Análise preditiva de tendência a falha em módulos de software utilizando programação genética

Vitor Oliveira Franco[1], Marco Antônio Pereira Araújo[1]

[1]Instituto Federal do Sudeste de Minas
Campus Juiz de Fora -- MG – Brasil

Vitoroli101@gmail.com, marco.araujo@ifsudestemg.edu.br

Abstract. *With the growing demand for bigger, safer and more complex softwares, software quality has never been more present in the life of a programmer. One of the most important fields in software quality is testing, which seeks to find faults in software modules. Be able to identify in advance if a software module has falts or not, allow better managment of a project's resources. Several techniques have been used to automate this task, since this procedure can be very expensive, but some have not yet been explored to its full potential, such as the case of genetic programming (GP). This research then seeks to demonstrate the effectiveness of GP in the task of predicting failures in software modules. The results show that the GP model has similar efficiency to the state-of-the-art models, obtaining an precision of 91.818%, sensitivity of 75.537% and accuracy of 77.297%.*

Resumo. *Com a crescente demanda por softwares maiores, mais seguros e mais complexos a qualidade de software nunca se fez tão presente na vida de um programador. Um dos campos mais importantes da qualidade de software são os testes, que buscam encontrar falhas em módulos de um software. Ser capaz de identificar previamente se um módulo de software possui falhas ou não, permite melhor gerenciamento dos recursos de um projeto. Diversas técnicas têm sido utilizadas para automatizar esta tarefa, uma vez que tal procedimento pode ser muito custoso, porém algumas técnicas ainda não foram exploradas em todo seu potencial, como o caso da programação genética (PG). Esta pesquisa busca então demonstrar a eficácia da PG na tarefa predição de falhas em módulos de software. Os resultados mostram que o modelo de PG possui eficiência afim dos modelos estado-da-arte, obtendo precisão de 91,818%, sensibilidade de 75,537% e acurácia de 77,297%*

Palavras-chave: Engenharia de Software, programação genética, tendência a falha.

1. Introdução

A qualidade de *software* é uma área extremamente importante da Engenharia de *Software* que envolve desde o início de um projeto de *software* até a manutenção do mesmo. Com a crescente demanda por *softwares* maiores, mais confiáveis e mais seguros, a qualidade de *software* nunca se fez tão presente na vida de um programador. Tais demandas por sistemas seguros, confiáveis e robustos acabam inevitavelmente aumentando a complexidade do sistema. A alta complexidade de *software* é um dos

principais fatores causadores de falhas, erros e atrasos em projetos de *software* [Basili e Barry 1984]. *Software* de qualidade exige um planejamento cuidadoso e balanceado dos recursos de teste [Naik e Priyadarshi 2011]. As fases de testes são algumas das fases mais críticas de um projeto, pois são nestas fases que os erros e inconsistências são encontrados e corrigidos, além de serem algumas das fases mais custosas de um projeto de *software*, podendo chegar a custar 50% do tempo e orçamento do projeto [Isong e Obeten 2013]. Assim, a utilização de uma estratégia eficiente, eficaz e inteligente para as fases de testes, pode ajudar bastante a minimizar os custos dessas etapas. De acordo com [Malhotra e Jain 2012], apenas 60% das falhas de *software* são encontradas durante revisões de código. Somando isto ao fato de não existir um método universal, simples e não muito custoso para detecção de falhas em um cenário genérico, expõe uma grande problemática para os modelos de desenvolvimento de *software*. Existem muitas metodologias de desenvolvimento que buscam, através da reordenação da fase de testes diminuir o impacto sobre os recursos, como o 'modelo em V' e outras práticas de metodologias ágeis, contudo uma das melhores resoluções para este problema seria conseguir automatizar o máximo possível do processo de identificação de falhas em módulos de *software*. De fato, vários estudos já mostraram resultados positivos para essa automação tais como [Kaur e Malhotra 2008], [Gondra 2008] e [De Carvalho et al. 2010] mas que ainda não alcançaram resultados suficientes para a implementação prática universal de tais métodos.

Uma questão recorrente tanto entre os projetos de automatização de detecção de falhas de software e nos métodos clássicos de revisão de código é a dificuldade da quantificação dos erros e da qualidade dos módulos de software devido à Engenharia de Software não possuir uma medição padrão amplamente aceita e com resultados sem fatores subjetivos [Shatnawi 2013], contudo as métricas de software ainda são as técnicas mais utilizadas para essa tarefa. Existem, de forma geral, duas classificações para uma medida de software: direta e indireta [Harrison et. al. 1998]. Medidas diretas são dados facilmente extraídos de um projeto de *software*, como custo, tempo e número de erros, ou seja, medidas diretas são medidas que não necessitam de nenhuma subjetividade para sua descrição. Já medidas indiretas são medidas como a qualidade de um software, a capacidade de refatoração e manutenção, assim, são medidas que acabam muitas vezes se utilizando de subjetividade para serem contabilizadas, contudo métricas como as métricas de Chidamber-Kemerer [Chidamber e Kemerer 1994], para projetos que se baseiam na utilização do paradigma da orientação a objetos, são utilizadas para evitar subjetividade nas análises. Como dito anteriormente, não existe método totalmente universal que se possa utilizar para dizer se um módulo de software com determinadas métricas possuirá erros, mas a análise das métricas de um módulo de software pode revelar muitos padrões de falhas e ajudar bastante no esforço de testes.

O conceito de inteligência artificial consiste basicamente em mecanismos computacionais que se baseiam no comportamento humano, e de outros seres vivos, para resolver problemas [Charniak 1985]. Dentro do vasto campo de conhecimento de inteligência artificial uma das mais promissoras áreas é o *Machine Learning* (ML), ou, em português, aprendizado de máquina, podendo ser definido como um sistema que pode modificar seu comportamento autonomamente tendo como base a sua própria experiência, com pouca ou nenhuma interferência externa após o início [Murphy 2012]. Nas últimas décadas as técnicas de ML, assim como outras técnicas de inteligência

artificial, ganharam não somente bastante popularidade nos meios acadêmico e industrial [Singh et al. 2018], como também um grande aumento de qualidade e precisão em seus resultados [Singh et al. 2018]. Uma das maiores vantagens do ML é encontrar padrões difíceis de serem encontrados manualmente [Murphy 2012].

A utilização de técnicas de ML como mecanismos de identificação na problemática de predição de tendência a falha em módulos de software é discutida de forma ampla desde a segunda metade da década de 2000 com trabalhos muito renomados, como o de Gondra [2008]. Diversos modelos de ML juntamente com diversas metodologias foram aplicados ao tema, porém devido a rápida evolução dos resultados e popularidade das técnicas utilizadas, alguns modelos de ML acabam sendo pouco aplicados, incluindo diversos modelos muito promissores. Como é o caso dos algoritmos evolutivos, representados pela programação genética (PG), que é justamente uma técnica de aprendizado de máquina que não exige conhecimento prévio de funções otimizadoras, ou seja, não é preciso saber quais combinações de métricas de software retornará na classificação do modulo como defeituoso.

Dessa forma utilização de algoritmos evolutivos, como a programação genética, possuem um potencial pouco explorado. O objetivo do trabalho então é buscar, esclarecer e determinar se a programação genética pode ser aplicada para classificar módulos de software, utilizando métricas de software como base de treinamento e, com isso, produzir resultados afins de técnicas de estado-da-arte da literatura.

2. Conceituação

A seguir são revisados e conceituados tópicos essenciais para que o entendimento dos métodos aplicados no projeto possa ser completo, além do detalhamento da metodologia de pesquisa e revisão sistemática da literatura realizada.

