Predição Just-In-Time de Defeitos em Software Utilizando Inteligência Artificial

Ismael Araújo Ramos[1], Márcio André Baima Amora[1]

[1]Programa de Pós Graduação em Engenharia Elétrica e Computação – PPGEEC
Campus de Sobral, Universidade Federal do Ceará – (UFC), Sobral - CE,
CEP 62.010-560, Brasil

ismaelramos@alu.ufc.br, marcio@dee.ufc.br

Abstract. *In the development or modification of a software, the software must have least amount of possible errors. Methods of predicting defects in software could be used for this. In this paper we present a study of the use of Just-In-Time (JIT) error identification using Artificial Neural Network (ANN) and decision tree (DT). The databases used as training, test and validation in this work were the same ones used and compiled by [Kamei et al. 2013]. The results obtained with both, ANN and DT, are on average higher than the work of [Kamei et al. 2013] and [Yang et al. 2017].*

Resumo. *Durante o desenvolvimento ou modificação de um software, deve ser garantido que o produto final chegue ao usuário com a menor quantidade de erros possíveis. Métodos de predição de defeitos em software podem ser usados para isso. Neste artigo é apresentado um estudo utilizando, para a solução do problema de identificação de erros Just-In-Time (JIT), rede neural artificial (Artificial Neural Network - ANN) e árvore de decisão (Decision Tree - DT). As bases de dados utilizadas como treino, teste e validação neste trabalho foram as mesmas utilizadas e compiladas por [Kamei et al. 2013]. Os resultados obtidos, tanto com a ANN e com a DT são em média superiores aos trabalhos de [Kamei et al. 2013] e [Yang et al. 2017].*

1. Introdução

A qualidade de um produto depende da qualidade em todo o seu processo de fabricação. Para o software não é diferente. O teste de software busca garantir que o produto final chegará ao seu destino com a menor quantidade de erros possíveis. O ponto chave é realizar testes eficientes no software, alocando o menor número de recursos possíveis e gastando o menor tempo possível. Os métodos de predição de defeitos podem ser usados para otimização desse processo. Sua atuação é no apontamento de áreas onde podem haver erros. Assim, a finalidade deste trabalho seria o estudo e a proposta de um modelo de predição de defeitos em software. Esse modelo poderá ser usado alocando recursos de forma inteligente, garantindo a qualidade da manutenção e consequentemente do produto final (software) mas, ao mesmo tempo, possibilitando a redução do uso dos recursos, e, portanto, dos custos.

Vários autores na literatura apresentam estudos e métodos relacionados com a predição de defeitos em softwares. Alguns desses métodos serão comentados na seção 2. Em geral, os autores utilizam métodos de predição baseados na análise dos arquivos do software desenvolvido, ou então baseadas na análise em tempo de execução, ou seja, *Just-In-Time* (JIT), que analisa diretamente as alterações realizadas no software para

verificar a possibilidade da indução de erros.

A abordagem de identificação de defeitos em software baseada na análise direta de arquivos, resulta, em muitos casos, em grande esforço e muito tempo envolvido, pois pode haver a necessidade da verificação de todos arquivos que formam o software em depuração. Por sua vez, analises de predição JIT são realizadas quando ocorre alguma mudança no código, podendo ser classes, métodos e etc, indicando as áreas do código com maior probabilidade de ter erros [Yang 2015]. Portanto, as analises JIT apresentam vantagens quanto ao número de arquivos analisados, que normalmente é menor, e identificação imediata de áreas do software propensas a erros.

Nos métodos desenvolvidos neste artigo para predição de defeitos em software e baseados em JIT, foram utilizados como métricas para análise no diagnóstico de erros: informações sobre os arquivos modificados no software e informações sobre os programadores. Essas métricas são as mesmas utilizadas em [Kamei *et al.* 2013] e [Yang *et al.* 2017]. Os bancos de dados de teste utilizados para validação e comparação dos resultados são também os utilizados por [Kamei *et al.* 2013] e [Yang *et al.* 2017]. Esses bancos de dados são relacionados aos seguintes projetos de software livre: Bugzilla, Columba, JDT, Mozilla, Platform e Postgresql. Os resultados obtidos dos métodos desenvolvidos e descritos neste artigo são em média melhores que os indicados nas referências utilizadas como comparação: [Kamei *et al.* 2013] e [Yang *et al.* 2017], que utilizam no diagnóstico de falhas em software, as técnicas árvore de decisão (*Decision Tree - DT*) e floresta de decisão (*Random Forest* - RD), respectivamente. Este artigo, descreve os resultados obtidos no desenvolvimento de métodos de predição JIT utilizando técnicas de IA (Inteligência Artificial): adotando rede neural artificial (*Artificial Neural Network - ANN*) e DT. A DT desenvolvida neste artigo utiliza modificações no tratamento de dados, em comparação com [Kamei *et al.* 2013] que também adota a mesma técnica.

