semeadura de Erros, Análise de Mutantes e Mutação de Interface. Esta técnica foi originalmente proposta por Demillo et al. (1978) e desde então vem sendo intensamente estudada. A técnica de Teste Baseada em Erros apoiada pelo critério Análise de Mutantes é apresentada de forma detalhada na próxima seção.

3.2 CRITÉRIO DE TESTE ANÁLISE DE MUTANTES

Entre os vários critérios propostos para se conduzir e avaliar a qualidade da atividade de teste destaca-se o Critério Análise de Mutantes da Técnica Baseado em Erros. O critério Análise de Mutantes, proposto por Demillo et al. (1978) , fundamenta-se em duas hipóteses, descritas a seguir.

(i) Hipótese do programador competente: programadores experientes escrevem programas muito próximos do correto. Assim, pode-se afirmar que erros são introduzidos nos programas a partir de pequenos desvios sintáticos que, mesmo que não causem erros sintáticos, alteram a semântica do programa e ocasionam defeitos.

(ii) E o feito de acoplamento: erros complexos são decorrentes de erros simples. Assim, espera-se que se existe um conjunto de testes que detecte erros simples cometidos pelos programadores em um programa então este poderá também revelar erros complexos.

A ideia básica do critério é gerar, para um programa que será testado, vários programas mutantes, cada mutante contempla pequenas alterações sintáticas, definidas pelos operadores de mutação. Os operadores de mutação mapeiam os erros mais frequentes cometidos pelos programadores. Logo, nota-se que os operadores de mutação dependem da linguagem de programação utilizada.

Para satisfazer o critério é necessário gerar um dado de teste que faça com que cada

mutante comporte-se diferentemente do programa original em teste. Neste caso, o mutante é dito morto. Quando não existir nenhum dado de teste capaz de distinguir um mutante do programa em teste, o mutante é equivalente. O escore da cobertura obtida através do número de mutantes mortos e número de mutantes gerados, descontando-se o número de mutantes equivalentes é usado para avaliar o conjunto de dados de teste. Os passos para aplicação do teste de mutação são detalhados na sequência.

3.2.1 Aplicação do Teste de Mutação

O teste de mutação realiza derivações do programa original criando diversos programas mutantes. Considerando a mutação de programa, um mutante é gerado se existir uma instrução correta, acerca da qual se pode aplicar um operador de mutação. Um determinado operador de mutação representa uma regra que especifica variações sintáticas de comandos que são geradas a partir de uma gramática. Assim, o resultado da aplicação de um operador de mutação sobre uma instrução válida é chamado de mutante (ELBAUM et al., 2003).

De acordo com Delamaro et al. (2007) , primeiramente um programa P é testado com um conjunto de casos de teste T criado para ele. À medida que defeitos vão sendo descobertas pelo teste, os erros causadores são corrigidos, até quando o conjunto de teste T não revelar mais defeitos. Nesse ponto, o programa P ainda pode conter defeitos que o conjunto T não conseguiu revelar. Em seguida, P sofre pequenas alterações dando origem a programas P', próximos de P, mas com erros simples, consistindo de pequenas alterações sintáticas. Para introduzir os erros, gerando cada um desses mutantes, operadores de mutação devem ser definidos com base nos erros típicos injetados por programadores. A seguir, cada um dos casos de teste do conjunto T é executado sobre o programa mutante P'. Se T for capaz de revelar o erro introduzido em P', falhando o teste, diz-se que o mutante foi morto. O objetivo é que todos os mutantes sejam mortos por T, porque quando algum mutante permanece vivo, significa que o conjunto de teste T é incapaz de revelar o defeito causado pelo erro sintático no ponto onde houve a mutação.

Durante a etapa de análise de mutantes do Teste de Mutação a intervenção do desenvolvedor se faz necessária. O nível de confiança da adequação dos casos de testes é obtido na análise de mutantes, a partir do escore de mutação, que relaciona o número de mutantes mortos com o número de mutantes não equivalentes gerados. Demillo et al. (1978), definem que dado um programa P e um conjunto de casos de teste T, o escore de mutação MS(P,T) é

calculado da seguinte forma: MS(PT) = DM(P T) / M(P) - EM (P) (DEMILLO et al., 1978).

Onde, de acordo com Demillo et al. (1978), as variáveis são assim definidas:

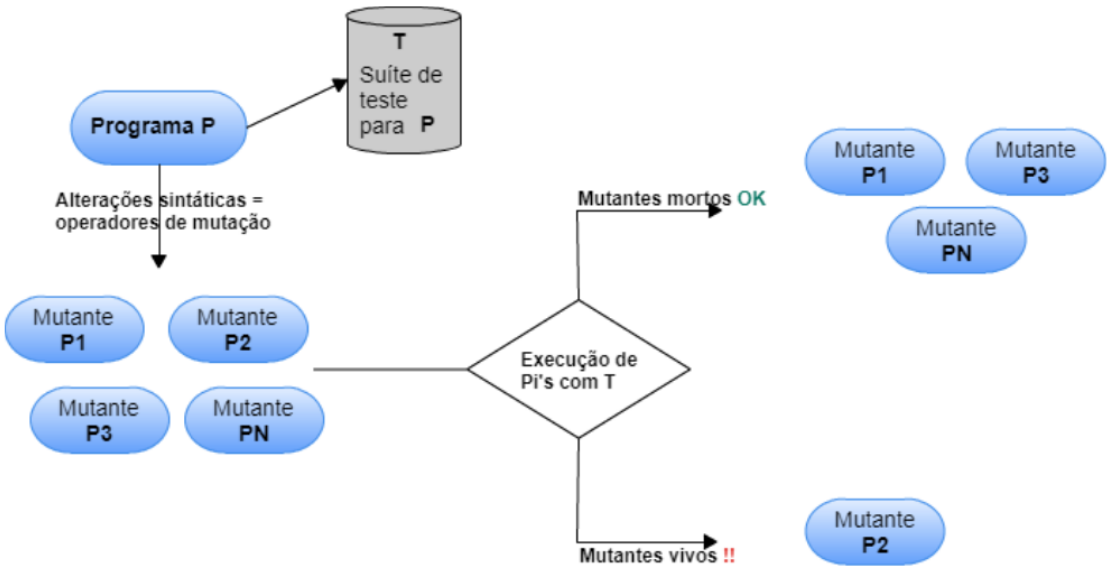
DM(P,T): define quantidade de mutantes mortos pelos casos de teste em T.

M(P): define quantidade total de mutantes gerados.

EM(P): define número de mutantes equivalentes a P.

O valor encontrado no escore de mutação varia entre 0 e 1, se o resultado obtido estiver mais próximo do valor 1, significa que o conjunto de casos de testes T tem um grau elevado de eficiência para testar o programa P, pois este foi capaz de conseguir matar um número considerável de mutantes gerados. O emprego do Teste de Mutação é realizado a partir da execução das fases seguintes: geração dos mutantes, execução do programa original, execução dos mutantes, análise dos mutantes vivos. A Figura 1 , ilustra as fases descritas para aplicação do Teste de Mutação, e descritos nas seções que se seguem.

Figura 1 – Esquema da aplicação do Teste de Mutação



Fonte – Elaborada pelo autor

Nas seções seguintes é apresentada uma explanação da Figura 1, para melhor entendimento.

3.2.2 Geração dos Mutantes e Operadores de Mutação

Quando é gerado um mutante, pretende-se que no conjunto de casos de testes exista pelo menos um capaz de reconhecer o erro criado.

Cada mutante é criado a partir de uma modificação sintática no código-fonte de P. Para geração do conjunto de mutantes P', as alterações são realizadas por meio dos operadores de mutação que foram aplicados. Esses operadores contêm regras responsáveis pelas alterações no programa original, cada um deles é aplicado em P para produzir um conjunto de mutantes, formando o conjunto de programas mutantes P'. A aplicação de um operador de mutação pode gerar mais que um mutante, pois o programa original P pode conter mais de uma estrutura sintática que está no domínio do operador. Dessa forma, o operador, quando aplicado, desempenha ações de mutação sobre cada uma dessas estruturas (OLIVEIRA, 2013).

A definição dos operadores de mutação é feita com o intuito de satisfazer dois objetivos. O primeiro consiste na reprodução de erros cometidos durante o processo de desenvolvimento de um sistema, de modo a que os testes criados sejam capazes de identificar estes erros. O segundo objetivo consiste em garantir que os testes são desenhados para validar com eficácia um determinado sistema.

Ma et al. (2005) definiram seis grupos de classes de operadores de mutação, com base nas características da linguagem Java: controle de acesso, herança, polimorfismo, sobrecarga, características específicas da linguagem Java e erros comuns de programação.

Nos primeiros quatro grupos estão representadas os erros associados às características comuns a todas as linguagens Orientadas a Objetos. No quinto grupo estão concretizados os erros específicos para a linguagem Java, como por exemplo, a definição de atributos da classe utilizando o modificador "this". Já o último grupo inclui operadores de mutação baseados nos erros comuns de programação.

Sendo a escolha dos operadores de mutação, um ponto importante nos testes de mutação e sabendo que a sua definição depende das características da linguagem de programação utilizada, é fundamental a priorização de casos de teste para a identificação dos operadores de mutação que satisfaçam a reprodução de erros cometidos durante o processo de desenvolvimento, com baixo custo.

3.2.3 Execução do Programa Original

Nesse passo, deve-se executar o programa original P para todos os casos de teste do conjunto T e aferir se o resultado é o esperado. Se o comportamento do programa for conforme o esperado, executa-se o passo seguinte (Execução dos Mutantes), caso o programa apresente um comportamento distinto para algum caso de teste, então um defeito foi revelado e o processo termina, uma vez que o programa original apresenta resultados incorretos para algum caso de teste. Os resultados da execução de cada caso de teste do conjunto T sobre o programa original P são armazenados para posterior comparação com os resultados do mesmo conjunto de casos de teste T sobre os programas mutantes contidos em P' (ELBAUM et al., 2003).

3.2.4 Execução dos Mutantes e Análise dos Mutantes Vivos

Este passo pode ser totalmente automatizado, não requerendo a intervenção do testador. Os casos de teste são executados nos programas mutantes, ou seja, cada um dos mutantes contidos em P' é executado contra os casos de teste de T Oliveira (2013). Os resultados das execuções dos mutantes são comparados com o resultado da execução do programa original P (saída esperada).

Considerando cada mutante individualmente, na execução dos casos de teste de T, são comparados os resultados obtidos para identificar os mutantes que foram mortos, se a saída gerada por um mutante for diferente da saída do programa original P, este mutante é dito morto, caso contrário, o mutante continua vivo.

A análise dos mutantes vivos é a etapa que objetiva decidir quais dos mutantes vivos são equivalentes ao programa original e requer a intervenção humana Oliveira (2013). Um mutante pode continuar vivo por um dentre dois motivos:

1. Ser equivalente ao programa original P;

2. O conjunto de casos de teste T selecionado não é adequado o suficiente para distinguir o comportamento entre o mutante e o programa original P.

Finalizado o processo de análise dos mutantes vivo para determinação dos equivalentes obtemos um indicativo da qualidade do conjunto de casos de teste T através do cálculo do escore de mutação conforme a fórmula apresentada anteriormente. Se o escore obtido for satisfa-

tório o processo termina; caso contrário, podem ser incluídos novos casos de teste no conjunto T a fim de matar os mutantes que ficaram vivos. Dessa maneira, é necessário retornar ao segundo passo do critério, executar P com os novos casos de teste, executar os mutantes, recalcular o escore de mutação e assim por diante até que se consiga um bom conjunto T (FUNABASHI, 2002).

Dependendo do número de mutantes gerados, bem como as características do programa em teste, para que a aplicação deste critério seja eficiente, é de fundamental importância o uso de ferramentas computacionais, principalmente na execução e comparação de resultados (FUNABASHI, 2002).

3.2.5 Custo de Execução do Critério Análise de Mutantes

Um dos principais problemas relacionado a aplicação do critério análise de mutantes é o seu custo de execução. O primeiro aspecto deste problema está diretamente relacionado com o grande número de mutantes que pode ser gerado mesmo para programas simples. Entre as pesquisas para contornar este aspecto de custo, têm sido investigadas abordagens para a redução do número de mutantes gerados.

Uma linha interessante descrita por Funabashi (2002) é a caracterização de um subconjunto de operadores de mutação que apresente o mesmo grau de adequação e eficácia, mas a um custo menor. Essa linha caracterizada por esses subconjuntos têm sido denominados de mutação restrita.

Também tem sido objetivo dessas pesquisas o estabelecimento de estratégias de teste incrementais que exploram as diversas características do critério, partindo-se inicialmente de operadores "mais fracos"e talvez menos eficazes para a avaliação da adequação do conjunto de casos de teste, e em função da disponibilidade e de tempo, incrementalmente, procurando-se satisfazer operadores "mais fortes"e eventualmente mais eficazes, porém, em geral, mais caros (ELBAUM et al., 2003).

A aplicação do critério análise de mutantes ao teste de regressão geralmente dispõem de menos recursos e tempo do que a primeira execução dos testes. Uma alternativa consiste em aplicar métodos capazes de reduzir a quantidade de casos de teste a serem utilizados para execução dos mutantes, diferentemente das linhas apresentadas anteriormente, esta não inter-

fere na geração e execução dos mutantes, mas sim na redução do conjunto de casos de teste, proporcionando também uma redução significativa do custo de execução do critério análise de mutantes.

3.2.6 A Ferramenta Mujava

A ferramenta Mujava é fruto da parceria entre as universidades *Korea Advanced Instituteof Science* e *Technology e George Mason University*. É uma ferramenta concebida para apoiar a aplicação do critério análise de mutantes para a linguagem Java. A Mujava possibilita a realização de testes automatizados, viabilizando a redução do ciclo de testes, aumentando a cobertura do software e, consequentemente, a sua qualidade. A ferramenta também permite que os testadores foquem seus esforços em outros tipos de teste ou em testes que não possam ser automatizados.

O critério Análise de Mutantes dispõe de operadores de mutação tradicionais, os quais permitem a modificação de operadores unários e binários, tais como operadores lógicos e aritméticos. Estes operadores foram adaptados para serem aplicados em programas Java (AMMANN; OFFUTT, 2008).

A Mujava possibilita a criação de mutantes para sistemas em Java de acordo com seus 47 tipos de operadores de mutação que são divididos em dois tipos: operadores cujas mutações são produzidas no nível de método (MA et al., 2005) e operadores cujas mutações são produzidas no nível de classe (MA et al., 2002). O primeiro grupo tem 19 operadores. Já o segundo grupo tem 28 operadores.

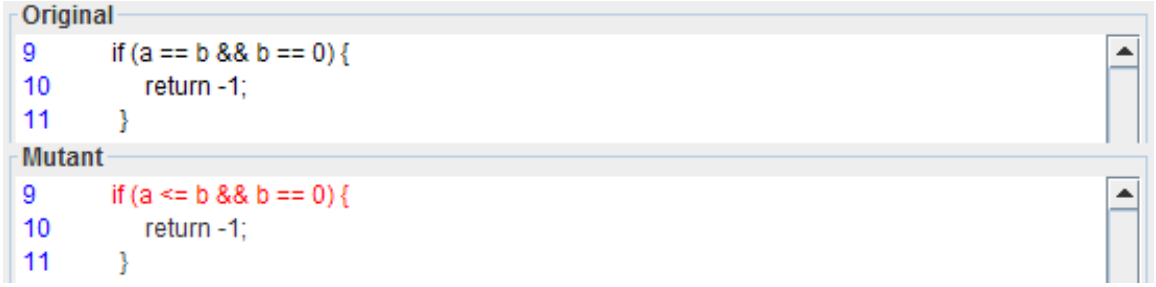
Cada operador de mutação da Mujava tem uma sigla para sua identificação. A primeira letra da sigla determina a característica da linguagem relacionada ao operador. Por exemplo, o operador de nível de método AORB significa "*Arithmetic Operator Replacement (Binary)*" e é um dos operadores do grupo aritmético (A = *Arithmetic*).