2.1 Defeito de software

Um software geralmente é composto por diversas funções, bibliotecas e arquivos, variando suas localizações, preferências e configurações de acordo com a tecnologia utilizada, por exemplo, se utilizada uma linguagem especializada em orientação à objetos esses componentes provavelmente estarão dispostos em classes. De forma geral, não especificando a tecnologia utilizada, um software é construído em partes, ou módulos, cada módulo é responsável por desempenhar um papel na execução desde software. Dessa forma pode-se definir um módulo como um conjunto de funções, instruções e arquivos desenvolvidos para executar uma tarefa no funcionamento do software. Contudo, durante o desenvolvimento podem ocorrer inconsistências de funcionamento, caracterizando assim o módulo de software com defeito ou falha.

Ao se tratar da área de testes é importante diferenciar alguns conceitos que podem, à primeira vista, ser bastante parecidos e interligados, os conceitos de: Defeito, de Erro e de Falha de software. Segundo a [ISO/IEC/IEEE 29119-3:2013] 'Falha' pode ser definida como um comportamento inesperado do software; 'Defeito' pode ser definido como uma inconsistência no software, algo que foi implementado de maneira incorreta, ocorre em linha de código, como uma instrução errada ou um comando incorreto; 'Erro' pode ser definido por um resultado de um defeito ou uma falha, como um retorno, que por causa de uma falha teve um valor diferente do que esperado. Apesar dessa diferença ser extremamente importante dentro do contexto de testes de

software, durante o desenvolvimento deste trabalho 'defeito' e 'falha' assumirão uma definição semelhante, sendo praticamente sinônimos, sendo considerados como causadores de erros e anormalidades no comportamento do software.

2.2 Métricas de software

Uma das primeiras definições de métricas de software foi proposta por Fenton [Fenton e Neil 2000]. Em seu trabalho, definem métricas de software como um termo coletivo usado para descrever uma grande gama de atividades ligadas à mensuração na Engenharia de Software. Essas atividades estão principalmente ligadas a produção de números caracterizadores de propriedades do código através de modelos que ajudam a predizer a utilização dos recursos e a qualidade do software. Diversas outras definições podem ser encontradas na literatura, mas de forma geral há um consenso com relação à definição de métricas de software. Contudo, como explicam Chidamber e Kemerer [Chidamber e Kemerer 1994], muitas métricas de software desenvolvidas para tentar atender as demandas de gerentes de software, foram recebidas com severas críticas, principalmente baixo referencial teórico em suas criações.

Chidamber e Kemerer propuseram então um conjunto de métricas de software [Chidamber e Kemerer 1994], focadas no desenvolvimento pelo paradigma da orientação a objeto, embasado em formulações algébricas e teóricas. Essas métricas são conhecidas pelo nome dos autores, métricas de Chidamber e Kemerer, e formam as principais métricas utilizadas na literatura ao se tratar de desenvolvimento orientado à objetos.

As métricas a serem utilizadas são: Acoplamento entre objetos de classe (CBO), profundidade na árvore de herança (DTI), falta de coesão dos métodos da classe (LCOM), Número de filhos da classe (NOC), FAN IN, resposta para uma classe (RFC), complexidade dos métodos da classe (WMC), complexidade ciclomática de McCabe (v(g)), Complexidade média do método (Avg Method Cmp.) e linhas de código (LOC).

2.3 Programação genética

Programação genética (PG) é um ramo dos algoritmos evolutivos, e por consequência é inspirada por fenômenos biológicos, como a evolução. A PG cria diversos “programas” de computador que irão performar tarefas pré-definidas, e através de adaptações, sem assumir conhecimento prévio, busca otimizar estes “programas” para executar a tarefa pré-definida, seguindo os passos:

i. todos os programas que irão executar a tarefa são gerados;

ii. depois, cada programa é executado e validado por uma função *fitness*, para medir quão bem executou a tarefa;

iii. após, uma nova população de programas é criada:

- de todos os programas, os melhores são carregados para a nova geração;
- mutações podem ocorrer para criar novos programas;
- *crossover* também ocorrem para evitar vício nos resultados.

iv. finaliza com o melhor programa, de todas as gerações, sendo criado e exibindo o resultado da Programação Genética.

Técnicas de PG são diferentes de técnicas de algoritmos genéticos (AG). Apesar de ambas serem parte do ramo dos algoritmos evolutivos e terem algumas semelhanças, principalmente na forma de evolução dos modelos. A principal diferença entre a PG e o AG é a representação de seus indivíduos, também chamados de cromossomos, em PG são usadas estruturas baseadas em árvores de tamanho variável, tornando a PG muito flexível, mas menos eficiente do que AG para problemas de otimização [Koza 1997]. Já o AG representa seus indivíduos por *strings* de caracteres, binários, de tamanho fixo, sendo menos flexível, mas mais eficiente do que a PG para otimização de parâmetros. Dessa forma a PG é um modelo focado em otimização de problemas onde as estruturas e correlações não são conhecidas e o AG é um modelo focado na otimização de parâmetros em problemas em que as estruturas são relativamente conhecidas.

3. Revisão sistemática

Todo o escopo do trabalho foi estabelecido para responder à pergunta: "Programação Genética pode apresentar resultados afins de outras técnicas encontradas na revisão bibliográfica efetuada?".

3.1 Planejamento

Para atender o objetivo do trabalho de atestar a viabilidade da utilização de Algoritmos Evolutivos, mais especificamente a PG, na construção de preditores de tendência a falha em módulos de um software, utilizando métricas de software como indicadores, uma base de pesquisa foi selecionada. A definição de uma boa fonte de pesquisa é essencial para a revisão sistemática da literatura eficiente. Foi selecionado o Google Acadêmico como o indexador escolhido para a busca, devido ao vasto acervo acurado e pela acessibilidade de disposição de publicações. Com a intenção de identificar e entender os achados específicos de artigos [Goodwill e Pearman 2006] sugere a utilização de cinco critérios *Population*, *Intervention*, *Comparison*, *Outcomes* e *Context*, chamados de PICOC (ou em português: população, intervenção, comparação, resultado e contexto). A definição da metodologia PICOC é de extrema importância para estruturar o pensamento científico durante a pesquisa. A Tabela a seguir descreve o PICOC:

Tabela 1. PICOC

População	Desenvolvedores, Testers, Equipe de qualidade
Intervenção	Machine Learning, Estimativa de falha
Comparação	Métodos estatísticos, Random Forest
Resultado	Estimativa de falha, métricas
Contexto	Manutenção de software, Teste de software

Durante a revisão sistemática da literatura foram encontrados 89 artigos sobre a *string* de busca ' "*Machine learning techniques" AND "prediction" AND "software fault proneness" AND "software metrics" AND "software quality" AND "software Maintenance" AND Evolutionary algorithms"* '. Após a leitura dos títulos dos artigos, 43 foram eliminados, e após a leitura dos resumos dos artigos restantes mais 18 foram eliminados, através da metodologia *Snow Balling* [Wohlin 2014] foram adicionados

mais 9 artigos e, após a leitura completa dos artigos mais 8 artigos foram eliminados assim totalizando 29 artigos selecionados, de acordo com critérios descritos a seguir.

Os critérios de inclusão definidos para esta pesquisa foram:

- estudos sobre o "estado da arte" de técnicas de *Machine Learning* para predizer tendência falha em módulos de software;
- publicações contendo estudos experimentais, casos de estudo ou revisão de técnicas de *Machine Learning* para predizer tendência a falha em módulos de software.

Os critérios de exclusão definidos para esta pesquisa foram:

- produções secundárias de materiais didáticos;
- publicações que não abordam a utilização de métricas de software;
- publicações que não abordam o campo das ciências da computação ou o tópico de Engenharia de Software;
- publicações que não puderam ser acessadas por completo;
- publicações relacionadas a predição de falha de módulos de software, mas que não abordam técnicas de *Machine Learning.*

4. Trabalhos relacionados

A busca pela identificação automática de falhas em módulos de software utilizando técnicas de ML tem gerado muitos trabalhos nos últimos anos [Askari e Bardsiri 2014]. Devido ao grande esforço de pesquisa feito nesta área, a identificação de módulos de software defeituosos por técnicas de ML vem avançando gradativamente.