O artigo está divido em seções, citadas a seguir. Na seção 2 é apresentada uma breve revisão bibliográfica de métodos de predição de defeitos em softwares. Na seção 3 é apresentada uma breve descrição sobre diagnósticos de falhas em softwares, utilizando JIT. Na seção 4 são fornecidos conceitos de redes neurais artificiais. Na seção 5, são comentados os conceitos sobre árvores de decisão. Os detalhes da metodologia executada e dos bancos de dados utilizados neste trabalho são apresentados na seção 6. Os resultados obtidos da ANN e da DT desenvolvidas são discutidos na seção 7. Na seção 8 são apresentadas as conclusões deste trabalho e as perspectivas futuras da pesquisa.

2. Revisão Bibliográfica

A seguir, vários estudos de predição de defeitos em softwares, e baseados na análise direta de arquivos, são comentados. Em [Okutan e Yldz 2014], algumas métricas para previsão de defeitos em softwares são analisadas, tanto métricas relacionadas ao código em si, como também métricas relacionadas a forma de desenvolvimento (metodologia escolhida, distribuição de tarefas, análise de requisitos e etc.). Os autores concluem afirmando que das métricas analisadas, as que foram mais efetivas na previsão de defeitos foram: resposta por classe, linhas de código, e falta de qualidade do código. Já em [Arar e Ayan 2015], é proposta uma predição de defeito de software, em nível de arquivos utilizando uma combinação de ANN e colônia artificial de abelhas (*Artificial Bee Colony* - ABC). O método ABC é usado para encontrar os pesos ideais da ANN. Os

autores em [Jindal, Malhotra e Jain 2014] propõem um modelo que usa técnicas de mineração de texto para atribuir um certo nível de severidade a cada relatório de defeito do software, com essas informações alimentando um método de aprendizado de máquina que utiliza uma rede de funções de base radial (*Radial Basis Function* - RBF).

No estudo de [Li *et al.* 2017], é proposto um método de diagnóstico de erros em software baseado em uma rede neural convolucional (*Convolutional Neural Network* - CNN), construído em três etapas. Primeiro, são extraídas informações do software analisado. Em segundo lugar, uma rede CNN é treinada para aprender automaticamente os recursos semânticos e estruturais do software. E, finalmente, os autores combinam os recursos aprendidos com recursos obtidos manualmente, para prever com maior precisão os defeitos de software. Em [Fenton e Neil 1999], um estudo crítico dos modelos de predição de defeitos é realizado. Os autores argumentam que há problemas de estatística e qualidade nos dados que prejudicam a validade dos modelos. No mesmo estudo, é proposto um modelo de análise utilizando uma rede bayesiana (*Bayesian Belief Network* - BBF).

Em relação a métodos que utilizam uma abordagem de identificação de erros JIT. Em [Kamei *et al*. 2013], os autores utilizam JIT através de uma DT, sendo obtidos resultados de recuperação de um pouco mais de 70% em média, analisando apenas as partes do software que sofreram alteração, cerca de 20% das linhas de código. Já em [Yang *et al*. 2017] é utilizado uma RD, também numa abordagem JIT, obtendo uma recuperação em média de quase 76%.

3. Diagnóstico de Falhas Baseados em JIT

JIT é uma técnica baseada na análise instantânea do conjunto de mudanças realizadas no software (*commits)*. Faz-se simplesmente uma análise do arquivo envolvido no *commit* e é avaliado se ele têm ou não possibilidade de defeitos. Numa análise desse tipo, existem quatro etapas, como descrito em [Kamei *et al.* 2013]:

1. Identificação de rótulo para os dados de treinamento: Para cada alteração, é definido um rótulo como tendo erro ou não, explorando o histórico de revisão do projeto e o sistema de rastreamento de erros.

2. Obtenção dos valores para as métricas adotadas: São extraídos valores para as métricas envolvidas com as alterações realizadas no software. Neste trabalho são adotadas as métricas indicadas na Tabela 1, envolvendo as seguintes dimensões: a difusão da mudança (que representa o número de diretórios, subsistemas e arquivos que uma alteração envolve), o tamanho da alteração (que representam os números de linhas do código envolvidos em uma alteração, e a entropia da modificação), o objetivo da alteração (se é uma correção de defeito ou não), informações sobre o histórico de mudanças, e informações sobre a experiência dos programadores envolvidos.

3. Modelo de aprendizagem: Construir um modelo de *machine learning* baseado em mudanças realizadas no software. Nessa etapa, dependendo da técnica escolhida, haverá uma etapa de treinamento do modelo desenvolvido, utilizando um conjunto de *commits* com métricas e rótulo previamente conhecidos (etapa 1).

4. Aplicação da técnica escolhida: Após treinado, o produto gerado pela técnica de *machine learning* escolhido é executado em casos de alterações em softwares, gerando uma previsão de ocorrência de erros ou não.