O número reduzido de operadores de método implementado pela Mujava é devido à abordagem seletiva adotada para criar tal conjunto de mutantes (MA et al., 2002). Os operadores a nível de método definidos para linguagem java, estão descritos no anexo A deste trabalho.

As categorias dos operadores de método são: *Arithmetic*, com 6 operadores; *Relational*, com 1 operador; *Conditional*, com 3 operadores; *Shift*, com 1 operador; *Logical*, com 3

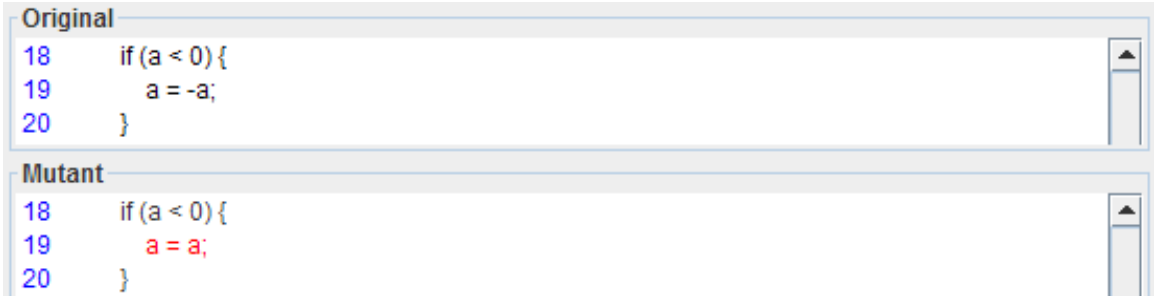
operadores; já as categorias *Assignment*, *Variable*, Constant, *Operator* e *Statement* possuem 1 operador cada uma. Uma amostra da utilização dos operados pode ser vista nas Figuras 2 e 3:

Figura 2 – Interface para criação do ARFF



Fonte – Elaborada pelo autor

Figura 3 – Interface para criação do ARFF



Fonte – Elaborada pelo autor

Os operadores de classe implementados pela Mujava são desenvolvidos com base em recursos de linguagem orientada a objetos e características específicas de Java. As características de orientação a objetos implementadas são: *Inheritance, Polymorphism e Overloading*, com 8, 7 e 3 operadores de mutação por característica, respectivamente. Há ainda mais 10 operadores: 6 operadores de mutação específicos de Java e 4 operadores relacionados aos erros mais comuns de programação (MA et al., 2005).

É importante a definição de casos de teste efetivos, devido ao custo e demora que os mesmos podem provocar no processo de teste. Com a aplicação de técnicas ou critérios de teste que assegurem a qualidade e abrangência dos casos de teste, é possível a realização de forma eficiente do teste de regressão.

4 TESTE DE REGRESSÃO

Segundo Sommerville (2010), a manutenção do sistema é o processo geral de mudança em um sistema depois que ele é liberado para uso. Ainda que um sistema tenha sido submetido a testes rigorosos durante todo o ciclo de desenvolvimento, e que os mesmos não verifiquem que ele possui defeitos para aquele conjunto de casos de teste, tendo o sistema sofrido modificações depois de ser entregue, não se exclui a necessidade da reexecução dos casos de teste, pois as mudanças podem provocar novos defeitos no sistema. Os testes de regressão geralmente são executados após a correção de algum defeito ou após a adição de uma nova funcionalidade, com objetivo de assegurar que nenhum defeito foi adicionado ao sistema após sua modificação.

Tanto as funcionalidades modificadas quanto as que não sofreram modificações devem ser testadas novamente, com a intenção de garantir que as funcionalidades não modificadas não tenham sido afetadas negativamente pela manutenção. Normalmente, este tipo de teste é realizado através de ferramentas de automação, porque um problema encontrado durante esta fase é a falta de tempo para executar novamente casos de teste já executados, logo o teste de regressão, muitas vezes, é deixado em segundo plano por ser considerado muito custoso.

A reexecução dos casos de teste existentes é muito cara. Assim, como alternativa, um subconjunto de casos de teste pode ser selecionado para retestar o programa alterado. A escolha é feita de acordo com uma amostra representativa dos testes que usam todas as funções do software. Entretanto, nem sempre os casos de teste existentes são satisfatórios para avaliar as modificações feitas no software, e novos casos de teste devem ser criados.

Elbaum et al. (2003), apresenta quatro metodologias que reusam o conjunto de testes T da versão original do software P com o intuito de elevar o custo-benefício do teste de regressão.A seguir cada uma dessas metodologias é apresentada:

(i) Retest-all: reutilização de todos os casos de teste do conjunto original T para execução no software modificado. Casos de testes obsoletos devem ser reformulados ou descartados. Essa é a técnica mais utilizada atualmente. Porém, essa técnica pode ser muito cara, já que executar todas as situações de teste exige muito tempo e grande esforço humano.

(ii) Seleção: reusa casos de teste, porém, seletivamente, de acordo com algum

critério, centrando-se sobre subconjuntos do conjunto original T. Uma técnica de seleção de testes de regressão segura deve garantir que os casos de testes que não foram selecionados não revelem falhas na nova versão. Uma forma de garantir a segurança é executar um grande número de casos de testes, mas isso faz com que a técnica ganhe em segurança e perca em eficiência.

(iii) Redução: a redução do conjunto original T é dada eliminando casos de teste redundantes de forma permanente. À medida que o software evolui, novos casos de testes são criados para validar novas funcionalidades. Com isso, alguns testes podem se tornar repetidos. A metodologia de redução aumenta a eficiência do conjunto de testes eliminando os testes redundantes.

(iv) Priorização: estabelece uma ordenação dos casos de testes de T de modo que aqueles com prioridade mais elevada, de acordo com algum critério, sejam executados mais cedo no ciclo de teste de regressão do que os casos de testes com baixa prioridade.

Na literatura, muitos trabalhos utilizam Técnicas de AM para apoiar as metodologias de teste de regressão conforme apresentado na seção de trabalhos relacionados. Essas técnicas são utilizadas para encontrar relacionamentos entre os conjuntos de dados oriundos das atividades de teste de software. Têm-se como exemplos desses dados: o domínio de entrada do programa, os caminhos percorridos pelos casos de teste, defeitos revelados pelos casos de teste, cobertura dos operadores de mutação, entre outros. Esses relacionamentos são utilizados para seleção, priorização e redução do conjunto de dados de teste para o teste de regressão.

4.1 REDUÇÃO DO CONJUNTOS DE CASOS DE TESTE

A redução do Conjunto de Caso de Teste requer a exclusão de parte do conjunto dos casos de teste de modo permanente, conseguindo um conjunto menor e conservando a propriedade de completude do conjunto (ROTHERMEL et al., 2004). Os conjuntos de teste gerados pelos métodos clássicos comumente apresentam redundâncias, ou seja, na medida em que o software evolui, os novos casos de teste criados podem gerar redundância no conjunto original.

Após atualizações, inclusão de funcionalidades ou correção de defeitos, de P (programa) novos Casos de Teste (CT) precisam ser adicionados a G(grupo ou conjunto de casos de teste) para validar novas funcionalidades. Técnicas de redução do conjunto de caso de teste

procuram usar informações em P e G para permanentemente remover CT redundantes em G, deixando subsequentes utilizações de G mais eficientes (ROTHERMEL et al., 2004). Redução do conjunto de casos de teste proporcionam a remoção de CT de G de modo permanente, também comumente utilizadas independentemente de P.

Com a redução do tamanho do conjunto de casos de teste, as futuras versões do software se beneficiam das técnicas de redução de testes no que diz respeito aos custos de execução, validação e gerenciamento de conjuntos de teste. A redução do conjunto de testes pode produzir economias substanciais com pouco custo para a eficácia da detecção de falhas.

4.2 TRABALHOS RELACIONADOS

Trabalhos empíricos e teóricos revelam que é de fundamental importância o desenvolvimento de abordagens que viabilizem a aplicação prática do teste de mutação, devido a sua eficácia em revelar a presença de defeitos.

Lenz (2010) implementa a ferramenta RITA (*Relating Informationfrom Testing Activity*) para teste na linguagem C, aplicando técnicas de Aprendizado de Máquina (AM), para relacionar informações como, por exemplo: dados de entrada, saída produzida e elementos cobertos por critérios de teste estrutural, funcional e baseado em erros. A partir das informações os dados são agrupados em classes funcionais. Os resultados obtidos são submetidos a um algoritmo de classificação, para geração de regras que são utilizadas na redução dos casos de teste, para reduzir os esforços e aumentar a eficácia dos testes de regressão.

Santos (2016) propõem a redução de casos de teste na linguagem Java para o teste de regressão, com a aplicação de técnicas de AM. Combinando informações geradas a partir da execução e análise de dados coletados das técnicas de teste estruturais, funcional e baseado em erros. E aplica sobre estas informações o agrupamento em classes funcionais. Os resultados gerados são então submetidos a um algoritmo de classificação, para geração de regras a serem utilizadas na seleção, priorização e redução de dados de teste.

Mateo e Uaola (2012) propõem a redução do custo de execução dos testes de mutação a partir da melhoria da técnica, reduzindo o número de execuções necessárias e identificando possíveis laços infinitos em um conjunto de casos de teste utilizando a clustering (agrupamento). Permitindo reduzir o custo do teste sem comprometer sua qualidade.

Hanman e Yoo (2009) aplicam a técnica *Pair-Wise* (Matrizes Ortogonais) que consiste em testar um software sem ter que utilizar 100% das possibilidades de variações existentes a fim de dar prioridade a casos de teste, combinada com a priorização de caso de teste baseado na técnica de clustering (agrupamento). Agrupando casos de teste, com base na sua dinâmica e comportamento de tempo de execução, para reduzir o número necessário de comparações e assim reduzir o custo de execução dos testes.

M.Kartheek e Rajshekhar (2013) propõem um método para extrair comportamentos dos conjuntos de casos de teste e agrupá-los utilizando mineração de dados. Com o objetivo de encontrar redundâncias em conjuntos de casos de testes. Para isso, utilizam a ferramenta WEKA e escolhem como melhor algoritmo de clusterização para identificar *clusters* o *K-Means*, e a partir da execução conseguem agrupar os casos de testes com o mesmo comportamento e avaliam a cobertura do código obtendo bons resultados.

Na na Tabela 1 é mostrado um resumo comparativo das pesquisas mais próximos do trabalho desenvolvido.

Tabela 1 – Resumo Comparativo de Trabalhos Relacionados

Característica	Classificação	Lenz (2010)	Santos (2016)	Pesquisa
Linguagem	C	X		
	Java		X	X
Agrupamento	Hierárquico		X	X
	K-Means	X	X	X
	EM	X	X	X
	COBWEB	X	X	
	DBSCAN		X	
Classificação	J48	X	X	X
Dados Analisados	Técnica Estrutural	X	X	
	Análise de Mutantes	X	X	X

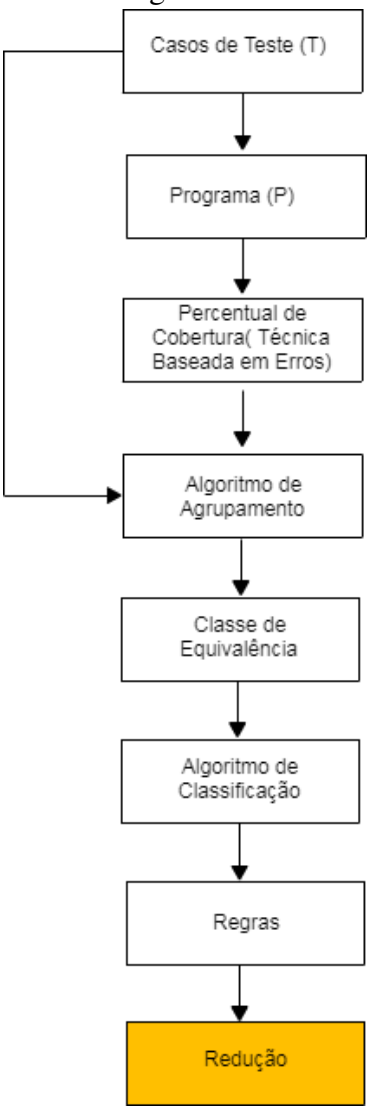
5 MÉTODO DE REDUÇÃO DE CONJUNTOS DE CASOS DE TESTE

O presente trabalho é parte do projeto de pesquisa AI+RTESTING - Uma Metodologia para Otimização do Testes de Regressão Aplicando Técnicas de Inteligência Artificial, a partir do desenvolvimento de uma ferramenta que automatize a metodologia proposta em (LENZ, 2010) para linguagem Java. A ferramenta tem como objetivo a aplicação de técnicas de aprendizado de máquina para apoiar o teste de regressão na seleção, priorização e redução do conjunto de casos de teste, combinando os critérios das três principais técnicas de teste.

Em particular, este trabalho limita-se à redução do conjunto de casos de testes diminuindo assim o tempo de execução da atividade de Teste de Mutação do critério Análise de Mutantes, apoiado por técnicas de aprendizado de máquina com o propósito de identificar classes de equivalência de um dado conjunto de casos de teste T e, a partir delas, gerar regras, que são interpretadas e armazenadas em uma estrutura de dados.

A Figura 4, apresenta o fluxograma da metodologia que foi adaptada de (LENZ, 2010), na qual foi inserida a etapa de redução e exclui-se as etapas de seleção e priorização, bem como os elementos de cobertura da Técnica Estrutural.

Figura 4 – Fluxograma AI+RTESTING



Fonte – Adaptado de (LENZ, 2010)

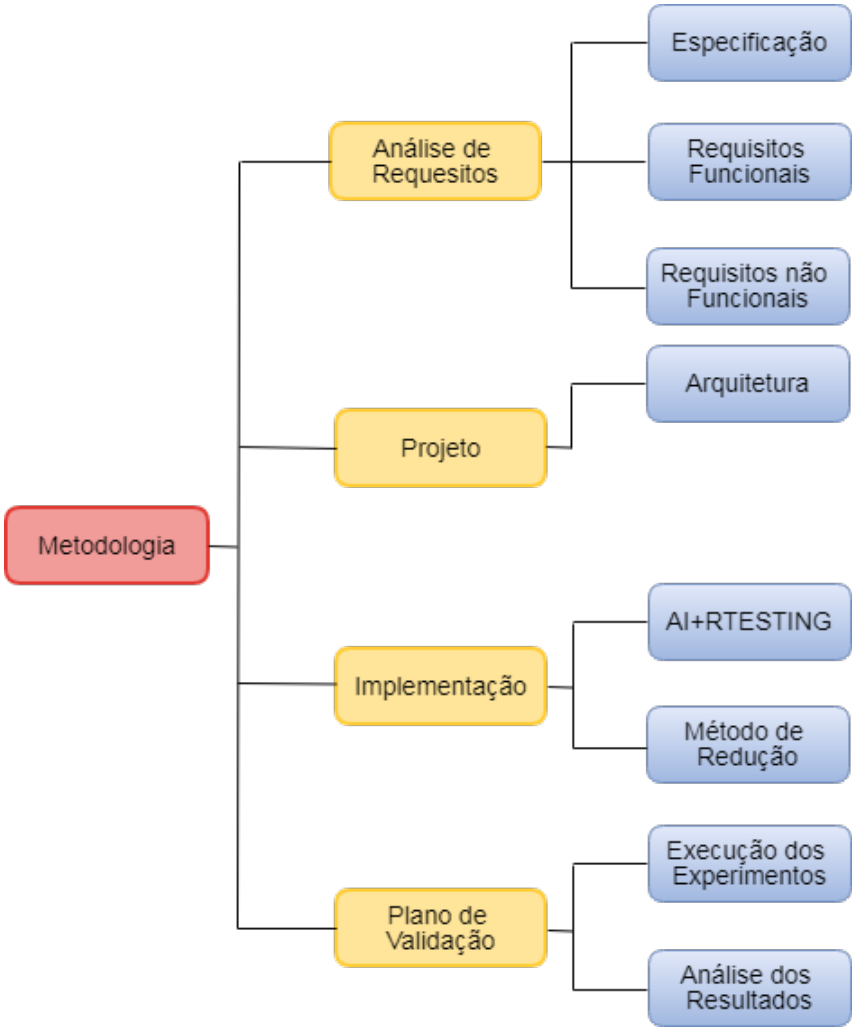
Um conjunto de casos de teste T é executado na ferramenta criada para apoio ao teste baseado em erros, retornando o número de mutantes vivos, número de mutantes mortos e o escore de mutação de T, para cada operador de mutação é calculado o percentual de cobertura do caso de teste da seguinte forma na equação abaixo (SANTOS, 2016):

$$P.Cobertura = \frac{TotalMutantesMortosDoOperador}{TotalMutantesDoOperador} \tag{5.1}$$

Com base nos resultados coletados durante a execução da ferramenta, é gerado um único arquivo com o conjunto de dados, sobre os quais os algoritmos de agrupamento são aplicados. De acordo com Lenz (2010), é possível identificar as classes de equivalência do

programa em teste P, e os casos de teste associados a cada classe. O agrupamento que é gerado produz um atributo nominal, utilizado como atributo de entrada de um algoritmo de classificação. O algoritmo de classificação gerar regras como saída, as quais são interpretadas para posteriormente ser aplicado o método que realiza a redução dos casos de teste. A metodologia de desenvolvimento do trabalho pode ser visualizada na Figura 5.