Um dos primeiros e importantes trabalhos publicados dentro da área foi o de Gondra [2008], que analisou a utilização das técnicas SVM e redes neurais na problemática, e constatou bons resultados.

Uma grande variedade de técnicas de ML é usada na literatura [Askari e Bardsiri. 2014], como: *Random Forest* (RF), *Adaboost, Bagging, Support Vector Machine* (SVM), Redes Neurais Artificiais (ANN), kNN, C4.5 e *Naive Bayes*.

Técnicas que se utilizaram de *Random Forest* (RF) se mostraram como as mais precisas e com melhores resultados [Kaur e Malhotra 2008];

Os melhores resultados utilizando PG vieram de [Malhotra e Jain 2012], apesar disso, o estudo não dá foco na utilização de Algoritmos Evolutivos por si, é mais focado na utilização de diversos métodos e a comparação entre eles.

Em geral, para os trabalhos que realizaram o estudo estatístico da correlação entre as métricas de software e a falha nos módulos de software, mostram que as

métricas acoplamentos entre objetos (CBO), resposta para uma classe (RFC), número de linhas totais (LOC), métodos ponderados por classes (WMC) se mostraram as mais significativas para a predição de falha [Hammouri et al. 2018].

Existe uma grande escassez de base de dados com os elementos para a formulação dos modelos de ML, uma das causas para esse efeito é o fato de poucas empresas privadas disponibilizarem os seus dados de falhas e métricas. Devido a isso, as bases de dados disponíveis são utilizadas diversas vezes por várias publicações diferentes, sendo as bases mais usadas a PROMISE [Promise 2013] e bases com projetos *open source*. A literatura também mostra uma quase inexistência de artigos, sendo replicados a maioria das pesquisas na busca abordagens ligeiramente diferentes para a tarefa de classificação de tendência a falha.

5. Materiais e métodos

Estudos e trabalhos anteriores já provaram que técnicas de ML podem ser aplicadas a diversas áreas e tarefas distintas [Murphy 2012], como por exemplo, mercado financeiro, epidemiologia ou recomendações em lojas online. Contudo o profissional, operador ou pesquisador de cada utilização deve ser capaz de identificar os pontos fortes e as deficiências de cada implementação, para assim tomar o melhor curso de ação. A pesquisa buscou analisar e comparar uma das técnicas mais acuradas e promissoras da literatura sobre predição de falha em módulos software utilizando técnicas de ML, a RF, com uma técnica de Algoritmos Evolutivos, até então pouco utilizados para a problemática, a PG.

5.1 Ambiente de desenvolvimento

Existem diversas formas de se implementar uma técnica de ML. No âmbito da implementação em mais baixo nível existem diversas linguagens especializadas em áreas e técnicas de ML, como por exemplo a LISP ou C para PG [Koza 1990], que exigem conhecimento profundo da técnica a ser implementada, sendo necessário a implementação, configuração de ambiente, tratamento de dados e otimização de parâmetros por conta do próprio desenvolvedor. No âmbito de mais alto nível, software com um conjunto de técnicas já pré-selecionado em que os modelos são criados basicamente variando parâmetros específicos de cada técnica de ML e o tratamento da base de dados são bastante utilizados, como por exemplo o WEKA [Malhotra e Jain 2012]. Mas, ao passo que facilitam e proporcionam resultados muito significativos para aplicações especificas acabam tirando, de certo modo, a liberdade de implementação que pode ser necessária para determinada problemática. E, interpostos à essas duas categorias, existem os ambientes de programação que proporcionam auxilio na construção de modelos de ML, através de bibliotecas e estruturas já pré-implementadas. Contudo, devido à alta capacidade de configuração e a possibilidade de combinar diversas técnicas diferentes na mesma implementação, não comprometem a liberdade e adaptabilidade de modelos a serem implementados como, por exemplo, as linguagens Python e C++, que possuem diversas ferramentas e bibliotecas especializadas na construção de técnicas de ML, especialmente a linguagem Python.

A linguagem Python, está cada vez mais se estabelecendo como uma das linguagens mais populares no mundo e, em diversas aplicações, como desenvolvimento de software e dispositivos IOT, mas também para a computação científica. Devido à sua interação de alto nível e a grande comunidade de desenvolvimento de bibliotecas

cientificas, é uma escolha muito atrativa para o desenvolvimento de modelos de ML. Scikit-learn [Pedregosa et. Al. 2011] é a principal ferramenta que une e controla esse rico ambiente de desenvolvimento, provendo implementações de estado da arte de algumas das mais populares e eficientes técnicas de ML, enquanto mantém uma interface de fácil utilização extremamente integrada com a linguagem Python, sem abrir mão da liberdade de implementação. A biblioteca Scikit-learn também difere de outras ferramentas pra construção de modelos de ML, por ser distribuída em licença BSD, sendo assim, de livre utilização [Pedregosa et. Al. 2011], e também por incorporar códigos compilados, visando uma maior eficiência das execuções.

A biblioteca Scikit-learn, apesar de possuir uma diversidade extremamente grande de implementações de técnicas de ML, não possui suporte para todos os modelos de ML, e a PG é um deles. Devido a esse fato, a utilização de um *framework* que estende justamente as capacidades e escopo do Scikit-learn é extremamente razoável. Assim, para a construção do modelo de PG será utilizado o *framework* gplearn [Stephens 2015].

O *framework* gplearn se utiliza e concentra sua ação em problemas de regressão simbólica, resolvidos através da PG. Sendo uma regressão simbólica um subgrupo dentro da PG, ao invés de encarar cada "indivíduo" criado durante a execução como um "programa de computador" a abordagem é feita através de expressões matemáticas. Seguindo os mesmos quatro passos gerais da PG, primeiro cria-se uma população de expressões aleatória que tentam descrever uma relação, depois todas as expressões são validadas por uma função *fitness* e, em seguida, cada geração irá evoluir a partir dos dados da anterior. Por fim, o melhor indivíduo será o que dará o resultado do modelo.

Utilizar o mesmo conjunto de dados para o desenvolvimento e o teste de um modelo de ML certamente implica em complicações com relações aos resultados, pois um modelo que somente repete rótulos que já tenha conhecimento falhará ao ser utilizado com novos dados, apesar de acertar muito bem com o conjunto de dados original, essa situação é chamada de *overfitting*. Para evitar *overfitting,* uma técnica muito eficiente é dividir o conjunto de dados a disponíveis em três grupos, um conjunto para o treinamento do modelo, um conjunto para a validação e um conjunto para testes. Contudo, ao se particionar os dados disponíveis em três conjuntos, os números de amostras que podem ser utilizadas no desenvolvimento são drasticamente reduzidos.

A solução aplicada para resolver este problema é o procedimento de validação cruzada dez vezes, ou em inglês *tenfold cross-validation*. Onde um conjunto de dados para teste ainda é guardado para a avaliação final, mas o conjunto de validação não é mais necessário, o conjunto de treinamento é dividido em dez conjuntos separados, um modelo é treinado usando nove dos dez conjuntos de treino e o resultado é comparado com o conjunto de teste. A *performance* final da validação cruzada dez vezes é a média dos valores computados nas iterações.

5.2 Programação genética

Atualmente, o desenvolvimento de software é feito programando cada resposta e caminho possível em uma execução do mesmo software. O grande objetivo do aprendizado de máquina, especialmente de técnicas de algoritmos evolutivos como a PG, é ser possível apenas descrever uma tarefa para um computador e ele aprender sem interferência externa especifica a realizar tal tarefa.