Figura 5 – Metodologia de Desenvolvimento.



Fonte – Elaborada pelo autor.

Como observado na Figura 5, o desenvolvimento do método proposta para resolução do problema é composta pelas seguintes etapas:

Análise de Requisitos: Compreensão do domínio da aplicação para projetar a arquitetura da solução; Obter informações sobre os requisitos para a extração e validação dos dados gerados a partir do acionamento dos algoritmos da ferramenta Weka; Análise e entendimento da estrutura da ferramenta Mujava para o desenvolvimento da extensão de funcionalidades a ser

realizada.

Arquitetura e Implementação: necessária para o Especificação da aplicação assim como desenvolvimento do ambiente experimental; Implementação das extensões projetadas para a ferramenta Mujava.

Validação: Para validar a ferramenta desenvolvida é necessário a definição e execução de um conjunto de experimentos conforme os passos a seguir: Pesquisar e selecionar programas na linguagem java para serem usados durante a execução dos testes; Implementar conjuntos de casos de teste para os programas selecionados; Executar conjuntos de casos de teste na ferramenta desenvolvida para cada um dos programas escolhidos; Avaliar o custo do tempo de execução e a eficácia em relação ao escore de mutação do novo conjunto de casos de teste gerado para cada um dos programas definidos.

Análise dos Resultados: Comparativo entre o conjunto de casos de teste reduzido em relação à abordagem *retest all*.

5.1 ANÁLISE DE REQUISITOS

A partir do levantamento do conjunto de necessidades estabelecidas pode-se identificar as diferentes funcionalidades que se deseja que o sistema realize, definindo a estrutura e o comportamento do software AI+RTESTING. Essas funcionalidades foram descritas em requisitos funcionais e não funcionais identificados no projeto AI+RTESTING, como podem ser observados na Tabela 2 e na Tabela 3, respectivamente:

Tabela 2 – Requisitos Funcionais.

RF01 – Integrar a ferramenta Mujava a ferramenta Jabuti.
RF02 – Desenvolver rotinas para extrair e armazenar os dados da execução das Ferramentas Jabuti e Mujava.
RF03 – Desenvolver rotina para escrever no arquivo ARFF os resultados da execução das Ferramentas Jabuti e Mujava.
RF04 –Integrar a ferramenta Weka a Jabuti através da biblioteca .jar.
RF05 – Criar rotinas para acionar os algoritmos de agrupamento.
RF06 – Criar rotinas para escrever no arquivo ARFF os resultados gerados a partir da execução dos algoritmos de agrupamento.
RF07 – Desenvolver rotinas para acionar o algoritmo de classificação J48.
RF08 – Desenvolver rotinas para extrair e armazenar os resultados da aplicação do algoritmo de classificação J48.
RF09 – Desenvolver rotinas para armazenar as regras de classificação geradas pelo J48 em uma estrutura de dados (árvore).
RF10 – Desenvolver rotina para redução dos casos de teste, aplicando Progressão Geométrica.
RF11 – Desenvolver interface gráfica para acionar o método de redução de casos de teste.

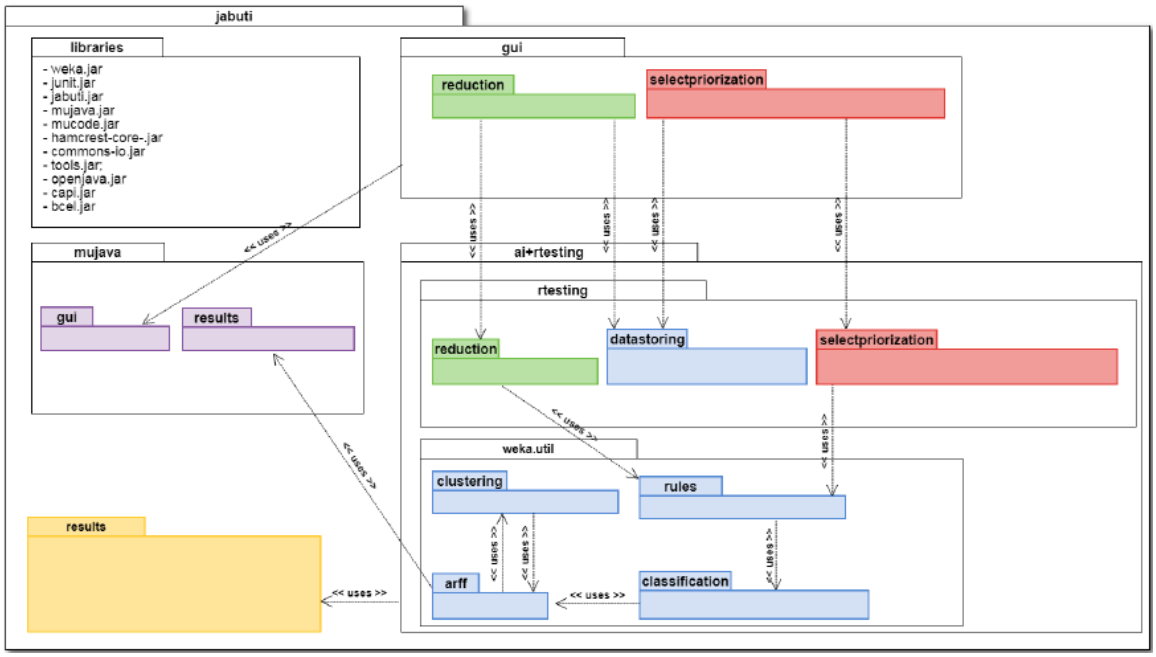
Tabela 3 – Requisitos Não Funcionais.

RNF01 – Manter a modularidade das ferramentas.
RNF02 – Documentar os códigos desenvolvidos para uma possível manutenção da ferramenta.
RNF03 – Utilizar padrões de desenvolvimento.

5.2 PROJETO E IMPLEMENTAÇÃO

A execução do Projeto é realizada com base no diagrama de pacotes do projeto AI+RTESTING, definido na fase anterior ao projeto, o diagrama pode ser observado na Figura 6.

Figura 6 – Diagrama de Pacotes AI+RTESTING



Fonte – (CRISTO, 2017)

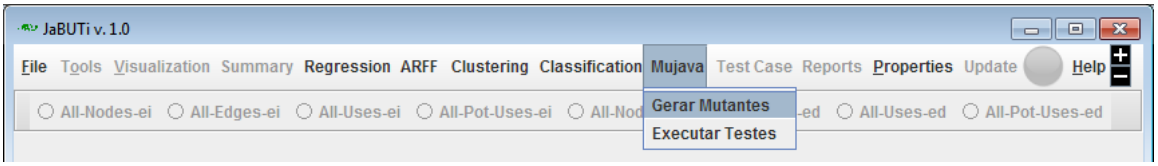
O projeto AI+RTESTING foi organizado para refletir as classes que foram modificadas, adicionadas e criadas dentro do projeto, para agrupar classes de funcionalidades similares ou relacionadas, os pacotes descritos na sequência se restringem a metodologia de redução proposta na pesquisa. Para integração da ferramenta Mujava, todos os pacotes da mesma foram adicionados ao pacote mais externo, o pacote Jabuti. No pacote jabuti.result encontra-se a classe que organiza os dados coletados no teste estrutural e no pacote mujava.result a classe que organiza os dados coletados no teste de mutação. No pacote weka.util foi criado os pacotes weka.util.clustering que agrupa as classe que executam os algoritmos de agrupamento, weka.util.classification com a classes que realiza a chamada do algoritmo de classificação, weka.util.rule com as classes para tratamento das regras de classificação e weka.util.arff com a classe que gera um arquivo de saída com os dados coletados no formato ARFF. O pacote ai+rtesting representa o desenvolvimento das três metodologias do teste de regressão, agrupando suas respectivas classes. O método de redução desenvolvido está representado no pacote reduction. No pacote mujava.gui foram modificadas as classes que permitiram a extração de dados do critério Análise de Mutantes e no pacote mujava.result encontra-se a classe que organiza os dados coletados no teste de Análise de Mutantes.

5.2.1 Integração das Ferramentas Mujava e Jabuti

A integração das ferramentas de teste estrutural Jabuti com a ferramenta da técnica baseada em erros Mujava, deu origem à nova ferramenta desenvolvida no projeto AI+RTESTING, com o intuito de prover funcionalidades que permitam aos usuários a realização de teste de regressão com base nos métodos de redução, seleção e priorização de casos de teste. Este trabalho contempla a funcionalidade que permite utilizar o método de redução do conjunto de casos de teste, com a utilização da técnica baseada em erros.

Um conjunto de casos de teste definido para testar um programa é executado na ferramenta, a qual retorna o escore de mutação e para cada operador de mutação o percentual de cobertura associado por caso de teste. A integração da Mujava à Jabuti pode ser observada na Figura 7.

Figura 7 – Item de menu Mujava.



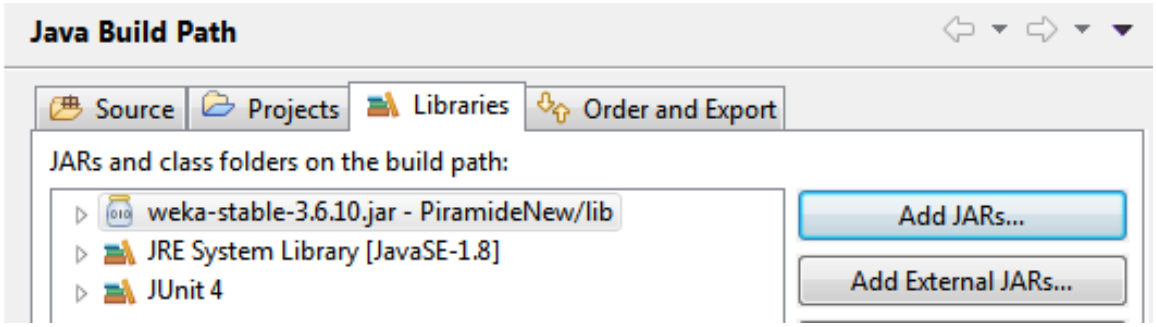
Fonte – Elaborada pelo autor

Os casos de teste são armazenados com as informações coletadas a partir do teste de mutação. Em seguida, é possível gerar o um arquivo ARFF de saída, que é utilizado como entrada para posterior execução dos algoritmos de agrupamento, os quais também geram como saída um arquivo ARFF a ser usado como dados de entrada na execução do algoritmo de classificação.

5.2.2 Integração da Ferramenta Weka

A API Weka para código Java permite aplicar mineração de dados a partir do emprego de algoritmos sofisticados para analisar grandes bases de dados, procurando extrair das mesmas informações que estejam implícitas. A API Weka foi inserida à ferramenta desenvolvida, com a adição do weka.jar como uma biblioteca do projeto, como pode ser visto na Figura 8, possibilitando dessa forma a utilização dos algoritmos de agrupamento (K-Means, Agrupamento Hierárquico e EM) além do algoritmo de classificação (J48).

Figura 8 – Adição da API Weka



Fonte – Elaborado pelo autor

Para carregar dados no WEKA, é necessário colocá-los no formato ARFF (Formato de Arquivo de Atributo-Relação), onde é possível definir o tipo e os dados que estão sendo carregados. No arquivo, definimos cada coluna e o que cada coluna contém. O arquivo inicial do projeto, que servirá de entrada para os algoritmos de agrupamento, está limitado a 23 colunas do tipo *numeric*, que armazenará o percentual de cada operador e uma coluna do tipo *text* com a identificação do respectivo caso de teste.

5.2.3 Implementação das Classes de Agrupamento

Para executar os algoritmos de agrupamento definidos na pesquisa foi necessário implementar as classes *ClusterHierarchical*, *ClusterEM* e *ClusterSimpleKMeans*. As quais realizam as chamadas de cada classe de agrupamento descritas nos Código Fonte 1, 2 e 3. E geram os *clusters* associados aos casos de teste com o percentual de cobertura de cada operador de mutação.

```
1  public  void createClusterEM(String DataTraining, String
       DataResult) {
2    FileReader reader = new FileReader(DataTraining);
3    Instances iData = new Instances(reader);
4    iData = removeAttributes.listIntances(iData);
5    EM clusterer = new EM();
6    clusterer.setNumClusters(4);
7    clusterer.setSeed(18);
8    clusterer.setMaxIterations(100);
9    clusterer.setMinStdDev(1.0E-6);
10 }
```

Código-fonte 1 – Classe *Cluster* EM

Na linha 1 o método recebe como parâmetro o caminho do arquivo ARFF com o conjunto de dados para agrupamento, na linha 4 são removidos os atributos de valor zero, ficando somente os percentuais válidos de cobertura de cada operador, na linha 5 é instanciado um objeto do tipo EM, em seguida nas linhas 6 a 9 são passados os parâmetros de agrupamento desejados, que são respectivamente; número de *cluster*, o *Seed* para geração de números aleatórios, o número máximo de iterações para executar e o valor mínimo para o desvio padrão ao calcular a densidade normal.

```
1  public void createClusterSimpleKMeans(String dataTraining, String
       DataResult){
2    iData = removeAttributes.listIntances( iData);
3    SimpleKMeans model = new SimpleKMeans();
4    model.setNumClusters(4);
5    model.setSeed(20);
6    model.setMaxIterations(100);
7    model.buildClusterer(iData );
8  }
```

Código-fonte 2 – Classe *Cluster* Simple K-Means

Na linha 1 o método recebe como parâmetro o caminho do arquivo ARFF com o conjunto de dados já agrupamento, na linha 2 é removido os atributos de valor zero, ficando somente os percentuais válidos de cobertura de cada operador, na linha 3 é instanciado um objeto do tipo *K-Means*, em seguida nas linhas 4 a 7 são passados os parâmetros de agrupamento desejados, que são respectivamente; número de *clusters*, o *Seed* para geração de números aleatórios e o número máximo de iterações para executar.

```
1 public  void createClusterHierarchical(String dataTraining, String
     dataResult) {
2    iData = removeAttributes.listIntances( iData);
3    HierarchicalClusterer h = new HierarchicalClusterer();
```

```
4     h.setNumClusters(10);
5   }
```

Código-fonte 3 – Classe *Cluster Hierarchical*

Na linha 1 o método recebe como parâmetro o caminho do arquivo ARFF com o conjunto de dados para agrupamento, na linha 2 é removido os atributos de valor zero, ficando somente os percentuais válidos de cobertura de cada operador, na linha 3 é instanciado um objeto do tipo *Hierarchical*, em seguida na linha 4 é passado o parâmetro desejado com número de clusters.

5.2.4 Implementação da Classe *Classifier*J48

Para gerar árvores de classificação foi preciso realizar a chamada do algoritmo J48 da API Weka, a partir da implementação da classe *Classifier*J48. O Código Fonte 4 mostra a implementação do método *callJ48*.

```
1 public void callJ48(String path) throws Exception {
2     FileReader reader = null;
3     int index = returnValueIndex(path) - 3;
4     reader = new FileReader(path);
5     Instances instances = null;
6     instances = new Instances(reader);
7     instances.setClassIndex(index);
8     arvore.buildClassifier(instances);
9   }
```

Código-fonte 4 – Método *call*J48 da classe ClassifierJ48

Na linha 1 o método recebe como parâmetro o caminho do arquivo ARFF com o conjunto de dados de treinamento, Na linha 8 é realizada a chamada ao método *buildClassifier*, passando como entrada a base de dados de treinamento, na linha 7 é utilizado o método *setClassIndex*, que permiti que quando uma base for importada para a memória, cada atributo receba um índice de acordo com a ordem em que foi especificado no arquivo ARFF.

5.2.5 Método de Redução de Casos de Teste

De acordo com Rothermel (2002) as técnicas de redução de conjuntos de casos de teste tentam reduzir o custo da manutenção de software reutilizando casos de teste do conjunto inicial já definido, eliminando casos de teste redundantes. Sendo possível otimizar o teste de regressão, reduzindo os casos de teste e consequentemente os operadores de mutação gerados associados aos casos de teste.

A partir do algoritmo de classificação foram geradas regras, formadas pelos atributos, percentual de mutantes mortos por operador, clusters e quantidade de casos de teste, essa regras devem ser interpretadas, tornando possível identificar quais classes de teste são mais importantes, ou seja, quais geram um percentual maior de cobertura (SANTOS, 2016). Com base no percentual de cobertura de cada operador de mutação, é possível identificar quais casos de teste tem maior percentual de cobertura, e assim, efetuar uma redução apropriada.

A classificação produzida a partir das regras estabelece um padrão que define o modo tal como será organizada a saída de uma nova instância de dados (elementos que compõem a regra). A saída produzida a partir da execução do algoritmo J48 é uma árvore de decisão em que cada nó da árvore corresponde a um estado no qual uma decisão deve ser tomada, sendo necessário a interpretação da árvore para associação dos elementos que fazem parte da regra. Uma nova árvore foi criada, para guardar a regra, partindo do nó raiz onde é inserido o primeiro atributo da nova arvore, e com a interpretação dos operadores é possível identificar quais atributos seguintes devem ser inseridos movendo-se para os próximos nós à esquerda ou à direita do nó raiz, até alcançar uma folha, que mostre a saída da árvore com os clusters ligados ao número de casos de teste e ordem de prioridade das classes de equivalência.

Lenz (2010), propôs uma abordagem baseada em Progressão Geométrica (PG) para que fosse selecionada a quantidade de casos de testes que devem ser executados de cada classe gerada a partir da interpretação da regra. O método proposto define de forma equilibrada o número de casos que devem ser mantidos ou removidos.

Para iniciar o processo de redução o testador deve informar o fator de redução, para determinar a quantidade dos casos de teste de maior prioridade que ele deseja executar, esse valor é utilizado para definir o enésimo termo da PG a ser calculado, que define a quantidade de casos de teste que deve ser selecionada para redução.

Cada elemento da PG corresponde à porcentagem de casos de teste que deve ser selecionada para cada classe de equivalência. A fórmula do termo geral da PG é dada por:

$$a_n = a_k * q^{n-1} \tag{5.2}$$

. Onde:

- **k** Termo qualquer da PG, define o primeiro termo da PG;
- **n** quantidade de elementos da PG.
- **q** intervalo padrão, conhecido como fator multiplicativo ou razão

Continuando com PG. O cálculo da razão será realizado da seguinte forma:

$$q = \frac{a_n}{a_{n-1}} \tag{5.3}$$

De acordo com a PG genérica: (a_1, a_2, a_3, a_4, ... , a_n, ...) , na qual a_1 é o primeiro termo, e a_n é o n-ésimo termo, isto é, o termo de ordem n. Sendo q a razão da PG, da definição podemos escrever:

$$a_2 = a_1.q \tag{5.4}$$

$$a_3 = a_2.q = (a_1.q).q = a_1.q_2 \tag{5.5}$$

Portanto a abordagem da PG pode ser dada, como se segue:

1. Define-se o valor do primeiro termo como a divisão de 1(um) pelo total de casos de teste do cluster de menor um prioridade.
2. Define-se o valor do enésimo termo a partir do fator de redução passado pelo usuário.
3. Calcula-se a razão através das propriedades da PG.
4. A partir do valor da razão e da fórmula que calcula um elemento da PG, obtem-se os valores dos outros elementos contidos na PG..

O método de redução foi implementado com base no diagrama mostrado na Figura 9.

Figura 9 – Diagrama de classe do método de redução

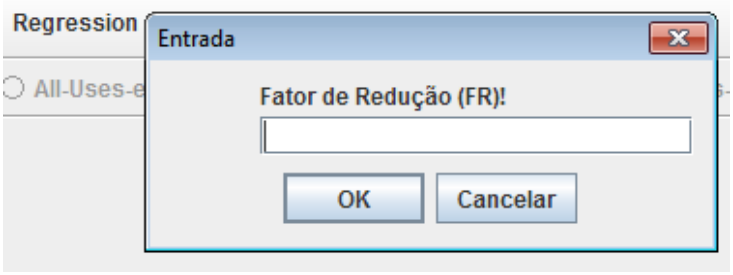
Fonte – Elaborada pelo autor

A classe *Reduction* liga-se a classe *GeometricProgression* por uma associação unidirecional, com valor de multiplicidade zero ou um, bem como a classe *ReductionResults*, pela mesma associação mas com valor de multiplicidade zero ou muitos. Na classe *Reduction* foi definido o método *reducer*() para calcular o número de casos de teste a serem selecionados para redução por cada classe de equivalência definida pela regra gerada anteriormente, cálculo esse que se baseia na progressão gerada, a partir do método *setProgression(Double, Double, Double)* da classe *GeometricProgression*. O resultado da redução é então passado para a classe *ReductionResults*.

Na execução do método de redução o testador deve indicar através da interface

gráfica o fator de redução desejado como apresentada na Figura 10

Figura 10 – Interface de entrada do FR



Fonte – Elaborada pelo autor

As interfaces desenvolvidas no projeto AI+RTESTING para o método de redução, podem ser visualizadas no apêndice A.

No capítulo seguinte é apresenta avaliação da abordagem proposta, através da condução de experimentos e demonstração da análise dos resultados.

6 EXPERIMENTOS E RESULTADOS

A realização dos experimentos e validação dos novos conjuntos de casos de teste, foram analisados por meio da comparação com a abordagem *retest all*, tomando como parâmetros o tempo e porcentagem de cobertura dos critérios encontrados na execução dos testes.

A partir da execução dos experimentos, foi possível atestar a identificação das classes de equivalência, com a aplicação dos algoritmos de agrupamentos e gerar o conjunto de treinamento, utilizado no processo de classificação para produzir o modelo, de onde se infere as regras, que sustentam a aplicação da redução.

6.1 CONDUÇÃO DOS EXPERIMENTOS

A metodologia apresentada foi aplicada à linguagem Java, apoiada pela ferramenta AI+RTESTING para teste unitário de programas em nível de método. E empregou a técnica de teste baseado em erros. Em conjunto com a API WEKA, para utilização dos algoritmos de agrupamento EM, Hierárquico e K-Means e o algoritmo de classificação J48.

Na seção seguinte são apresentados os dois algoritmos selecionados MDC e Pirâmide, escritos na linguagem Java para execução dos experimentos, bem como os respectivos resultados da aplicação da metodologia.

6.2 EXECUÇÃO DOS EXPERIMENTOS

Conforme mencionado anteriormente, foram utilizados dois algoritmos implementados na linguagem java para realização dos experimentos: O algoritmo MDC define um método para calcular o máximo divisor comum entre dois ou mais números naturais, recebendo como argumento dois parâmetros de tipo inteiro como entrada e calcula o MDC entre os dois números. O algoritmo Pirâmide define um método que recebe como argumento dois parâmetros de tipo inteiro como entrada, e verifica se é possível formar uma pirâmide triangular regular. Uma pirâmide triangular regular é formada por 4 faces, sendo uma a base e três faces laterais, todas triângulos equiláteros. A implementação dos métodos são apresentados no apêndice B.

6.2.1 Geração de Casos de Teste

Com o intúido de realizar os testes como método MDC, inicialmente foi criada a classe MDCTest (conjunto original) com 26 casos de teste criados, no formato JUnit, a partir da mesma, foi possível derivar a classe MDCRedundanTest a qual ficou com 101 casos de teste redundantes. Para o algoritmo Pirâmide foi criada a classe de teste PiramideTest com 30 casos de teste, no formato JUnit, da qual derivou a classe PiramideRedundanTest com 104 casos de teste redundantes. Nos Códigos Fonte 5 e 6 é possível visualizar alguns casos de teste criados para os métodos MDC E Pirâmide respectivamente.

```
1  import org.junit.Test;
2  import static org.junit.Assert.*;
3  public class MDCTest {
4      private static int VALOR_MINIMO = -2147483640;
5    private static int vmin_a = VALOR_MINIMO;
6    private static int vmin_b = VALOR_MINIMO;
7    private static int VALOR_MAXIMO = 2147483640;
8    @Test
9    public void mdc_0_0() {
10     int a = 0;
11     int b = 0;
12     int valorEsperado = -1;
13     int valorAlcancado = MDC.calcularMDC(a, b);
14     assertEquals(valorEsperado, valorAlcancado);
15   }
16   /* matando mutante ror2 */
17   @Test
18   public void mdc_1_0() {
19     int a = 1;
20     int b = 0;
21     int valorEsperado = 1;
22     int valorAlcancado = MDC.calcularMDC(a, b);
23     assertEquals(valorEsperado, valorAlcancado);
24   }
```

Código-fonte 5 – Classe MDCTest

```
1  import org.junit.Test;
2  import static org.junit.Assert.*;
3  public class PiramideTest {
4    // Vari veis formam uma pir mide tri ngular regular 1
5    @Test
6    public void piramideTetraedroRegular1() {
7      assertEquals(3, Piramide.piramide(1, 1, 1, 1, 1, 1));
8    }
9    // Vari veis formam uma pir mide tri ngular regular 2
10   @Test
11   public void piramideTetraedroRegular2() {
12     assertEquals(3, Piramide.piramide(2, 2, 2, 2, 2, 2));
13   }
14   // Vari veis formam uma pir mide tri ngular regular 3
15   @Test
16   public void piramideTetraedroRegular14() {
17     assertEquals(3, Piramide.piramide(14, 14, 14, 14, 14, 14));
18   }
```

Código-fonte 6 – PiramideTest

Os casos de teste criados foram desenvolvidos respeitando as classes de equivalência definidas para os métodos em teste. A Tabela 4 apresenta as classes de equivalência definidas manualmente e sua respectiva quantidade de casos de teste do método MDC e a Tabela 5 as classes de equivalência do método Pirâmide.

Tabela 4 – Método MDC: Classes de equivalência geradas manualmente

Classe Equivalência	Descrição	Caso de Teste
0	A igual a B	1
1	A zero e B zero	1
2	A zero e B negativo	3
3	A negativo e B zero	7
4	A zero e B positivo	3
5	A positivo e B zero	3
6	A negativo e B negativo	2
7	A negativo e B positivo	3
8	A positivo e B negativo	1
9	A positivo e B positivo	2
Total		26

Tabela 5 – Método Pirâmide: Classes de equivalência geradas manualmente

Classe de Equivalência	Descrição	Caso de Teste
0	Variáveis fora dos limites máximo	6
1	Variáveis fora dos limites mínimo	6
2	Valores da base e faces em ordem crescente	7
3	Valores da base e faces não formam triângulos	2
4	Valores da base e faces não formam triângulos equiláteros	5
5	Variáveis formam uma pirâmide triangular regular	4
Total		30

6.3 EXECUÇÃO DOS TESTES

Considerando a metodologia proposta, inicialmente foi necessário submeter as classes de teste para execução com o JUnit, em seguida os mesmos casos de teste foram executados na ferramenta Mujava, com a finalidade de verificar o percentual de cobertura dos operadores de mutação. Pequenas amostras dos resultados de execução para os métodos MDC e Pirâmide são apresentadas na Tabela 6 e Tabela 7, respectivamente.

Tabela 6 – Amostra de cobertura do teste baseado em erros do método MDC

Caso de Teste	Cluster	COR	LOI	ROR	ODL
A igual a B	8	33%	40%	41%	33%
A zero B negativo	0	0%	41%	39%	33%
A negativo B negativo	5	33%	41%	40%	40%
A negativo B positivo	3	0%	47%	39%	26%
A zero e B positivo	2	15%	16%	16%	17%

Tabela 7 – Amostra de cobertura do teste baseado em erros do método Pirâmide

Caso de Teste	Cluster	COR	LOI	ROR	ODL
Variáveis fora dos limites máximos	2	35%	4%	2%	14%
Base e face em ordem crescente	1	32%	27%	35%	6%
Base e face não formam triângulo equiláteros	4	38%	60%	54%	10%
Base e face em ordem crescente	5	3%2	4%	2%	10%
Variáveis fora dos limites máximos	0	13%	9%	29%	4%

A partir dos dados coletados foi possível escrever o arquivo ARFF. Que posteriormente é submetido aos algoritmos de agrupamento para definição de classes de equivalência do conjunto de casos teste. O ARFF foi definido com os seguintes atributos:

- Nome identificador dos casos de teste.
- Percentuais de cobertura dos operadores de mutação: AORB, AORS, AOIU, AOIS,AODU, AODS, ROR, COR, COD, COI, SOR, LOR, LOI, LOD, ASRS, SDL, VDL,CDL, ODL.

Nas Figuras 11 e 12 são apresentadas a definição dos atributos e a amostra da representação dos dados a serem averiguados pelos algoritmos de agrupamento, respectivamente.

Figura 11 – Amostra dos atributos contidos no arquivo Arff.

```
1  @relation AI+RTesting
2  @attribute 'Casos de teste' { }
3  @attribute AOIS numeric
4  @attribute AODU numeric
5  @attribute AOIU numeric
6  @attribute AORB numeric
7  @attribute VDL numeric
8  @attribute SDL numeric
9  @attribute COI numeric
10 @attribute COR numeric
11 @attribute LOI numeric
12 @attribute ROR numeric
13 @attribute AORS numeric
14 @attribute AODS numeric
15 @attribute COD numeric
16 @attribute SOR numeric
17 @attribute LOD numeric
18 @attribute CDL numeric
19 @attribute LOR numeric
20 @attribute ASRS numeric
21 @attribute ODL numeric
```

Fonte – Elaborado pelo autor

Figura 12 – Amostra dos dados contidos no arquivo Arff.

```
@data
base1maiorQue15,1,11,100,0,0,13,16,35,4,2,14
base1maiorQue15_01,1,11,100,0,0,13,16,35,4,2,14
base1maiorQue15_07,1,11,100,0,0,13,16,35,4,2,14
base1menorQue1,0,11,100,0,0,13,16,35,2,2,14
base1menorQue1_01,0,11,100,0,0,13,16,35,2,2,14
base1menorQue1_07,0,11,100,0,0,13,16,35,2,2,14
base2maiorQue15,1,11,100,0,0,13,16,38,4,2,14
base2maiorQue15_02,1,11,100,0,0,13,16,38,4,2,14
base2maiorQue15_08,1,11,100,0,0,13,16,38,4,2,14
baseNaoFormaTriangulo1,8,11,100,12,0,24,67,32,25,35,2
baseNaoFormaTriangulo1_01,8,11,100,12,0,24,67,32,25,35,2
baseNaoFormaTriangulo1_03,8,11,100,12,0,24,67,32,25,35,2
baseNaoFormaTriangulo1_05,8,11,100,12,0,24,67,32,25,35,2
baseNaoFormaTrianguloEquilatero1,0,11,100,37,50,27,69,32,27,35,6
baseNaoFormaTrianguloEquilatero1_01,0,11,100,37,50,27,69,32,27,35,6
baseNaoFormaTrianguloEquilatero1_06,0,11,100,37,50,27,69,32,27,35,6
baseNaoFormaTrianguloEquilatero2,5,11,100,37,50,27,72,38,27,37,8
baseNaoFormaTrianguloEquilatero2_02,5,11,100,37,50,27,72,38,27,37,8
baseNaoFormaTrianguloEquilatero2_07,5,11,100,37,50,27,72,38,27,37,8
baseCrescente1_08,5,11,100,0,0,13,60,38,18,28,6
baseCrescente2,2,11,100,0,0,13,55,35,16,26,2
baseCrescente2_02,2,11,100,0,0,13,55,35,16,26,2
baseCrescente2_09,2,11,100,0,0,13,55,35,16,26,2
baseCrescente3,8,11,100,0,0,13,58,35,18,28,4
baseCrescente3_03,8,11,100,0,0,13,58,35,18,28,4
baseCrescente3_10,8,11,100,0,0,13,58,35,18,28,4
baseNaoFormaTriangulo1,8,11,100,12,0,24,67,32,25,35,2
baseNaoFormaTriangulo1_01,8,11,100,12,0,24,67,32,25,35,2
baseNaoFormaTriangulo1_03,8,11,100,12,0,24,67,32,25,35,2
baseNaoFormaTriangulo1_05,8,11,100,12,0,24,67,32,25,35,2
```

Fonte – Elaborado pelo autor

6.3.1 Combinações de Atributos e Parâmetros de Configuração dos Algoritmos de Agrupamento

Para execução dos experimentos foram considerados os atributos de porcentagem de cobertura dos operadores de mutação.