A PG que será utilizada neste trabalho é focada nos problemas de regressão simbólica, ou seja, busca relacionar as variáveis independentes (as métricas de software) com a variável dependente (a classificação dos módulos como tendo ou não falha) através de expressões matemáticas, e por não requerer uma especificação *a priori* do relacionamento entre as variáveis, é capaz de evitar *vieses* humanos e explorar relacionamentos contra intuitivos do ponto de vista analítico, além de não ser afetado por especificidades de teorias ainda não conhecidas [Koza 1990].

O aprendizado da PG é realizado, como parte dos algoritmos evolutivos, através de gerações de indivíduos progressivamente se adaptando à tarefa sendo realizada, isto é, diversas instancias (indivíduos) são avaliados por geração e os mais apto a realizar a tarefa é reproduzido na próxima geração juntamente com variações, realizadas por operadores genéticos, desse modo com o passar das gerações indivíduos cada vez mais aptos a realizar a tarefa são criados performando assim o aprendizado da PG é realizado.

5.2.1 Representação

A representação teórica da PG é feita através de expressões matemáticas e se utiliza de combinações de variáveis independentes para representar a variável dependente. Em um cenário hipotético, as variáveis independentes "a" e "b" interagem entre si de acordo com a expressão (1) para definirem a variável dependente "y".

$$y = a^2 - b + 13 \qquad (1)$$

Computacionalmente, utilizando a biblioteca NumPy da linguagem Python, a expressão (1) é descrita apresentado na expressão (2).

```
y   = np.add(np.subtract(np.multiply(a, a), b), 13)          (2)
```

Ainda computacionalmente, outra forma muito comum de se representar os indivíduos da PG é através de Árvore de Análise Sintática, como apresentado na Figura 1, onde em amarelo tem-se as operações matemáticas utilizadas e de vermelho tem-se os valores e variáveis independentes. Apesar da representação por árvores ser extremamente comum e por muito tempo intrínseca à PG, diversos estudos mostram que outras estruturas podem ser usadas [Banzhaf et. al. 1998]. Por facilidade de visualização e compreendimento, neste trabalho será utilizada a representação por árvores.

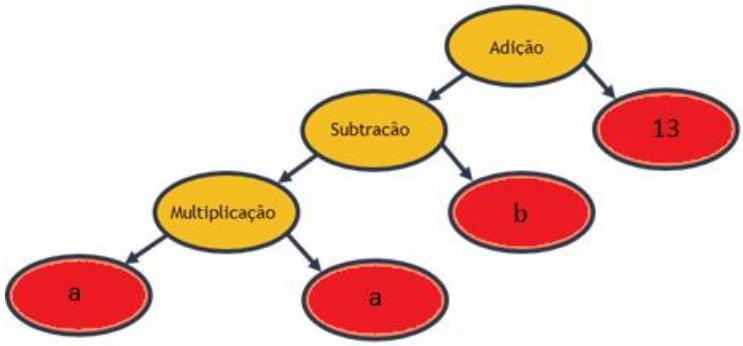

Figura 1: Exemplo de árvore de análise sintática (Fonte: Do autor)

Em cada geração diversos indivíduos aleatórios são criados, essa geração aleatória ocorre a partir de um número aleatório de operações sendo selecionado, após o número de variáveis independentes, podendo ser as métricas de software ou um número

real também aleatório, necessários para atender os operadores é selecionado e o indivíduo assim é gerado.

5.2.2 Fitness

A representação dos indivíduos, tanto teoricamente, através das arvores sintáticas, quanto computacionalmente para a PG, são extremamente úteis, mas tão importante quanto conseguir uma representação dos indivíduos é a capacidade de avaliar o quão o indivíduo conseguiu *performar* na tarefa apresentada. Essa medida é chamada de função *fitness*. A função de perda de entropia cruzada, também chamada de *log loss*, será a métrica utilizada para a otimização e avaliação dos resultados, ou seja, a função *fitness* será dada pela função *log loss*. O objetivo da função *fitness*, no que tange um problema de classificação, é penalizar predições ruins e incentivar predições boas, e a *log loss* busca realizar isso através da utilização de logaritmos. Para problemas de classificação binária a *log loss* é uma das medidas mais indicadas [Murphy 2012]. A função de perda de entropia cruzada é dada pela equação (3), com N sendo a quantidade de indivíduos, y_i sendo a classe predita para o i-ésimo indivíduo e $p(y_i)$ sendo a probabilidade do i-ésimo indivíduo ser de uma das classes do problema de classificação.

$$H(p,q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \times \log\big(p(y_i)\big) + (1-y_i) \times \log\big(1-p(y_i)\big) \quad (3)$$

5.2.3 Operadores Genéticos

Através da função *fitness* os melhores indivíduos podem ser escolhidos, mas visando evitar *vieses* lógicos e azar de uma geração de indivíduos evoluir para versões menos eficientes, os melhores indivíduos da geração anterior passam por processos, inspirados na biologia genética da evolução, de alteração de sua estrutura. Para efetuar a evolução dos modelos serão utilizadas as técnicas: *Crossover*, Mutação pontual e de sub-árvores, Mutação de subtração e reprodução.

Crossover é o princípio da biologia principal responsável por misturar o material genético entre indivíduos, no caso da PG, as árvores sintáticas são enxergadas como o material genético e assim ocorre a troca de um sub-árvore dos indivíduos, como exemplificado na figura (2). Este procedimento é utilizado para possibilitar e incentivar uma geração de indivíduos com maior diversidade, reduzindo *vieses* possivelmente negativos. A mutação pontual é outro conceito baseado na biologia genética, afim de aumentar a variedade "genética" nos indivíduos (leia-se aumentar a diferença entre as árvores) alguns nós de cada árvore sofrem mudanças aleatórias, como visto na figura (2). A mutação de sub-árvores segue o mesmo princípio da pontual, porém é mais agressiva em suas mudanças, ao invés de alterar apenas alguns nós, altera sub-árvores inteiras, também com o objetivo de buscar uma maior diversidade genética entre os indivíduos, como visto na figura (2). Ainda seguindo o princípio da mutação pontual, a mutação de subtração seleciona alguns nós de uma árvore e os remove do indivíduo, buscando através disso retirar componentes genéticos pouco eficientes ou enviesados das próximas gerações, como visto na figura (2).

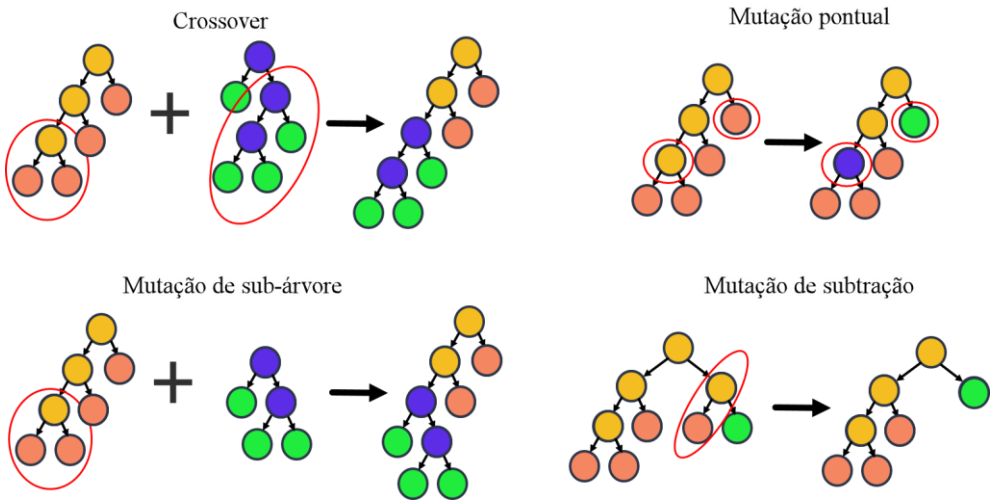

Figura 2: Exemplo de operações de evolução. Amarelo e azul representam operações matemáticas; Vermelho e verde representam operadores (Fonte: Do autor)

E, por fim, a reprodução é o princípio de manter um exemplar replicado um-para-um do indivíduo mais apto da geração, assim evitando que conjuntos ou indivíduos ineficientes possam ser reproduzidos para a próxima geração.