Na execução dos algoritmos de agrupamento foi necessária a definição de parâmetros de configuração, como mostrado na Tabela 8. Santos (2016) alcançou resultados satisfatórios, na maioria dos experimentos, em relação aos *clusters* gerados pelos algoritmos e as classes de equivalência manuais encontradas. Portanto, nos experimentos realizados fez-se uso desses parâmetros, definidos e validados. Em cada experimento foi considerada a quantidade de classes de equivalência identificadas e que geraram casos de teste para estabelecer o parâmetro do número de *clusters*.

Os algoritmos de agrupamento e os métodos em teste receberam a seguinte numera-

ção:

1. MDC.
2. Pirâmide

1. Agrupamento Hierárquico.
2. EM.
3. K-means.

Tabela 8 – Valores dos parâmetros dos algoritmos de agrupamento.

Experimento	Clusters	MaxIterations	MinStdDev	Seed
1.1	10			
1.2	10	100	1.0E-6	18
1.3	10	100		20
2.1	6			
2.2	6	100	1.0E-6	18
2.3	6	100		20

6.4 IDENTIFICAÇÃO DAS CLASSES DE EQUIVALÊNCIA

Com os experimentos realizados foi possível obter as classes de equivalência geradas através dos algoritmos de agrupamento (EM, Kmeans e Hierarquical) e organizados nas Tabelas 9, 10, 11, 12, 13 e 14. A seguir são apresentados os resultados relacionando as classes geradas com a quantidade de casos de teste de cada cluster encontrado para os métodos MDC e Pirâmide, respectivamente.

Tabela 9 – Experimento 1.1: Classes de equivalência geradas pelo algoritmo hierárquico para o método MDC

Classe de Equivalência	Casos de Teste
0	8
1	4
2	2
3	7
4	2
5	3
6	3
7	1
8	1
9	1
Total	32

Tabela 10 – Experimento 1.2: Classes de equivalência geradas pelo algoritmo EM para o método MDC

Classe de Equivalência	Casos de Teste
0	8
1	2
2	2
3	2
4	4
5	1
6	1
7	1
8	1
9	1
Total	23

Tabela 11 – Experimento 1.3: Classes de equivalência geradas pelo algoritmo K-Means para o método MDC

Classe de Equivalência	Casos de Teste
0	8
1	4
2	9
3	2
4	4
5	4
6	1
7	1
8	1
9	1
Total	35

Tabela 12 – Experimento 2.1: Classes de equivalência geradas pelo algoritmo hierárquico para o método pirâmide

Classe de Equivalência	Casos de Teste
0	4
1	3
2	4
3	2
4	1
5	1
Total	15

Tabela 13 – Experimento 2.2: Classes de equivalência geradas pelo algoritmo EM para o método pirâmide

Classe de Equivalência	Casos de Teste
0	8
1	7
2	3
3	2
4	2
5	1
Total	23

Tabela 14 – Experimento 2.3: Classes de equivalência geradas pelo algoritmo K-Means para o método pirâmide

Classe de Equivalência	Casos de Teste
0	8
1	5
2	4
3	2
4	2
5	1
Total	22

6.5 CLASSIFICAÇÃO

Definidas as classes de equivalência, é possível obter as regras necessárias para realizar a redução através da execução do algoritmo classificador J48, com a interpretação da árvore de decisão gerada. Nas Figuras 13, 14, 15, 16, 17 e 18, podemos visualizar o modelo com a ordenação, para estabelecer a saída de uma nova instância de dados (regra). Cada nó na árvore representa um ponto onde uma decisão deve ser tomada com base na entrada. Possibilitando definir para cada caso de teste os operadores de mutação de maior prioridade a partir da análise do valor percentual de cobertura do operador.

Figura 13 – Experimento 1.1: Regras geradas pelo algoritmo hierárquico para o método MDC

```
AODU <= 0
|   COI <= 75
|   |   AOIU <= 50
|   |   |   COR <= 16: cluster3 (9.0)
|   |   |   COR > 16: cluster0 (4.0)
|   |   AOIU > 50: cluster6 (8.0)
|   COI > 75: cluster7 (16.0)
AODU > 0
|   COR <= 16
|   |   COR <= 0
|   |   |   AOIU <= 50: cluster1 (12.0)
|   |   |   AOIU > 50: cluster9 (4.0)
|   |   COR > 0
|   |   |   COI <= 16: cluster5 (20.0)
|   |   |   COI > 16: cluster8 (4.0)
|   COR > 16
|   |   AOIS <= 48: cluster2 (8.0)
|   |   AOIS > 48: cluster4 (16.0)
```

Fonte – Elaborada pelo autor

O experimento 1.1 foi realizado com o algoritmo Hierárquico combinado com algoritmo MDC, a Figura 13 demostra que a regra foi definida com base em cinco atributos. AODU (exclui operadores aritméticos unários básicos) ,COI (insere operadores condicionais

unários), COR (substitui operadores condicionais binários por outros operadores condicionais), AOIU (insere um operador aritmético unário básico) e AOIS (insere um operador aritmético de atalho).

Figura 14 – Experimento 1.2: Regras geradas pelo algoritmo EM para o método MDC

```
COR <= 16
|   COR <= 0
|   |   AOIU <= 50: cluster1 (21.0)
|   |   AOIU > 50
|   |   |   AOIS <= 48
|   |   |   |   AODU <= 0: cluster2 (4.0)
|   |   |   |   AODU > 0: cluster1 (4.0)
|   |   |   AOIS > 48: cluster1 (4.0)
|   COR > 0: cluster3 (24.0)
COR > 16
|   AODU <= 0: cluster4 (20.0)
|   AODU > 0
|   |   AOIS <= 48
|   |   |   AOIS <= 24: cluster6 (4.0)
|   |   |   AOIS > 24: cluster0 (4.0)
|   |   AOIS > 48: cluster9 (16.0)
```

Fonte – Elaborada pelo autor

Para o experimento 1.2 foi utilizado o algoritmo EM e o algoritmo MDC a Figura 14 demostra que a regra foi definida com base em quatro operadores de mutação. COR (substitui operadores condicionais binários por outros operadores condicionais), AODU (exclui operadores aritméticos unários básicos), AOIU (insere um operador aritmético unário básico) e AOIS (insere um operador aritmético de atalho).

Figura 15 – Experimento 1.3: Regras geradas pelo algoritmo K-Means para o método MDC

```
AODU <= 0
|   COR <= 16
|   |   AOIU <= 50: cluster3 (9.0)
|   |   AOIU > 50
|   |   |   AOIS <= 48: cluster6 (4.0)
|   |   |   AOIS > 48: cluster1 (4.0)
|   COR > 16
|   |   AOIS <= 48: cluster4 (4.0)
|   |   AOIS > 48: cluster8 (16.0)
AODU > 0
|   COR <= 16
|   |   COR <= 0
|   |   |   AOIU <= 50: cluster0 (12.0)
|   |   |   AOIU > 50: cluster2 (4.0)
|   |   COR > 0: cluster9 (24.0)
|   COR > 16
|   |   AOIS <= 48: cluster5 (8.0)
|   |   AOIS > 48: cluster7 (16.0)
```

Fonte – Elaborada pelo autor

No experimento 1.3 foi empregado o algoritmo K-Means juntamente com o algoritmo MDC, na Figura Figura 15 é possível visualizar a regra definida com base em quatro atributos. AODU (exclui operadores aritméticos unários básicos) , COR (substitui operadores condicionais binários por outros operadores condicionais), AOIS (insere um operador aritmético de atalho) e AOIU (insere um operador aritmético unário básico)

Figura 16 – Experimento 2.1: Regras geradas pelo algoritmo hierárquico para o método pirâmide

```
VDL <= 0
|   ROR <= 2: cluster0 (40.0)
|   ROR > 2: cluster1 (13.0)
VDL > 0
|   AODU <= 0: cluster5 (14.0)
|   AODU > 0
|   |   VDL <= 50: cluster2 (20.0)
|   |   VDL > 50
|   |   |   LOI <= 48: cluster3 (8.0)
|   |   |   LOI > 48: cluster4 (7.0)
```

Fonte – Elaborada pelo autor

A Figura 16 demonstra que a regra gerada para o experimento 2.1 utilizando o algoritmo Hierárquico e Pirâmide foi definida de acordo com quatro atributos: VDL (exclui todas as referências de variáveis de todas as expressões), ROR (substitui operadores relacionais por outros operadores relacionais e substitui todo o predicado com verdadeiro e falso), AODU (exclui operadores aritméticos unários básicos) e LOI (insere um operador lógico unário)

Figura 17 – Experimento 2.2: Regras geradas pelo algoritmo EM para o método pirâmide

```
VDL <= 0
|   ROR <= 2
|   |   COI <= 11: cluster0 (22.0)
|   |   COI > 11: cluster2 (18.0)
|   ROR > 2: cluster5 (13.0)
VDL > 0
|   AODU <= 0: cluster3 (14.0)
|   AODU > 0
|   |   VDL <= 50: cluster1 (20.0)
|   |   VDL > 50: cluster4 (15.0)
```

Fonte – Elaborada pelo autor

Para o experimento 2.2 foi utilizado o algoritmo EM e Pirâmide a Figura 17 demostra que a regra foi definida com base em quatro atributos. VDL (exclui todas as referências de variáveis de todas as expressões), ROR (substitui operadores relacionais por outros operadores

relacionais e substitui todo o predicado com verdadeiro e falso), AODU (exclui operadores aritméticos unários básicos) e COI (substitui operadores condicionais bináriospor outros operadores condicionais)

Figura 18 – Experimento 2.3: Regras geradas pelo algoritmo K-Means para o método pirâmide

```
ROR <= 35
|   ROR <= 2
|   |   COI <= 11: cluster5 (22.0)
|   |   COI > 11: cluster2 (18.0)
|   ROR > 2: cluster3 (16.0)
ROR > 35
|   VDL <= 50: cluster0 (17.0)
|   VDL > 50
|   |   AODU <= 0: cluster1 (14.0)
|   |   AODU > 0: cluster4 (15.0)
```

Fonte – Elaborada pelo autor

A Figura 18 demonstra a regra que foi definida no experimento 2.3 fazendo uso do algoritmo K-Means e Pirâmide com base em quatro atributos. ROR (substitui operadores relacionais por outros operadores relacionais e substitui todo o predicado com verdadeiro e falso), VDL (exclui todas as referências de variáveis de todas as expressões), AODU (exclui operadores aritméticos unários básicos) e LOI (insere um operador lógico unário).

6.6 REDUÇÃO

Com as regras geradas é possível identificar quais operadores de mutação geram um percentual maior de cobertura e determinar a ordem das classes de equivalência por relevância. Por meio dessa classificação foi possível basear-se na delimitação dessas propriedades para identificação e interpretação das regras.

De acordo com Lenz (2010), o conhecimento gerado é útil para auxiliar o testador na identificação dos melhores casos de teste. Possibilitando predizer propriedades que distinguem cada classe de equivalência, permitindo diferenciar classes que podem ser mais significativas que outras, conforme o aspecto que o testador deseja analisar e priorizar no teste de regressão. Desse modo as regras estabelecem as classes mais relevantes, considerando os melhores percentuais de cobertura, para execução do teste de regressão.

Com a utilização das regras busca-se reduzir o número de casos de teste utilizados e assegurar o percentual de cobertura dos operadores. Lenz (2010) , apresentou a abordagem baseada em progressão geométrica (PG), que permite definir o número de casos de testes que devem ser executados, de cada classe de equivalência.

Com base nos elementos que formam a PG é indicado o número de casos de testes que devem ser selecionados, de cada classe de equivalência. O primeiro passo é a ordenação das classes segundo as regras definidas, posteriormente, o número de casos de teste é selecionado com base na porcentagem gerada para o elemento da PG. Para a classe de equivalência com menor prioridade seleciona-se somente um caso de teste, desta forma é dado o primeiro elemento da PG. O testador deve indicar no decorrer da execução o valor do fator de redução, definindo assim o enésimo termo da PG.

Com os elementos informados é possível calcular a razão da PG e os demais elementos da PG. Baseados nos elementos da PG é indicada a quantidade de testes que deve ser selecionada de cada classe, para apoiar o teste de regressão.

6.6.1 Aplicação das Regras

Santos (2016), definiu as etapas para execução da redução aplicando a regra da PG como as listadas abaixo:

1. O valor do primeiro elemento da PG é definido como $\frac{1}{TotalCTP}$ onde Total CTP é o total de casos de teste da classe de menor prioridade;;
2. Valor inicial é dado através do Fator de Redução (FR) informado pelo usuário.
3. Razão q^{n-1} onde n é o número total de *clusters* encontrado pelo algoritmo de agrupamento;

Para adequação a presente pesquisa o item 2 passou a receber uma valor variado (fator de redução) de acordo com o indicado pelo testador e limitado aos valores 15, 30, 50, 80 e 100.

6.6.1.1 Aplicação das Regras para o método MDC

O interpretador inferiu a árvore de prioridade, na ordem que se segue: AOIS, COR, AOIU, COI, AODU, para o experimento 1.1. O critério AOIS é definido como de maior prioridade por situar-se no nó da árvore mais a direita, no qual está o cluster4 que apresenta os casos de

teste que geraram porcentagem de cobertura maior, sendo assim definido como um nó folha da direita do critério. O critério COR é de menor prioridade, encontrado mais próximo da raiz da árvore, o critério COR tem como nó folha a esquerda o cluster3, com os casos de teste de menor relevância. Sendo assim, de acordo com a interpretação das regras, a ordem de prioridade estabelecida para os clusters é: cluster4, cluster2, cluster8, cluster5,cluster9,cluster1,cluster7, cluster6, cluster0 e cluster3. Tomando-se como exemplo o FR 30 visualizado na Tabela 16, podemos inferir que: A quantidade de classes define o número de termos da PG = 10. Quando o testador informa o fator de redução de valor 30 (trinta), esse valor é dividido por 100 (cem) e é atribuído ao enésimo termo da PG. O *cluster* de menor prioridade contém 9 casos de teste e aplicando-se a regra definida anteriormente o primeiro termo da PG é definido como sendo a1 = 0.111. A razão da PG é 1.117 , data pela aplicação da formula geral da PG apresentada através da Equação 5.2, apresentada no Capítulo 5. Com esse valores definidos é possível aplicar a Equação 5.4 apresentada no Capítulo 5 e calcular os valores dos demais elementos.

A quantidade de casos de teste a ser selecionada é calculada a partir da multiplicação entre o número de casos de teste encontrado em cada *cluster*, por ordem de prioridade em ordem decrescente e os valores que formam a PG, a partir do enésimo termo, em ordem idêntica aos *clusters*. Nesse exemplo encontramos 25 casos de testes a serem executados como pode ser visto na Tabela 16. As Tabelas 15, 16, 17, 18 e 19 apresentam os valores gerados para PG e a quantidade selecionada de cada *cluster* para o experimento 1.1.