5.2.4 Classificação

Obviamente a execução do método de PG tem de ser parada e seus dados analisados, para isso são usados dois critérios de parada. O primeiro é o número máximo de gerações, caso o modelo execute e crie um determinado número de gerações sem melhora na resposta da função *fitness* a execução é parada. O segundo critério é alcançando um resultado "perfeito" para o *fitness*, o termo "perfeito" não é necessariamente obter 100% de acertos em todas as tarefas executadas, uma vez que este fato é pouco provável e pode indicar até mesmo um *overfitting* do modelo, mas sim o valor "ótimo" previamente estipulado para a tarefa.

Ao ser alcançado um critério de parada, o resultado do modelo, até o momento um resultado numérico, é transformado através de uma função sigmoide que utilizará este resultado numérico e calculará a probabilidade de o resultado pertencer a uma das duas classes, "módulo possui falha" e "módulo não possui falha". Basicamente, ao resultado ser aplicado na função sigmoide, se o novo resultado for mais próximo à 0 ele pertencerá a uma classe e, se for mais próximo de 1 pertencerá a outra classe

5.3 Algoritmo proposto para Programação Genética

Para a construção do modelo de PG foram utilizados os seguintes passos:

Passo 1: gerar aleatoriamente a primeira geração de indivíduos que possuem configurações aleatórios, suas árvores sintáticas não seguem nenhuma pré configuração;

Passo 2: calcular a função *fitness* de cada indivíduo para a classificação de módulos de software, o *fitness* é calculado pela predição aplicada a função de perda de entropia cruzada, também chamada de *log loss*;

É importante ressaltar, que por usar expressões matemáticas, como a divisão, alguns casos de indivíduos podem gerar erros computacionais, como por exemplo uma divisão por zero. Para evitar esse tipo de problema, o algoritmo é equipado com as ações preventivas a seguir:

- em operações envolvendo divisão, se o denominador estiver no intervalo $[-0{,}001;\ 0{,}001,]$ o retorno será 1;
- em operações envolvendo raízes quadradas retornam o valor da raiz absoluta do argumento sendo passado;
- em operações envolvendo logaritmos, se o valor informado para o argumento estiver no intervalo $0;\ 001,]$ o retorno será 0;
- em operações envolvendo inversões, se o valor informado para o argumento estiver no intervalo $[-0{,}001;\ 0{,}001,]$ o retorno será 0.

Passo 4: a condição de cem gerações sem melhora foi escolhida como condição de parada. Existem muitas variações em relação a condição de parada. Esse critério foi escolhido baseado na literatura onde foi observado que 100 gerações sem melhora foram suficientes para gerar bons resultados para métodos evolutivos semelhantes;

Passo 5: 60% da população é selecionada aleatoriamente para competir em um sistema de torneio, onde o indivíduo mais apto é selecionado. A porcentagem da população que irá participar do torneio afeta a velocidade com que o modelo consegue atender um critério de parada pois, geralmente torneios com uma maior porcentagem da população selecionada para a disputa tendem a convergir mais rapidamente para a resposta, o valor de 60% da população foi observado empiricamente como obtendo resultados bons sem uso excessivo de poder computacional;

Passo 6: o vencedor do torneio tem 70% de sofrer *crossover*. O *crossover* se performado escolhe uma sub-árvore aleatória do vencedor do torneio para ser trocada. Um segundo torneio é realizado e o vencedor desse será o doador de uma sub-árvore contida em si, que tomará o lugar da sub-árvore escolhida no vencedor do primeiro torneio. O resultado forma então um indivíduo filho na próxima geração;

Passo 7: o indivíduo vencedor do torneio tem 5% de sofrer uma mutação de sub-árvore, onde uma sub-árvore do vencedor do torneio é selecionada e substituída por uma sub-árvore gerada aleatoriamente. O resultado forma então um indivíduo filho na próxima geração;

Passo 8: o indivíduo vencedor do torneio tem 5% de sofrer uma mutação de subtração, onde uma sub-árvore do vencedor do torneio é selecionada e eliminada de seu código genético. O resultado forma então um indivíduo filho na próxima geração;

Passo 9: o indivíduo vencedor do torneio tem 10% de sofrer uma mutação pontual, onde um número aleatório de nós para serem substituídos por outros nós aleatórios, respeitando a relação de que um nó de variável independente pode ser substituído somente por outra variável independente ou valor numérico e um nó de operação somente pode ser substituído por outro nó de operação. O resultado forma então um indivíduo filho na próxima geração;

Passo 10: o indivíduo vencedor original do torneio através da reprodução também é clonado para próxima geração, buscando assim evitar retrocesso nos resultados;

Passo 11: a nova geração tem seus indivíduos restantes gerados aleatoriamente, assim formando uma nova população. Inicia-se então um *loop*, voltando para o passo 2 até que um critério de parada seja atingido, ou seja, um indivíduo alcance o *fitness* ótimo ou ocorram 100 gerações consecutivas sem aprimoramento do *fitness* obtido, assim encerrando a execução do algoritmo.

5.4 *Randon Forest*

Random Forest é uma técnica de ML de classificação que consiste em uma coleção de classificadores estruturados em árvores de decisões [Breiman 2001]. Seu funcionamento, de forma geral, é examinar uma nova entrada de amostra em cada uma das árvores de decisão que o compõem, cada árvore "vota" na classe resultante, assim a "floresta" elege o resultado total do modelo por votos majoritários. Cada árvore é criada seguindo os passos:

1. considerando o número de amostras como N, um conjunto de dados de tamanho n é extraído da base de dados original aleatoriamente, com reposição. Esse conjunto de dados então passa a ser o conjunto de treinamento para se desenvolver uma árvore de decisão;
2. em cada nó da árvore de decisão uma quantidade p de preditores aleatórios é extraída de todos os P preditores disponíveis na base de treinamento, com $P > p$ e usualmente utilizado $p = \sqrt{P}$, o nó é dividido entre caminhos para classificar a amostra, a melhor divisão entre esses m preditores é usada para a construção da árvore;
3. cada árvore é construída até o tamanho máximo possível, utilizando os p preditores selecionados no passo anterior.

Quando o conjunto de treinamento da árvore atual é construída, cerca de um terço da base de dados, estatisticamente, é inutilizada por não ter sido selecionada. Esse conjunto de amostras não utilizadas é chamado de *oob*, em inglês *out-of-bag*, e esses dados são utilizados então como conjunto de testes obtendo assim um resultado imparcial da classificação. Assim, não necessitando a utilização da validação cruzada ou a separação de um conjunto de dados especificamente para testes [Breiman 2001].

Métodos utilizando o *Random Forest* têm se mostrado mais eficiente para a problemática de predição de tendência a falha em módulos de software do que a maioria dos outros métodos [Kaur e Malhotra 2008], e segundo [Guo, Lan, et al 2004] uma das possibilidades para isso é o fato de *Random Forest* conseguir estimar muito bem dados faltantes.

5.5 Parâmetros de avaliação

Ao se tratar de problemas de classificação em modelos de ML, existem diversas formas de exibir um resultado obtido, contudo uma das maneiras mais visuais e ricas de exibir os resultados é através de uma matriz de confusão. Uma matriz de confusão é uma tabela que permite a visualização da performance de uma técnica de classificação, cada linha da matriz representa uma classe real do objeto, enquanto que cada coluna representa a classe da predição realizada. Cada predição pode ser descrita por um de quatro tipos de classificações, sendo eles:

- **Verdadeiro-positivos (TP):** módulos corretamente classificados como possuidores de falhas;

- **Verdadeiro-Negativos (TN):** módulos corretamente classificados como não possuidores de falhas;
- **Falsos-positivos (FP):** módulos erroneamente classificados como possuidores de falhas;
- **Falsos-Negativos (FN):** módulos erroneamente classificado como não possuidores de falhas.

Esses termos são utilizados para medir a eficiência dos resultados da predição de classificação. Para realizar essa análise de eficiência são utilizados os parâmetros acurácia, precisão, sensibilidade e *F-measure* definidos como:

- **Acurácia (Ac):** proporção geral de classificações realizadas corretamente. Matematicamente: $\frac{TP+TN}{TP+TN+FN+FP}$
- **Precisão (Pr):** proporção de módulos de software corretamente classificados como defeituosos. Matematicamente: $\frac{TP}{TP+FP}$
- **Sensibilidade ou Recall (Re):** proporção de módulos de software defeituosos corretamente classificados como defeituosos. Matematicamente: $\frac{TP}{TP+FN}$
- ***F-Measure*(Fm):** é a média harmônica entre a precisão e a sensibilidade. Utilizada para comparar predições diferentes. Matematicamente: $\frac{2\times Pr\times Re}{Pr+Re}$

Por definição, os parâmetros acurácia, precisão e sensibilidade são dados em porcentagem e, logicamente, quanto mais próximos de 100%, melhores serão os resultados de predição de classificação do método utilizado. Contudo, dificilmente um método de classificação resultará em proporções sem erros para acurácia, precisão e sensibilidade simultaneamente. Portanto, identificar quais parâmetros extraem uma medição mais efetiva e descritiva da predição é de extrema importância. Uma alta precisão identifica que a predição teve uma taxa alta de acerto ao classificar cada módulo de software defeituoso, uma alta sensibilidade identifica que a predição teve uma taxa alta de acerto em classificar módulos defeituosos e uma alta acurácia identifica uma alta taxa de acerto classificando cada módulo. A *F-Measure*, por ser a média harmônica entre a precisão e sensibilidade, também é dada em porcentagem, sua utilização é mais útil em comparações de modelos, uma vez que facilita a comparação entre modelos em que se obteve resultados muito diferentes para a precisão e a sensibilidade.

6. Resultados

Antes da discussão dos resultados dos métodos de ML é importante destacar as capacidades e as deficiências da base de dados utilizada, bem como descrever o relacionamento entre cada uma das variáveis independentes com a variável dependente. Primeiramente será apresentada uma descrição geral da base de dados incluindo estatísticas descritivas de cada uma das variáveis independentes (métricas de software) e teste de normalidade da distribuição dessas métricas. Por fim, uma análise uni-variada buscando entender a correlação, se existir, entre cada variável individualmente e o número de falhas em módulos de software.

6.1 Descrição da base PROMISE

A base utilizada é uma base do projeto PROMISE [Promise 2013], que possui diversos módulos de software, desenvolvidos nas linguagens JAVA e C++, detalhados com diversas métricas de software e a classificação se os módulos possuem ou não falhas. Como visto na revisão sistemática da literatura, existe uma escassez muito grande de bases de dados com módulos de software, contudo a base PROMISE é amplamente estudada e utilizada na literatura, por esses motivos foi escolhida. A distribuição dos 145 módulos de software é dada por: 85 dos módulos não possuem falha reportada e os 60 restantes apresentam falhas. A tabela da figura 3 exibe algumas estatísticas descritivas de cada uma das métricas utilizadas, incluindo a média, o desvio padrão, a variância, os valores mínimos e máximos, os quartis em 25%, 50% e 75%.

Estatísticas

Variável	N	Média	DesvPad	Variância	Mínimo	Q1	Mediana	Q3	Máximo
CBO	145	8,317	6,377	40,663	0,000	3,000	8,000	14,000	24,000
DTI	145	2,000	1,258	1,583	1,000	1,000	2,000	2,500	7,000
LCOM	145	68,72	36,89	1360,77	0,00	56,50	84,00	96,00	100,00
NOC	145	0,2138	0,6991	0,4887	0,0000	0,0000	0,0000	0,0000	5,0000
FAN IN	145	0,6345	0,6954	0,4835	0,0000	0,0000	1,0000	1,0000	3,0000
RFC	145	34,38	36,20	1310,65	0,00	10,00	28,00	44,50	222,00
WMC	145	17,42	17,45	304,47	0,00	8,00	12,00	22,00	100,00
v(g)	145	2,834	1,863	3,472	1,000	1,000	2,000	4,000	12,000
Avg Method Cmp	145	2,579	1,623	2,634	1,000	1,000	2,000	3,000	9,000
LOC	145	296,3	448,0	200718,1	1,0	43,5	162,0	322,0	2883,0

Figura 3 -Tabela estatísticas descritivas da base PROMISE.

As estatísticas demonstram de maneira geral que os projetos, de onde a base PROMISE foi baseada, contaram com um bom planejamento, uma vez que quase 60% dos módulos de *software* não possuíram nenhuma falha reportada. Ademais, métricas como a complexidade ciclomática, se mantém baixas para a grande maioria dos módulos respeitando até o terceiro quartil uma complexidade ciclomática inferior a 10, ideal para evitar complexidade desnecessária.

6.2 Exploração dos relacionamentos

Para determinar o método de análise de relacionamentos entre as variáveis independentes (métricas de software) e a variável dependente (possui ou não falha) é preciso identificar o tipo de distribuição em que cada variável se encaixa, assim partindo de um nível de significância (α) igual a 5%, têm-se então as hipóteses:

- **Hipótese nula (H_0)**: A distribuição de cada métrica em questão é semelhante à distribuição normal;
- **Hipótese alternativa (H_1)**: A distribuição de cada métrica em questão não é semelhante à distribuição normal.

Como todas as métricas possuem um número de elementos superior à 50, utiliza-se o teste de Kolmogorov-Smirnov [Araújo et. al. 2006] para a checagem de normalidade de todas as métricas (Figura 4). Assim, como todas a métricas resultaram em uma significância inferior à 0,05 ($Sig. < 0{,}05$) rejeita-se a hipótese nula (H_0) para todas as variáveis. Sendo assim, adotada a hipótese alternativa (H_1). Dessa forma é possível afirmar que as todas as variáveis CBO, DTI, LCOM, NOC, RFC, FAN IN, WMC, v(g), Avg Method Cmp. e LOC não seguem uma distribuição normal.

Testes de Normalidade

	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Estatística	gl	Sig.	Estatística	gl	Sig.
CBO	,126	145	,000	,939	145	,000
DTI	,256	145	,000	,776	145	,000
LCOM	,239	145	,000	,742	145	,000
NOC	,496	145	,000	,348	145	,000
RFC	,177	145	,000	,738	145	,000
FAN IN	,288	145	,000	,751	145	,000
WMC	,169	145	,000	,673	145	,000
v(g)	,225	145	,000	,836	145	,000
Avg Method Cmp	,246	145	,000	,837	145	,000
LOC	,255	145	,000	,611	145	,000

a. Correlação de Significância de Lilliefors

Figura 4. Tabela testes de normalidade para as métricas de software.

O método mais adequado para a exploração de relacionamentos, quando a distribuição dos dados não é normal é o método da correlação de postos de Spearman [Araújo et. al. 2006]. Assim, as duas hipóteses para cada uma das correlações de métricas são dadas genericamente por:

- **Hipótese nula (H_0)**: Não há correlação estatística significante entre a métrica em questão e as falhas de *software* no módulo;
- **Hipótese alternativa (H_1)**: Há correlação estatística significante entre a métrica em questão e as falhas de *software* no módulo.