Tabela 15 – Experimento 1.1: Redução dos casos de teste utilizando o algoritmo hierárquico para o método MDC com fator de redução 15

Classe de Equivalência	Valores PG	Casos de Teste
4	0.15	3
2	0.145	2
8	0.14	1
5	0.136	3
9	0.131	1
1	0.127	2
7	0.123	2
6	0.119	1
0	0.115	1
3	0.111	1
Total		17

Tabela 16 – Experimento 1.1: Redução dos casos de teste utilizando o algoritmo hierárquico para o método MDC com fator de redução 30

Classe de Equivalência	Valores PG	Casos de Teste
4	0.3	5
2	0.269	3
8	0.241	1
5	0.216	5
9	0.193	1
1	0.173	3
7	0.155	3
6	0.138	2
0	0.124	1
3	0.111	1
Total		25

Tabela 17 – Experimento 1.1: Redução dos casos de teste utilizando o algoritmo hierárquico para o método MDC com fator de redução 50

Classe de Equivalência	Valores PG	Casos de Teste
4	0.5	8
2	0.423	4
8	0.358	2
5	0.303	7
9	0.256	2
1	0.217	3
7	0.183	3
6	0.155	1
0	0.131	1
3	0.111	1
Total		32

Tabela 18 – Experimento 1.1: Redução dos casos de teste utilizando o algoritmo hierárquico para o método MDC com fator de redução 80

Classe de Equivalência	Valores PG	Casos de Teste
4	0.798	13
2	0.641	6
8	0.515	3
5	0.413	9
9	0.332	2
1	0.267	4
7	0.214	4
6	0.172	2
0	0.138	1
3	0.111	1
Total		45

Tabela 19 – Experimento 1.1: Redução dos casos de teste utilizando o algoritmo hierárquico para o método MDC com fator de redução 100

Classe de Equivalência	Valores PG	Casos de Teste
4	0.1	16
2	0.785	7
8	0.615	3
5	0.481	10
9	0.377	2
1	0.295	4
7	0.231	4
6	0.181	2
0	0.142	1
3	0.111	1
Total		48

O interpretador inferiu a árvore de prioridade para o experimento 1.2 da seguinte forma: ordem adotada AOIS, AODU, AOIU, onde o operador AOIS é o critério de maior prioridade, e é encontrado no nó da árvore mais externo a direita, no qual o cluster7, que apresenta os casos de teste que geraram porcentagem de cobertura maior, é inserido na árvore como o nó folha da direita do operador AOIS. O critério de menor prioridade é encontrado mais próximo da raiz da árvore, o critério AOIU que tem como no folha a esquerda o cluster0 que é o de menor prioridade. As Tabelas 20, 21, 22, 23 e 24, apresentam a ordem dos *clusters* e sua respectiva quantidade de casos de teste a ser selecionada.

Tabela 20 – Experimento 1.2: Redução dos casos de teste utilizando o algoritmo EM para o método MDC com fator de redução 15

Classe de Equivalência	Valores PG	Casos de Teste
9	0.15	3
0	0.13	1
6	0.113	1
4	0.098	2
3	0.085	3
1	0.064	1
7	0.055	1
2	0.048	1
8	0.115	1
5	0.111	1
Total		15

Tabela 21 – Experimento 1.2: Redução dos casos de teste utilizando o algoritmo EM para o método MDC com fator de redução 30

Classe de Equivalência	Valores PG	Casos de Teste
9	0.3	5
0	0.238	1
6	0.189	1
4	0.151	4
3	0.12	3
1	0.095	1
7	0.076	1
2	0.06	1
5	0.048	1
Total		18

Tabela 22 – Experimento 1.2: Redução dos casos de teste utilizando o algoritmo EM para o método MDC com fator de redução 50

Classe de Equivalência	Valores PG	Casos de Teste
9	0.499	8
0	0.372	2
6	0.278	2
4	0.207	2
3	0.155	4
1	0.115	1
5	0.086	1
2	0.064	1
7	0.048	1
Total		22

Tabela 23 – Experimento 1.2: Redução dos casos de teste utilizando o algoritmo EM para o método MDC com fator de redução 80

Classe de Equivalência	Valores PG	Casos de Teste
9	0.798	13
0	0.562	3
6	0.395	2
4	0.278	6
3	0.196	5
1	0.138	1
8	0.097	1
2	0.068	1
7	0.048	1
Total		33

Tabela 24 – Experimento 1.2: Redução dos casos de teste utilizando o algoritmo EM para o método MDC com fator de redução 100

Classe de Equivalência	Valores PG	Casos de Teste
9	0.685	16
0	0.469	3
6	0.321	2
4	0.219	7
3	0.15	6
1	0.103	1
5	0.07	1
2	0.068	1
8	0.048	1
Total		38

A partir do experimento 1.3, o interpretador definiu a árvore de prioridade, com os critérios AOIS, AOIU. Sendo o operador AOIS o critério de maior prioridade, e é encontrado no nó da árvore mais a direita, no qual é encontrado o cluster9, que apresenta os casos de teste que geraram porcentagem de cobertura maior, é inserido na árvore como o nó folha da direita do operador AOIS. O critério de menor prioridade é encontrado mais próximo da raiz da árvore, o critério AOIU que tem como no folha a esquerda o cluster1 que é o de menor prioridade. s Tabelas 25, 26, 27, 28 e 29, apresentam a ordem dos *clusters* e a respectiva quantidade de casos de teste a ser selecionada.

Tabela 25 – Experimento 1.3: Redução dos casos de teste utilizando o algoritmo K-Means para o método MDC com fator de redução 15

Classe de Equivalência	Valores PG	Casos de Teste
7	0.15	3
5	0.145	2
9	0.014	4
2	0.136	1
0	0.131	2
8	0.127	3
4	0.123	1
1	0.119	1
6	0.115	1
3	0.111	1
Total		20

Tabela 26 – Experimento 1.3: Redução dos casos de teste utilizando o algoritmo K-Means para o método MDC com fator de redução 30

Classe de Equivalência	Valores PG	Casos de Teste
7	0.3	5
5	0.269	3
9	0.241	6
2	0.216	1
0	0.193	3
8	0.173	3
4	0.155	1
1	0.138	1
6	0.124	1
3	0.111	1
Total		25

Tabela 27 – Experimento 1.3: Redução dos casos de teste utilizando o algoritmo K-Means para o método MDC com fator de redução 50

Classe de Equivalência	Valores PG	Casos de Teste
7	0.5	8
5	0.423	4
9	0.358	9
2	0.303	2
0	0.256	4
8	0.217	4
4	0.183	1
1	0.155	1
6	0.131	1
3	0.111	1
Total		35

Tabela 28 – Experimento 1.3: Redução dos casos de teste utilizando o algoritmo K-Means para o método MDC com fator de redução 80

Classe de Equivalência	Valores PG	Casos de Teste
7	0.111	13
5	0.138	6
9	0.172	13
2	0.214	2
0	0.267	4
8	0.332	5
4	0.413	1
1	0.515	1
6	0.641	1
3	0.798	1
Total		47

Tabela 29 – Experimento 1.3: Redução dos casos de teste utilizando o algoritmo K-Means para o método MDC com fator de redução 100

Classe de Equivalência	Valores PG	Casos de Teste
4	0.111	16
2	0.142	7
8	0.181	3
5	0.231	10
9	0.295	2
1	0.377	4
7	0.481	4
6	0.615	2
0	0.785	1
3	0.819	1
Total		50

6.6.1.2 Aplicação das Regras no Pirâmide

Tomando como base o experimento 2.3, o interpretador definiu a árvore de prioridade, com os critérios: COI, AODU, ROR, VDL, sendo o critério AODU o critério de maior prioridade, o mesmo situar-se no nó da árvore mais a direita, no qual é encontrado o cluster4, que apresenta os casos de teste que geraram porcentagem de cobertura maior, o cluster4 é inserido na árvore como o nó folha da direita do critério AODU. Estabelecendo sua maior prioridade sobre os demais *clusters*. O critério de menor prioridade é encontrado mais próximo da raiz da árvore, o critério VDL que tem como no folha a esquerda o cluster0. As Tabelas 30, 31, 32, 33 e 34, apresentam os valores gerados para PG e a quantidade selecionada de cada *cluster* para os experimentos 2.3

Tabela 30 – Experimento 2.1: Redução dos casos de teste utilizando o algoritmo hierárquico para o método pirâmide com fator de redução 15

Classe de Equivalência	Valores PG	Casos de Teste
4	0.15	2
3	0.105	1
2	0.073	2
5	0.051	1
1	0.036	1
0	0.025	1
Total		8

Tabela 31 – Experimento 2.1: Redução dos casos de teste utilizando o algoritmo hierárquico para o método pirâmide com fator de redução 30

Classe de Equivalência	Valores PG	Casos de Teste
4	0.3	3
3	0.183	2
2	0.111	3
5	0.068	1
1	0.041	1
0	0.025	1
Total		11

Tabela 32 – Experimento 2.1: Redução dos casos de teste utilizando o algoritmo hierárquico para o método pirâmide com fator de redução 50

Classe de Equivalência	Valores PG	Casos de Teste
4	0.5	5
3	0.275	3
2	0.151	3
5	0.083	2
1	0.046	1
0	0.025	1
Total		15

Tabela 33 – Experimento 2.1: Redução dos casos de teste utilizando o algoritmo hierárquico para o método pirâmide com fator de redução 80

Classe de Equivalência	Valores PG	Casos de Teste
4	0.8	5
3	0.4	5
2	0.02	3
5	0.01	3
1	0.05	1
0	0.025	1
Total		18

Tabela 34 – Experimento 2.1: Redução dos casos de teste utilizando o algoritmo hierárquico para o método pirâmide com fator de redução 100

Classe de Equivalência	Valores PG	Casos de Teste
4	0.1	7
3	0.478	5
2	0.229	3
5	0.109	3
1	0.052	1
0	0.025	1
Total		20

Para o experimento 2.2, o interpretador definiu a árvore de prioridade, com os os operadores VDL, AODU, ROR, COI. Sendo o operador VDL o critério de maior prioridade. O mesmo está no nó da árvore mais a direita, no qual situar-se o cluster4, que apresenta os casos de teste que geraram porcentagem de cobertura maior, e é inserido na árvore como o nó folha da direita do critério, determinando sua maior prioridade sobre os demais *clusters*. O critério de menor prioridade é encontrado mais próximo da raiz da árvore, o critério COI que tem como no folha a esquerda o cluster0 que é o de menor prioridade. As Tabelas 35, 36, 37, 38 e 39, apresentam os valores gerados para PG e a quantidade selecionada de cada *cluster* para os experimentos 2.2

Tabela 35 – Experimento 2.2: Redução dos casos de teste utilizando o algoritmo EM para o método pirâmide com fator de redução 15

Classe de Equivalência	Valores PG	Casos de Teste
4	0.15	3
1	0.118	2
3	0.093	2
5	0.073	3
2	0.057	1
0	0.045	1
Total		12

Tabela 36 – Experimento 2.2: Redução dos casos de teste utilizando o algoritmo EM para o método pirâmide com fator de redução 30

Classe de Equivalência	Valores PG	Casos de Teste
4	0.3	5
1	0.205	5
3	0.14	2
5	0.096	2
2	0.066	2
0	0.045	1
Total		17

Tabela 37 – Experimento 2.2: Redução dos casos de teste utilizando o algoritmo EM para o método pirâmide com fator de redução 50

Classe de Equivalência	Valores PG	Casos de Teste
4	0.5	8
1	0.309	7
3	0.191	3
5	0.118	2
2	0.073	2
0	0.045	1
Total		23

Tabela 38 – Experimento 2.2: Redução dos casos de teste utilizando o algoritmo EM para o método pirâmide com fator de redução 80

Classe de Equivalência	Valores PG	Casos de Teste
4	0.8	12
1	0.253	9
3	0.142	4
5	0.08	2
2	0.073	2
0	0.045	1
Total		30

Tabela 39 – Experimento 2.2: Redução dos casos de teste utilizando o algoritmo EM para o método pirâmide com fator de redução 100

Classe de Equivalência	Valores PG	Casos de Teste
4	0.1	14
1	0.537	11
3	0.289	4
5	0.156	3
2	0.084	2
0	0.045	1
Total		35

Para o experimento 2.3, o interpretador definiu a árvore de prioridade, com os operadores AODU, VDL, COI. Onde o operador AODU é definido o critério de maior prioridade, e é encontrado no nó da árvore mais a direita, no qual situar-se o cluster4, que apresenta os casos de teste que geraram porcentagem de cobertura maior, e é inserido na árvore como o nó folha da direita do operador AODU, definindo sua maior prioridade sobre os demais. O critério de menor prioridade é encontrado mais próximo da raiz da árvore, o critério COI que tem como no folha a esquerda o cluster5 que é o de menor prioridade. Os experimentos com fator de redução

definidos com valor 15, 30, 50, 80 e 100 apresentaram os melhores valores para a redução. As Tabelas 40 e 41, 42, 43 e 44, apresentam os valores gerados para PG e a quantidade selecionada de cada *cluster* para os experimentos 2.3.

Tabela 40 – Experimento 2.3: Redução dos casos de teste utilizando o algoritmo K-Means para o método pirâmide com fator de redução 15

Classe de Equivalência	Valores PG	Casos de Teste
4	0.8	3
1	0.45	2
0	0.253	2
3	0.142	2
2	0.08	2
5	0.045	1
Total		12

Tabela 41 – Experimento 2.3: Redução dos casos de teste utilizando o algoritmo K-Means para o método pirâmide com fator de redução 30

Classe de Equivalência	Valores PG	Casos de Teste
4	0.8	5
1	0.45	3
0	0.253	3
3	0.142	2
2	0.08	2
5	0.045	1
Total		16

Tabela 42 – Experimento 2.3: Redução dos casos de teste utilizando o algoritmo K-Means para o método pirâmide com fator de redução 50

Classe de Equivalência	Valores PG	Casos de Teste
4	0.8	8
1	0.45	5
0	0.253	4
3	0.142	3
2	0.08	2
5	0.045	1
Total		23

Tabela 43 – Experimento 2.3: Redução dos casos de teste utilizando o algoritmo K-Means para o método pirâmide com fator de redução 80

Classe de Equivalência	Valores PG	Casos de Teste
4	0.8	12
1	0.45	7
0	0.253	5
3	0.142	3
2	0.08	2
5	0.045	1
Total		30

Tabela 44 – Experimento 2.3: Redução dos casos de teste utilizando o algoritmo K-Means para o método pirâmide com fator de redução 100

Classe de Equivalência	Valores PG	Casos de Teste
4	0.1	15
1	0.537	8
0	0.289	5
3	0.156	3
2	0.084	3
5	0.045	2
Total		34

6.7 ANÁLISE DOS RESULTADOS

Com as execuções dos experimentos, empregando o método de redução, foi possível chegar a quantidade de casos de teste para cada classe de equivalência. Para avaliação do novo conjunto, foram tomados como parâmetros de análise o tempo de execução e escore de mutação, sendo possível dessa forma validar os resultados alcançados, confrontando com os dados das porcentagens de cobertura dos critérios de teste atingidos pela abordagem *retest all*.

Os resultados da execução dos algoritmos de teste MDC e Pirâmide com relação a abordagem *retest all*, são apresentados nas Tabelas 45 e 46, respectivamente..

Tabela 45 – Abordagem *retest-all* para o método MDC: escore de mutação x tempo de execução

Score Mutação	Tempo de Execução
93%	01:27:32

Para uma melhor compreensão e visualização dos resultados comparativos, resultantes da execução do novo conjunto e da abordagem *retest all*, os dados foram dispostos e organizados em gráficos de tendências.