Dessa forma, utilizado um nível de significância (α) igual a 5% é possível concluir que para as métricas de *software* CBO, LOC, Avg Method Cmp, v(g), WMC e RFC a hipótese nula (H_0) é rejeitada e a hipótese alternativa (H_1) é aceita devido ao Valor-p de cada uma ser inferior à 0,05 (0,00; 0,00; 0,00; 0,00; 0,001 e 0,016 respectivamente) (Figura 5), assim possuindo correlação estatistica entre essas métricas e falhas de software. Destacando-se as métricas CBO, LOC, Avg Method Cmp e v(g) que apresentam significância estatística a um Valor-p inferior a 0,001 ($p<0,001$). Já para as métricas NOC, FAN IN, LCOM e DTI a hipótese nula (H_0) é aceita devido ao Valor-p de cada uma ser superior à 0,05 (0,076; 0,452; 0,477 e 0,849 respectivamente), assim não existindo correlação estatística direta entre essas métricas e falhas de software.

Correlações de Spearman pareadas

Amostra 1	Amostra 2	N	Correlação	IC de 95% para ρ	Valor-p
CBO	Defect	145	0,524	(0,386; 0,640)	0,000
DTI	Defect	145	0,016	(-0,147; 0,178)	0,849
LCOM	Defect	145	-0,060	(-0,221; 0,105)	0,477
NOC	Defect	145	-0,148	(-0,304; 0,016)	0,076
FAN_IN	Defect	145	0,063	(-0,101; 0,224)	0,452
RFC	Defect	145	0,201	(0,037; 0,354)	0,016
WMC	Defect	145	0,281	(0,121; 0,428)	0,001
v(g)	Defect	145	0,447	(0,299; 0,574)	0,000
Avg Method Cmp.	Defect	145	0,457	(0,310; 0,582)	0,000
LOC	Defect	145	0,548	(0,413; 0,659)	0,000

Figura 5. Testes de correlação de Spearman paras as métricas de software.

6.3 Resultados dos modelos de Machine Learning

Esta pesquisa buscou demonstrar a eficiência da PG em classificar módulos de *software* defeituosos, através de métricas de *software*, para isso o modelo de PG foi comparado com o modelo de RF, que como visto na revisão sistemática da literatura apresenta os melhores resultados para esta problemática. Assim, construindo ambos os modelos utilizando as bibliotecas *opensource* "Scikit-learn" e a "gplearn" foram obtidos os resultados detalhados nas matrizes de confusão conforme figura (6).

Programação genética			
	Predição de classificação do módulo		
		Com falha	Sem falha
Classificação real do módulo	Com falha	202	18
	Sem falha	66	84

Random Forest			
	Predição de classificação do módulo		
		Com falha	Sem falha
Classificação real do módulo	Com falha	161	59
	Sem falha	17	133

Figura 6: Matrizes de confusão para os modelos PG e RF

As matrizes de confusão apontam para o método utilizando PG possuindo uma precisão de 91,818%, uma sensibilidade de 75,537% e uma acurácia de 77,297%; e para o método utilizando o *Randon Forest* (RF) uma precisão de 73,181%, uma sensibilidade de 90,449% e uma acurácia de 79,459%.

Para afirmar que os resultados possuem diferença estatisticamente significantes é necessário a utilização de testes de hipótese. Para tal será utilizado um nível de significância (α) igual a 5%. Para todos os testes de hipóteses será utilizado o valor dado pelo Teste Exato de Fisher [Upton 1992]. As hipóteses para cada teste são dadas genericamente por:

- **Hipótese nula (H_0)**: Não há diferença estatística significante entre a medida do parâmetro entre o modelo PG e o modelo RF;
- **Hipótese alternativa (H_1)**: Há diferença estatística significante entre a medida do parâmetro entre o modelo PG e o modelo RF.

Dentro de testes estatísticos de hipóteses um erro do Tipo I é a rejeição da hipótese nula verdadeira resultando no retorno um falso-positivo. Em outras palavras, é erroneamente inferir que a existência de um fenômeno que não existe de fato. Não é possível eliminar completamente a probabilidade de ocorrer erros do tipo I em testes de hipóteses [Miller et al 1997]. Contudo, é possível a minimização desses erros, visando dessa forma priorizar a inferência de um módulo possuir falha mesmo que não o possua ao invés de não inferir que o modulo possua falha, mas de fato possua [Miller et al 1997]. Deste modo todos os testes estatísticos de hipótese a seguir foram priorizados a minimização dos erros de tipo I, dentro dos níveis de significância estabelecidos.

6.4 Teste de hipótese para acurácia

A acurácia (Ac) de uma classificação é definida pela divisão do total predições corretas pelo total de módulos classificados na predição, dessa forma para o modelo utilizando PG a acurácia é $Ac_{PG} = \frac{286}{370}$ e para o modelo utilizando RF é $Ac_{RF} = \frac{294}{370}$. Como o valor do *p-value* para o Teste Exato de Fisher é maior do que 0,05 (p=0,532; p>0,05) não há significância estatística, assim aceita-se a hipótese nula (H_0) e rejeita-se a hipótese alternativa (H_1). Concluindo que o resultado do método PG teve estatisticamente uma acurácia semelhante ao método *Random Forest*.

6.5 Teste de hipótese para precisão

A precisão (Pr) de uma classificação é definida pela divisão do total Verdadeiros-positivos (TP) pela soma dos Verdadeiros-positivos com os Falsos-positivos (FP), dessa forma para o modelo utilizando PG a precisão é $\mathrm{Pr_{PG}} = \frac{202}{220}$ e para o modelo utilizando RF é $\mathrm{Pr_{RF}} = \frac{161}{220}$. Como o valor do *p-value* para o Teste Exato de Fisher é menor do que 0,05 (p=0,00; p<0,05) há significância estatística, assim rejeita-se a hipótese nula (H_0) e aceita-se a hipótese alternativa (H_1). Concluindo que o resultado do método utilizando PG teve estatisticamente uma precisão superior ao método utilizando *Random Forest*.

6.6 Teste de hipótese para sensibilidade

A sensibilidade, ou em inglês, *recall* (Re) de uma classificação é definida pela divisão do total Verdadeiros-positivos (TP) pela soma dos Verdadeiros-positivos mais os Falsos-Negativos (FN), dessa forma para o modelo utilizando PG a sensibilidade é $Re_{PG} = \frac{202}{268}$ e para o modelo utilizando RF é $Re_{RF} = \frac{161}{178}$. Como o valor do *p-value* para o Teste Exato de Fisher é menor do que 0,05 (p=0,00; p<0,05) há significância estatística, assim rejeita-se a hipótese nula (H_0) e aceita-se a hipótese alternativa (H_1). Concluindo que o resultado do método utilizando *Random Forest* teve estatisticamente uma sensibilidade superior ao método utilizando PG.

6.7 Teste de hipótese para *F-Measure*

A *F-Measure* (Fm), de uma classificação é definida pela média harmônica entre a precisão (Pr) e a sensibilidade (Re). Como o valor do *p-value* para o Teste Exato de Fisher é maior do que 0,05 (p=0,325; p>0,05) não há significância estatística, assim aceita-se a hipótese nula (H_0) e rejeita-se a hipótese alternativa (H_1). Concluindo que o resultado do método PG teve estatisticamente uma *F-measure* semelhante ao método *Random Forest*.

7. Conclusões

Durante a construção deste trabalho, foi possível constatar a ampla variedade de empregabilidade de técnicas de ML em identificação de módulos de software defeituosos. Essa é uma área que expande e evolui a passos largos somando isso ao impacto positivo que os resultados podem trazer à projetos de software em geral, evidenciam que é uma área com futuro promissor.