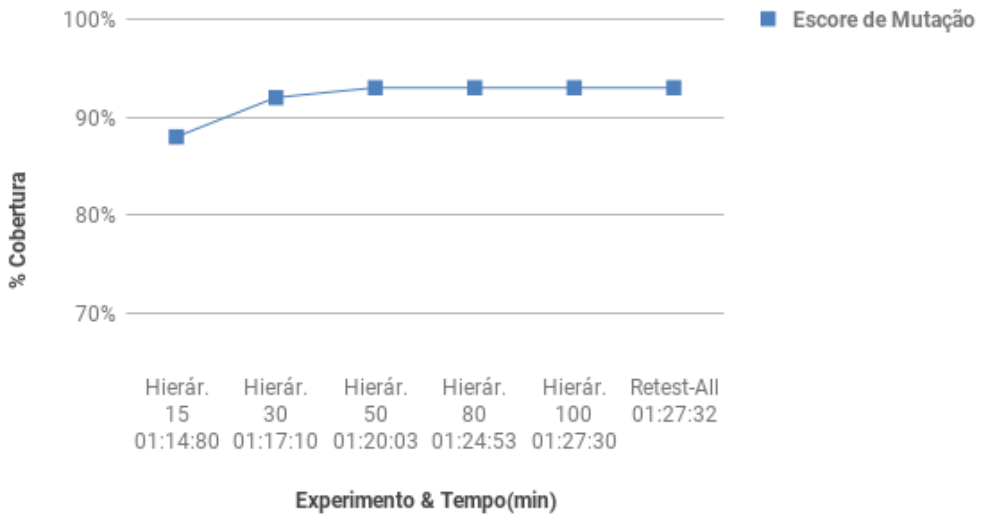
Tabela 46 – Abordagem *retest-all* para o método Pirâmide: escore de mutação x tempo de execução

Score Mutação	Tempo de Execução
92%	00:54:23

Nas Figuras 19, 20 e 21, são apresentados os valores encontrados em cada experimento utilizando-se o método MDC combinado com os algoritmos Hierárquico, EM e K-Means.

A Figura 19 apresenta o gráfico do escore de mutação e tempo de execução atingido pelo experimento 1.1, que combina o algoritmo Hierárquico e método MDC. Verificasse que na utilização dos fatores de redução de 50, 80 e 100, o novo conjunto definido manteve o escore de mutação da abordagem *retest all* e diminuiu o tempo de execução, observando-se que para o FR 100 essa diminuição foi de apenas 2 segundos.

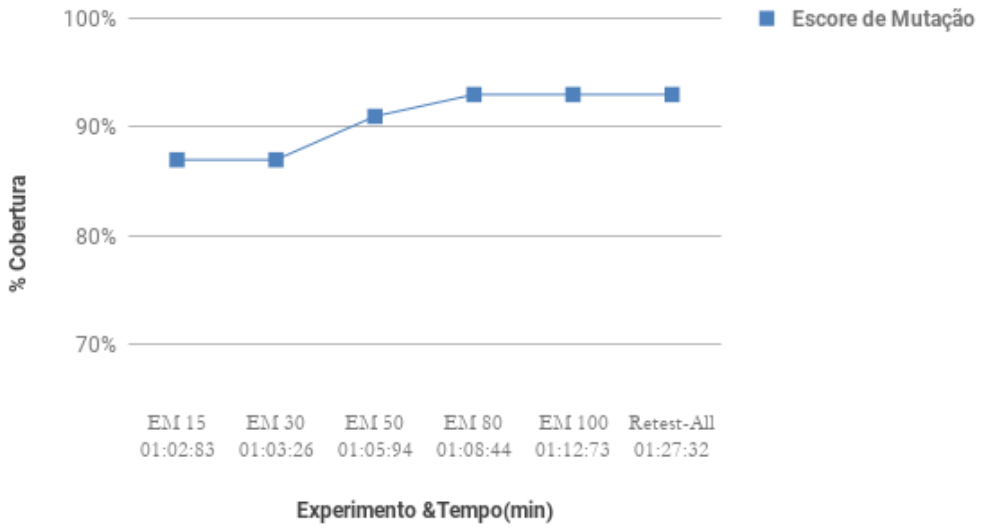
Figura 19 – Experimento 1.1: Tendência de cobertura e tempo de execução utilizando o algoritmo hierárquico para o método MDC



Fonte – Elaborada pelo autor

Os resultados dos experimentos 1.2 observados na Figura 20, com o método MDC mais o algoritmo EM, revelam que só a partir da utilização dos fatores de redução 80 e 100 o conjunto reduzido conseguiu manter o mesmo escore de mutação conseguido na abordagem *retest all*, diminuindo o tempo de execução.

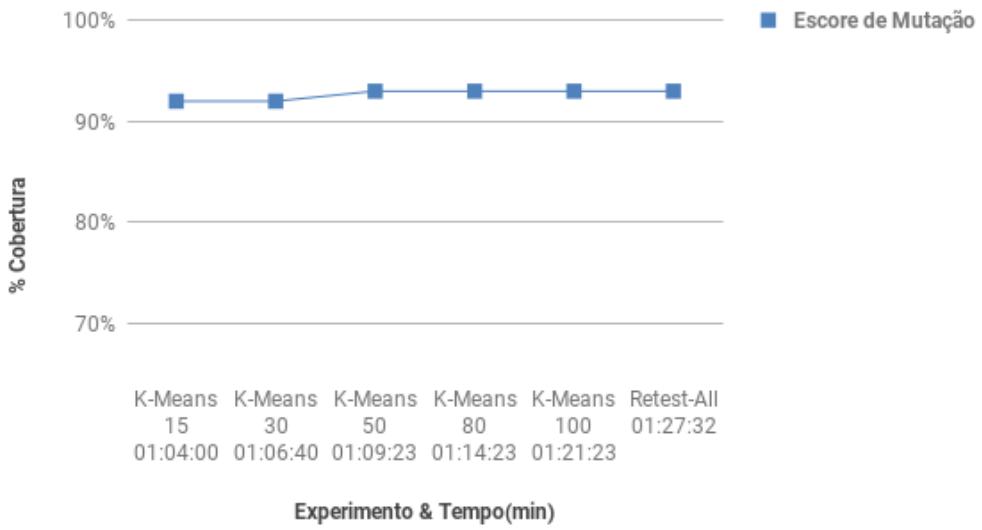
Figura 20 – Experimento 1.2: Tendência de cobertura e tempo de execução utilizando o algoritmo EM para o método MDC



Fonte – Elaborada pelo autor

Para o experimento 1.3, utilizando o algoritmo K-Maens e com fatores de redução definidos em 50, 80 e 100, obteve-se um conjunto de casos que manteve o mesmo escore de mutação da abordagem *retest all*. Com o uso dos fatores de redução 15 e 30, os resultados foram melhores que os combinados com os algoritmos Hierárquico e EM, mas não alcançaram os níveis da abordagem *retest all*. Os resultados podem ser vistos na Figura 21.

Figura 21 – Experimento 1.3: Tendência de cobertura e tempo de execução utilizando o algoritmo K-Means para o método



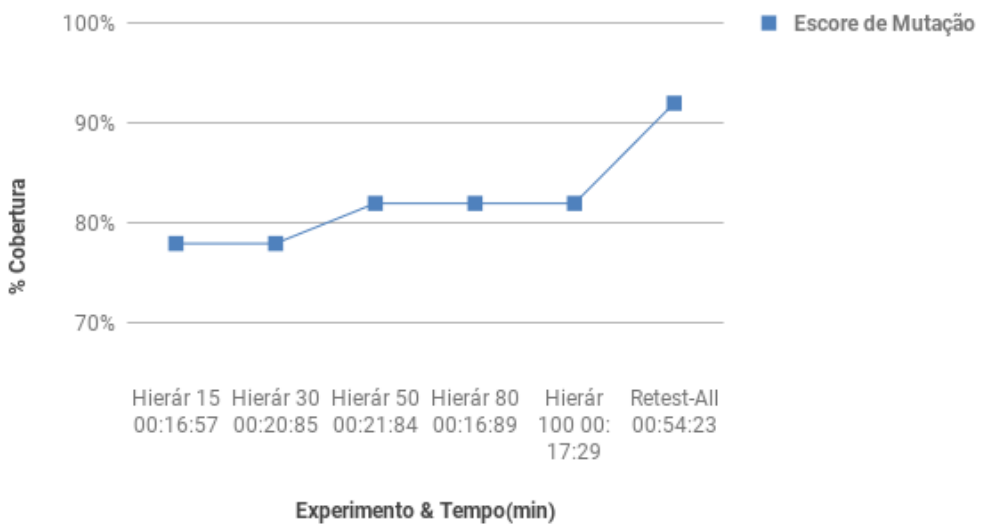
Fonte – Elaborada pelo autor

Nos gráficos expostos para os experimentos com o método MDC é possível notar que com os algoritmos K-Means e Hierárquico foi possível atingir a redução no esforço da atividade de teste de regressão através da redução do tempo gasto, com o uso dos fatore 50, 80 e 100, garantindo a maximização da cobertura dos critérios de teste. Combinado com o algoritmo EM a redução no esforço só é conseguida no uso dos fatores 80 e 100, destaca-se que o FR 80 obteve a melhor redução do tempo gasto para o novo conjunto entre todos os experimentos aliado com a maximização da cobertura do escore de mutação.

Nas Figuras 22, 23 e 24, são apresentados os valores encontrados em cada experimento utilizando-se o método Pirâmide combinado com os algoritmos Hierárquico, EM e K-Means.

O gráfico da Figura 22 apresenta os resultados da execução experimento 2.1 com o algoritmo Hierárquico. Os experimentos realizados só começam a apresentar valores consideráveis, mas abaixo do esperado a partir do fator de redução definido em 50 como pode ser observado.

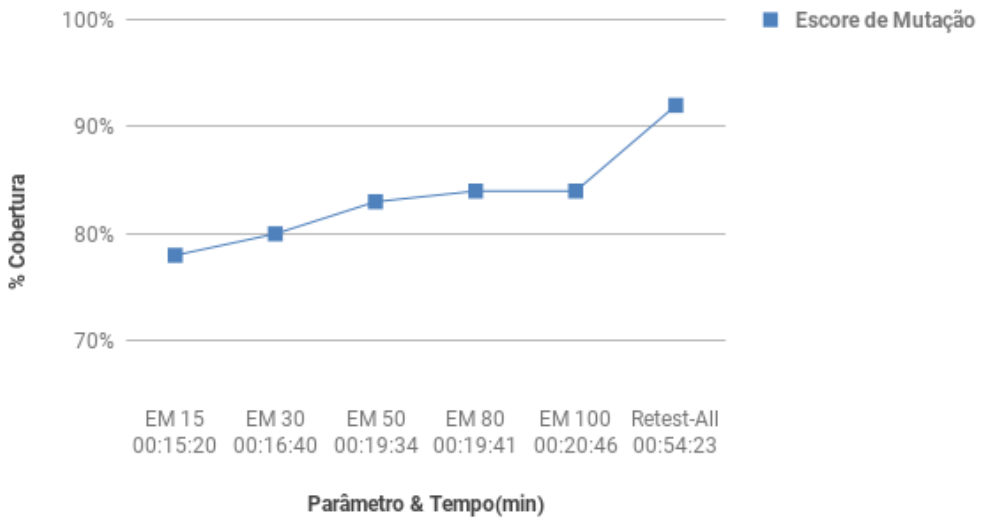
Figura 22 – Experimento 2.1: Tendência de cobertura e tempo de execução utilizando o algoritmo hierárquico para o método pirâmide



Fonte – Elaborada pelo autor

Os resultados alcançados com o experimento 2.2 usando o algoritmo EM, apresentou resultados não favoráveis para os fatores de redução, comparados ao *retest all*. Inicialmente é possível observar uma tendência de crescimento entre o uso dos FR 15 e 80, que se tornar estável entre 80 e 100, como mostrado no gráfico da Figura 23.

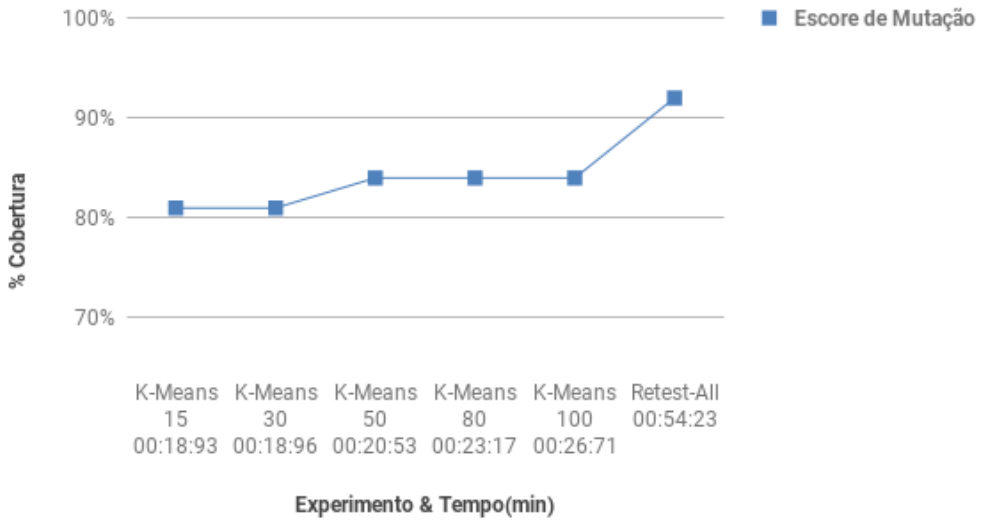
Figura 23 – Experimento 2.2: Tendência de cobertura e tempo de execução utilizando o algoritmo EM para o método pirâmide



Fonte – Elaborada pelo autor

O uso do algoritmo K-Means apresentou resultados abaixo do *retest all*. Apresentando percentuais de escore de mutação iguais entre os FR 15 e 30, passa a melhora os valores de percentual de cobertura com fator 50, mas estabiliza o crescimento do percentual apresentando valores iguais aos encontrados nos fatores de redução 80 e 100, como pode ser observado na Figura 24.

Figura 24 – Experimento 2.3: Tendência de cobertura e tempo de execução utilizando o algoritmo K-Means para o método pirâmide



Fonte – Elaborada pelo autor

Os resultados apresentados nos gráficos para o método Pirâmide e os algoritmos Hierárquico, EM E K-Means, a redução conseguida nos conjunto de casos, provocou perda na porcentagem de cobertura dos operadores, causando uma diminuição significativa desse percentual comparada a abordagem *retest all*. Um ponto importante a se observar e que o tempo de execução dos novos conjuntos é inferior a abordagem *retest all*, para todos os experimentos relacionados.

7 CONCLUSÕES E TRABALHOS FUTUROS

Santos (2016) executou a metodologia proposta por Lenz (2010), de forma manual e propôs que a mesma fosse aplicada por processos automatizados. A partir da proposta de automatização do método de redução do conjunto de casos de teste para apoiar o teste de regressão, se fez necessário a pesquisa e validação de todo o processo. Desde a identificação das classes de equivalência, com aplicação dos algoritmos de agrupamento, passando pela geração e interpretação das regras de classificação, aplicando o algoritmo J48.

A pesquisa automatizou com sucesso todo o processo, utilizando a linguagem Java, permitindo a identificação de redundâncias nos conjuntos de casos de teste. Entretanto a observação dos resultados obtidos com os experimentos realizados na fase de validação com os métodos MDC e Pirâmide não foram o esperado, demostrando que a metodologia aplicada usando progressão geométrica não consegue realizar a remoção de forma eficiente em programas com uma quantidade elevada de redundâncias.

Percebe-se a partir dos resultados apresentados, que a pesquisa conseguiu identificar a ordem de priorização das classes de equivalência de acordo com a regra gerada, Visto que os algoritmos de agrupamento conseguem definir similaridade definindo uma lógica de agrupamento entre os casos de teste. E é possível verificar quais classes de equivalência são mais importantes em relação a um determinado critério a ser investigado. Contudo a aplicação da metodologia usando PG para definição do número de casos a ser reduzido se mostrou pouco eficiente, evidenciando a necessidade de uma nova abordagem.

É importante ressaltar que na execução da pesquisa foi possível perceber que os algoritmos utilizados (EM, Agrupamento Hierárquico, K-means), mostraram-se apropriados para os experimentos realizados. Entretanto comparando os resultados obtidos com a aplicação dos mesmo junto aos métodos MDC e Pirâmide nota-se que a complexidade do programa em teste influencia no escore de mutação atingido do conjunto, revelando que a redução realizada apenas com o critério baseado em erros, não é suficiente, visto que, os critérios definidos na ferramenta não conseguem fornecer sozinhos parâmetros suficientes para definição dos *clustres*. Por tanto, faz-se necessário a combinação de outros critérios para uma melhor redução do conjunto de casos de teste.

7.1 TRABALHOS FUTUROS

A pesquisa apresentada avaliou a metodologia de redução do conjunto de casos de teste aplicada ao teste de regressão apoiada pela técnica baseada em erros, analisando os critérios de eficiência (tempo de execução) e suporte ao escore de mutação. Para atingir resultados mais precisos, sugere-se a definição de uma metodologia que consiga realizar uma redução mais significativa que a apresentada utilizando a progressão geométrica, uma vez que a mesma não se mostrou capaz de indicar uma redução satisfatória. Uma segunda proposta é avaliar a redução do conjunto de casos de testes combinando informações coletadas durante a atividade do teste estrutural e do teste baseado em erros. Tendo em vista a complementariedade das técnicas de teste, a inclusão do teste estrutural deve melhorar significativamente os resultados da redução. Por fim é proposto a definição de um método que consiga criar automaticamente o novo conjunto (classe de teste, .java) dos casos de teste selecionados para redução.

REFERÊNCIAS

AMMANN, P.; OFFUTT, J. Introduction to software testing. Cambridge University Press, 2008.

BARBOSA, E.; MALDONADO, J.; VINCENZI, A.; DELAMARO, M.; SOUZA, S. S.; JINO, M. Introdução ao teste de software. São Carlos: ICMCUSP, 2000.

BERKHIN, P. Survey of clustering data mining techniques. Accrue Software, San Jose, CA, 2002.

BILMES, J. A. e. a. **A gentle tutorial of the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models**. [S.l.]: International Computer Science Institute, 1998. v. 4. 126 p.

CRISTO, T. A. B. Ai-rtesting - um método de seleção e de priorização para apoiar o teste de regressão regressão utilizando aprendizado de maquina. Universidade do Estado da Bahia, 2017.

DELAMARO, M.; JINO, M.; MALDONADO, J. C. **Introdução ao teste de software**. [S.l.]: campus/elsevier, 2007.

DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. **Introdução ao teste de software**. [S.l.]: campus/elsevier, 2004.

DEMILLO, R.; LIPTON, R.; SAYWARD, F. **Hints on Test Data Selection: Help for the Practicing Programmer**. [S.l.: s.n.], 1978. v. 11. 34–41 p.

DEMPSTER, A. P. e. a. **Maximum likelihood from incomplete data via the EM algorithm**. [S.l.]: Journal of the Royal Statistical Society, 1977. B. 1-22 p.

DUDA, R.; HART P.AND STORK, D. Pattern classification. ohn Wiley and Sons, New York, 2001.

ELBAUM, S.; MALISHEVSKY, A.; KALLAKURI, P.; QIU, X.; ROTHERMEL, G. On test suite composition and cost effective regression testing. 2003.

FUNABASHI, J. R. Teste de mutação: subsídios para a redução do custo de aplicação. Instituto de Ciências Matemáticas e de Computação, USP, 2002.

HANMAN, M.; YOO, S. Clustering test cases to achieve effective scalable prioritisation incorporating expert knowledge. King's College London, Centre for Research on Evolution, 2009.

JAIN, A.; DUIN, R. P.; MAO. **Statictical pattern recognition: A review**. [S.l.]: IEEE Transactional on Pattern Analysis and Machine Intelligence, 2000. v. 22. 4 -37 p.

JEYAMALA, D.; SUBASHINI, B. Reduction of test cases using clustering technique. Department of Computer Applications Thiagarajar College of Engineering, Madurai, Tamil Nadu, india., 2014.

JIA, Y.; HARMAN, M. An analysis and survey of the development of mutation testing. Transactions on Software Engineering, 2011.

LENZ, A. R. Utilizando técnicas de aprendizado de máquina para apoiar o teste de regressão. Universidade Federal do Paraná, 2010.

MA, Y.-S.; OFFUTT, J.; KWON, Y. R. Inter-class mutation operators for java. International Symposium on Software Reliability Engineering, 2002.

MA, Y.-S.; OFFUTT, J.; KWON, Y. R. Mujava: an automated class mutation system. software testing. Verificationand Reliability, 2005.

MACQUEEN, J. Some method for classification and analysis of multivariate observations. In: Proceedin of the Fifth Berkeley Symposium on Mathematical Statistics and Probability. Berkeley, CA: University of California Press, p. 281–297, 1967.

MALDONADO, J. Critérios de teste de software: Aspectos teóricos, empíricos e de automatização. São Carlos: ICMCUSP, 1997.

MATEO, P. R.; UAOLA, M. P. Mutant execution cost reduction.international. Conference on Software Testing, 2012.

M.KARTHEEK; RAJSHEKHAR. A novel approach to test suite reduction using data mining. Indian Journal of Computer Science and Engineering (IJCSE), 2013.

MYERS, G. The art of software testing. New York: John Wiley Sons, 1979.

NORVIG, P.; RUSSEL, S. **Inteligência Artificial**. [S.l.]: Editora Campus, 2013. v. 3.

OLIVEIRA, A. A. L. Uma abordagem coevolucionária para seleção de casos de teste e mutantes no contexto do teste de mutação. Universidade Federal de Goiás, Instituto de Informática, 2013.

OSOBA, O. A. Noise benefits in expectation-maximization algorithms. A dissertation Presented to Faculty of the USC Graduate School University of Southern California Doctor or Philosophy in Electrical Engineering, 2013.

PRESSMAN, R. S. **Engenharia de Software**. [S.l.]: Makron Books, 2011. v. 7.

ROTHERMEL, G.; ELBAUM, S.; MALISHEVSKY, A.; KALLAKURI, P.; QIU, X. On test suite composition and cost-effective regression testing. Morgan Kaufmann Publishers, 2nd edition, San Francisco, California, v. 13, n. 3, p. 277–331, 2004.

ROTHERMEL, G.; ELBAUM, S.; MALISHEVSKY, A.; KALLAKURI P.; QIU, X. Data mining: Practical machine learning tools and techniques with java implementations. Morgan Kaufmann Publishers, 2nd edition, San Francisco, California, 2005.

ROTHERMEL, G. e. a. Empirical studies of test-suite reduction. Testing, Verification and Reliability, Wiley Online Library, v. 12, n. 4, p. 219–249, 2002.

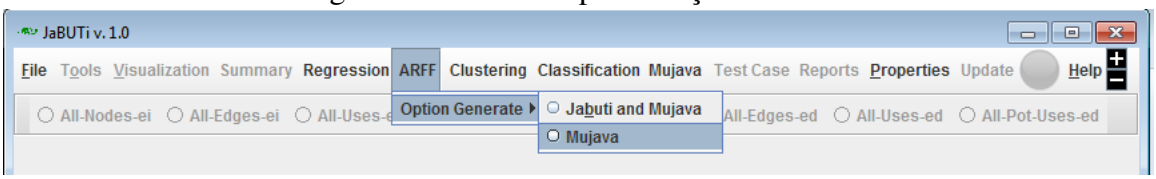
SANTOS, R. B. E. Redução de conjuntos de casos de teste aplicados ao teste de regressão em programas java utilizando algoritmos de aprendizado de máquina. Universidade do Estado da Bahia, 2016.

SOMMERVILLE, I. **Engenharia de Software**. [S.l.]: Editora Pearson, 2010. v. 9.

APÊNDICES

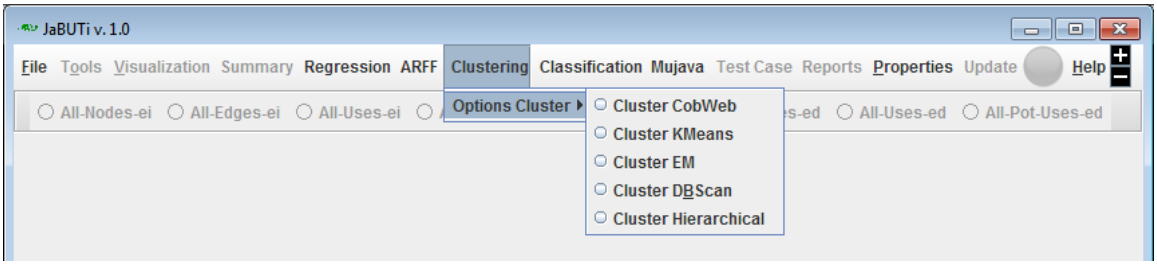
APÊNDICE A – Interfaces Desenvolvidas no Projeto AI+RTESTING

Figura 25 – Interface para criação do ARFF



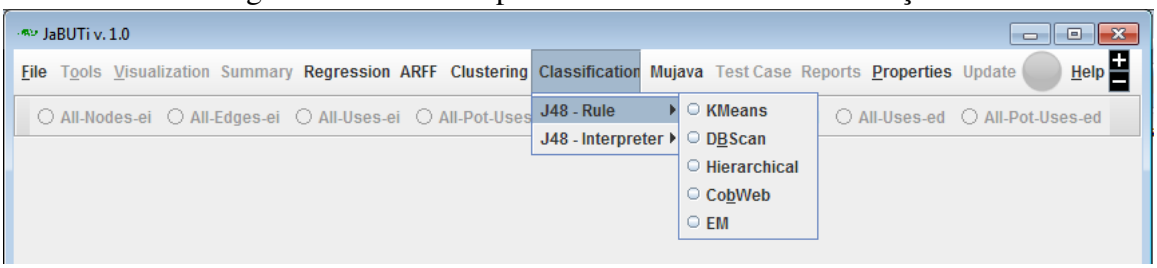
Fonte – Elaborada pelo autor

Figura 26 – Interface para Acionamento do Agrupamento



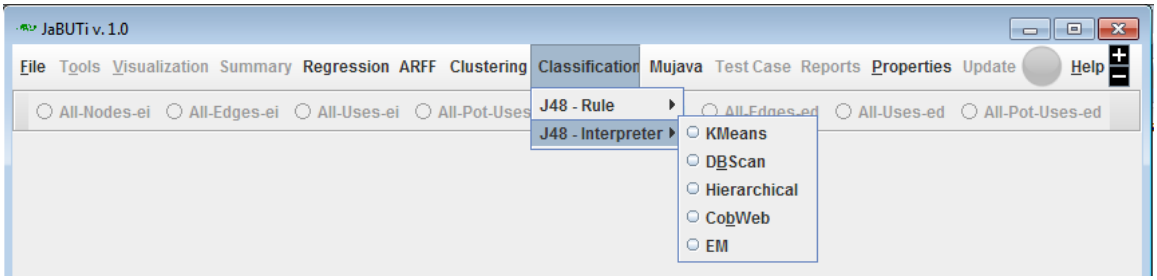
Fonte – Elaborada pelo autor

Figura 27 – Interface para Acionamento da Classificação



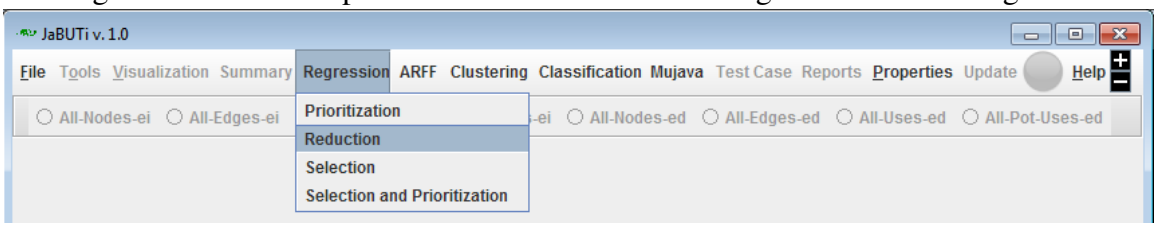
Fonte – Elaborada pelo autor

Figura 28 – Interface para Acionamento da Interpretação das Regras



Fonte – Elaborada pelo autor

Figura 29 – Interface para Acionamento das metodologias do Teste de Regressão



Fonte – Elaborada pelo autor

APÊNDICE B – Implementação dos métodos MDC e Pirâmide

```
1  public class Piramide {
2    public static int piramide(int base1, int base2, int base3, int
       l1, int l2, int l3) {
3      int classe = 1;
4      if (((base1 > 0) && (base1 <= 15)) && ((base2 > 0) && (base2 <=
         15)) && ((base3 > 0) && (base3 <= 15))
5          && ((l1 > 0) && (l1 <= 15)) && ((l2 > 0) && (l2 <= 15)) &&
             ((l3 > 0) && (l3 <= 15))) {
6        if ((base1 < base2) || (base1 < base3) || (base2 < base3)) {
7          classe = -4;
8        } else if (base1 < base2 + base3) {
9          if ((base1 == base2) && (base2 == base3)) {
10           if ((base1 < l1) || (base1 < l2) || (l1 < l2)) {
11             classe = -3;
12           } else if (base1 < l1 + l2) {
13             if ((base1 == l1) && (l1 == l2)) {
14               if (l1 < l3) {
15                 classe = -7;
16               } else if (l1 == l3) {
17                 classe = 3;
18               } else {
19                 classe = -8;
20               }
21             } else {
22               classe = -6;
23             }
24           } else {
25             classe = -5;
26           }
27         } else {
28           classe = -2;
29         }
30       } else {
```

```
31          classe = -1;
32        }
33      } else {
34        classe = -9;
35      }
36
37      return classe;
38    }
39  }
```

Código-fonte 7 – Método Pirâmide

```
1  public class MDC {
2    public static int calcularMDC(int a, int b) {
3      if(a == b && b == 0){
4        return -1;
5      }
6      if (a < -2147483640 || a > 2147483640) return -1;
7      if (b < -2147483640 || b > 2147483640) return -1;
8      if(a < 0) a = -a;
9      if(b < 0) b = -b;
10
11     int r = 0;
12     while (b != 0) {
13       r = a % b;
14       a = b;
15       b = r;
16     }
17     return a;
18   }
19 }
```

Código-fonte 8 – Método MDC

ANEXOS

ANEXO A – Operadores de Mutação a nível de Método

Os operadores de mutação a nível de método definidos para linguagem Java são (MA et al., 2005)

- AORB: Substitui operadores aritméticos binários por outros operadores aritméticos binários.
- AORS: Substitui operadores aritméticos unários por outros operadores aritméticos unários.
- AOIU: Adiciona um operador aritmético unário básico.
- AOIS: Adiciona um operador aritmético de atalho (operador seguido do símbolo "igual").
- AODU: Deleta operadores unários básicos.
- AODS: Deleta um operador aritmético atalho (operador seguido do símbolo "igual").
- ROR: Substitui operadores relacionais por outros operadores relacionais e substitui todo o predicado com verdadeiro e falso.
- COR: Substitui operadores condicionais binários por outros operadores condicionais binários.
- COI: Adiciona operadores condicionais unários.
- COD: Deleta operadores condicionais unários.
- SOR: Substitui operadores de deslocamento por outros operadores de deslocamento.
- LOR: Substitui operadores lógicos binários por outros operadores lógicos binários.
- LOI: Adiciona um operador lógico unário.
- LOD: Exclui um operador lógico unário.
- ASRS: Substitui operadores de atribuição de atalho (operador seguido do símbolo "igual") por outros operadores de atalho do mesmo tipo.
- SDL: Deleta cada instrução executável inserindo comentários. Quando aplicadas para controlar estruturas que incluem um bloco de instruções (por exemplo, if, while e for), o bloco inteiro é eliminado, assim como cada declaração.
- VDL: Todas as ocorrências de referências de variáveis são excluídas de todas as expressões. Quando é necessário preservar a compilação, os operadores também são excluídos.
- CDL: Todas as ocorrências de referências constantes são excluídas de todas as expressões. Quando é necessário preservar a compilação, os operadores também são excluídos.
- ODL: Cada operador aritmético, relacional, lógico, bit a bit, e de deslocamento é excluído de expressões e de atribuições.