A base de dados PROMISE é uma base extremamente utilizada na literatura, e é disponibilizada abertamente, o que a torna uma excelente base para a reprodutibilidade de publicações. Ademais é uma base com uma quantidade razoável de total de amostras, além de possuir um tratamento prévio, evitando dados faltantes e dados irrelevantes. Apesar disto, a base apresenta uma questão substancialmente prejudicial, que é apenas reunir dados de módulos de software trabalhados nas linguagens JAVA e C++. Uma maior diversidade de linguagens traria uma maior confiança para extrapolações para casos mais diversos, além de possivelmente diminuir o impacto que a arquitetura de uma linguagem tem sobre as métricas produzidas durante o desenvolvimento.

Conforme visto na literatura, as métricas de software CBO, RFC, LOC e WMC se mostraram extremamente correlacionadas com a tendência a falha em módulos de software. Além dessas, a complexidade ciclomática e a complexidade média de métodos também apresentaram significância correlacional com o aumento do número de erros. Demonstrando assim que a utilização de métricas de software na predição de tendência a falha é lógica e justificada. Tal fato somente reforça que a atenção para essas métricas de software pode ajudar bastante no desenvolvimento de sistemas.

Neste trabalho foi proposta uma implementação de algoritmos evolutivos, focando na PG, para predizer tendência a falha em módulos de software, utilizando métricas de software como base de treino. Diversos trabalhos da literatura apontam para a deficiência de trabalhos envolvendo algoritmos evolutivos [Singh et al. 2018], e neste trabalho não somente um modelo envolvendo algoritmos evolutivos foi implementado como seus resultados se mostraram afins de técnicas de estado-da-arte da literatura, mais especificamente o modelo *Random Forest*. Apesar de apresentarem resultados estatisticamente semelhantes dois parâmetros estão se destacaram, a sensibilidade do RF e a precisão da PG. O RF por possuir uma sensibilidade maior apresenta uma distribuição mais favorável ao reconhecimento de módulos defeituosos, ou seja, uma maior quantidade de módulos defeituosos é assinalada como defeituosos. Ao passo que a PG por possuir uma precisão maior apresenta um aproveitamento da classificação da classificação como um módulo contendo falhas, ou seja, uma maior quantidade de módulos classificados como defeituosos são de fato defeituosos. Mas como discutido, comparando os modelos pela *F-measure* a semelhança dos resultados é evidenciada demonstrando assim que ambas abordagens podem ser validas para ajudar na otimização da utilização dos recursos.

Para trabalhos futuros, a coleta e criação de uma base de dados maior e mais diversa, envolvendo linguagens diferentes, desenvolvedores diferentes, a utilização de métricas diferentes e de fontes diferentes como *open-source* e códigos proprietário se mostram como o melhor curso de ação, essa coleta não se trata de uma simples tarefa [Gondra 2008] mas certamente deverá trazer melhores resultados para a predição de módulos de software defeituosos. Além disso, a grande maioria das publicações tratam a

tendência a falha como uma tarefa de classificação básica entre “o módulo possui falha” e “o módulo não possui falha”, assim a evolução para um problema de classificação um pouco mais complexo, considerando a severidade e o número de erros pode trazer resultados interessantes para discussão.

Referências

Araújo, M. A., et al. "Métodos estatísticos aplicados em Engenharia de Software experimental." XXI SBBD-XX SBES (2006).

Askari, Mohamad Mahdi, and Vahid Khatibi Bardsiri. "Software defect prediction using a high performance neural network." International Journal of Software Engineering and Its Applications 8.12 (2014): 177-188.

Banzhaf, Wolfgang, et al. Genetic programming: an introduction. Vol. 1. San Francisco: Morgan Kaufmann Publishers, 1998.

Basili, Victor R., and Barry T. Perricone. "Software errors and complexity: an empirical investigation0." Communications of the ACM 27.1 (1984): 42-52.

Breiman, Leo. "Random forests." Machine learning 45.1 (2001): 5-32.

Charniak, Eugene. *Introduction to artificial intelligence*. Pearson Education India, 1985.

Chidamber, Shyam R., and Chris F. Kemerer. "A metrics suite for object oriented design." IEEE Transactions on software engineering 20.6 (1994): 476-493.

De Carvalho, Andre B., Aurora Pozo, and Silvia Regina Vergilio. "A symbolic fault-prediction model based on multiobjective particle swarm optimization." Journal of Systems and Software 83.5 (2010): 868-882.

Fenton, Norman E., and Martin Neil. "Software metrics: roadmap." Proceedings of the Conference on the Future of Software Engineering. 2000.

Gondra, Iker. "Applying machine learning to software fault-proneness prediction." Journal of Systems and Software 81.2 (2008): 186-195.

Goodwill, James, and Greg H. Pearman. Pro. NET 2.0 Extreme Programming. Springer, 2006.

Guo, Lan, et al. "Robust prediction of fault-proneness by random forests." 15th international symposium on software reliability engineering. IEEE, 2004.

Harrison, Rachel, Steve J. Counsell, and Reuben V. Nithi. "An evaluation of the MOOD set of object-oriented software metrics." IEEE Transactions on Software Engineering 24.6 (1998): 491-496.

Hammouri, A., Hammad, M., Alnabhan, M., & Alsarayrah, F. (2018). Software bug prediction using machine learning approach. International Journal of Advanced Computer Science and Applications, 9(2), 78-83.

ISO/IEC/IEEE International Standard. 2013. “Software and systems engineering -- Software testing --Part 3: Test documentation," in ISO/IEC/IEEE 29119-3:2013(E), pp.1- 138. IEEE

Isong, Bassey, and Ekabua Obeten. "A systematic review of the empirical validation of object-oriented metrics towards fault-proneness prediction." International Journal of Software Engineering and Knowledge Engineering 23.10 (2013): 1513-1540.

Kaur, Arvinder, and Ruchika Malhotra. "Application of random forest in predicting fault-prone classes." 2008 International Conference on Advanced Computer Theory and Engineering. IEEE, 2008.

Koza, John R. "Non-linear genetic algorithms for solving problems." U.S. Patent No. 4,935,877. 19 Jun. 1990.

Malhotra, Ruchika, and Ankita Jain. "Fault prediction using statistical and machine learning methods for improving software quality." Journal of Information Processing Systems 8.2 (2012): 241-262.

Miller, James et al. Statistical power and its subcomponents—missing and misunderstood concepts in empirical software engineering research. Information and Software Technology, v. 39, n. 4, p. 285-295, 1997.

Murphy, Kevin P. Machine learning: a probabilistic perspective. MIT press, 2012.

Naik, Kshirasagar, and Priyadarshi Tripathy. Software testing and quality assurance: theory and practice. John Wiley & Sons, 2011.

Pedregosa, Fabian, et al. "Scikit-learn: Machine learning in Python." the Journal of machine Learning research 12 (2011): 2825-2830.

Promise Software Engineering Repository Public Datasets (2013). <http:// promise.site.uottawa.ca/SERepository/datasets/cm1.arff>

Shatnawi, Raed. "Empirical study of fault prediction for open-source systems using the Chidamber and Kemerer metrics." *IET software* 8.3 (2013): 113-119.

Singh, Ajmer, Rajesh Bhatia, and Anita Singhrova. "Taxonomy of machine learning algorithms in software fault prediction using object oriented metrics." Procedia computer science 132 (2018): 993-1001.

Stephens, T. "Gplearn Model, Genetic Programming." (2015).

Upton, Graham JG. "Fisher's exact test." Journal of the Royal Statistical Society: Series A (Statistics in Society) 155.3 (1992): 395-402.

Wohlin, Claes. "Guidelines for snowballing in systematic literature studies and a replication in software engineering." Proceedings of the 18th international conference on evaluation and assessment in software engineering. 2014.