Nos estudos realizados nesta pesquisa, foram adotadas como técnicas de aprendizagem: ANN e DT. Essas técnicas serão comentadas nas seções seguintes. Os bancos de dados utilizados como teste neste trabalho são comentados com mais detalhes na seção 6.

Tabela 1: Descrição das Métricas

Dimensão	**Nome**	**Definição**	**Fundamentação [Kamei *et al.* 2013]**
Difusão	NS	Número de subsistemas modificados.	Alterações que modificam muitos subsistemas são mais propensas a serem defeituosas.
Difusão	ND	Número de diretórios modificados.	Alterações que modificam muitos diretórios são mais propensas a serem defeituosas.
Difusão	NF	Número de arquivos modificados.	As mudanças que afetam muitos arquivos são mais propensas a serem defeituosas.
Difusão	ENTROPIA	Distribuição do código modificado dentro dos arquivos.	Mudanças com entropia alta são mais propensas a apresentarem defeitos, porque um desenvolvedor terá que lembrar e rastrear um grande número de mudanças espalhadas em cada arquivo.
Tamanho	LA	Linhas adicionadas.	Quanto mais linhas de código se adiciona nos arquivos, mais chances de se introduzir um defeito.
Tamanho	LD	Linhas deletadas.	Quanto mais linhas se exclui de um arquivo, mais chances de se introduzir um defeito.
Tamanho	LT	Linhas totais.	Quanto maior for o arquivo, mais chances de se introduzir um defeito.
Propósito	FIX	Determina se a mudança é ou não para corrigir um defeito.	Corrigir um defeito significa que um erro foi cometido em uma implementação anterior; portanto, pode indicar uma área em que os erros são mais prováveis.
Histórico	NDEV	Número de desenvolvedores que modificaram o arquivo.	Quanto maior o NDEV, maior a probabilidade de um defeito ser introduzido, porque os arquivos revisados por muitos desenvolvedores geralmente contêm pensamentos de design e estilos de codificação diferentes.
Histórico	AGE	Intervalo de tempo médio entre a última modificação e a atual.	Quanto menor o AGE, ou seja, quanto mais recente a última alteração, maior a probabilidade de um defeito ser introduzido.
Histórico	NUC	Número de alterações únicas.	Quanto maior o NUC, maior a probabilidade de um defeito ser introduzido, porque um desenvolvedor terá que recuperar e rastrear muitas alterações anteriores.
Experiência	EXP	Experiência do desenvolvedor.	Quanto mais experiência um desenvolvedor tiver no sistema, menos chances de introduzir um defeito.
Experiência	REXP	Experiência recente do Desenvolvedor.	Um desenvolvedor que freqüentemente modificou os arquivos nos últimos meses tem menos probabilidade de introduzir um defeito, porque ele estará mais familiarizado com os desenvolvimentos recentes no sistema.
Experiência	SEXP	Experiência do desenvolvedor no subsistema.	O desenvolvedor familiarizado com os subsistemas modificados por uma alteração tem menor probabilidade de introduzir um defeito.

Fonte: Adaptado pelos autores a partir de [Kamei *et al.* 2013].

4. Rede Neural Artificial

Uma ANN é um modelo de inteligência computacional baseado no cérebro humano. Na técnica, os neurônios são um conjunto de unidades que operam como chaves de processamento, transmitindo informações quando um limiar de operação é alcançado.

Uma ANN do tipo MLP (*MultiLayer Perceptron*) [Haykin 2001] possui pelo menos três camadas (Figura 1): a camada de entrada, a camada escondida que podem ser múltiplas, e a camada de saída. A propagação dos dados, ou sinal de entrada, é na ordem direta desde a entrada até a saída da rede, camada a camada. As conexões entre os neurônios são representadas por pesos. Os pesos indicam a força ou importância de cada conexão do neurônio. O aprendizado da rede é baseado nos ajustes iterados dos pesos e dos valores de *bias* nos neurônios [Arar e Ayan 2015] [Gayathri e Sudha 2014].

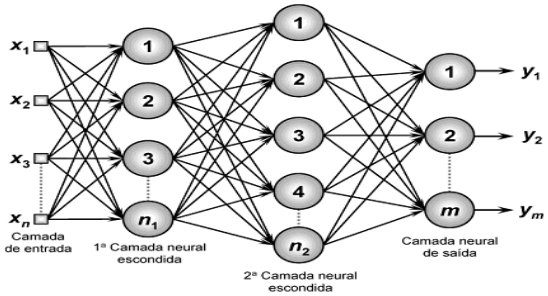

Figura 1: Exemplo de Uma ANN - MLP
Fonte: Batista, (2012).

Em (1) é demonstrado o modelo de cálculo para um neurônio artificial [Arar e Ayan 2015]. Onde y_i é a saída do neurônio i, n é o número total de entradas para este neurônio provenientes das entradas da ANN ou então provenientes das saídas de uma camada de neurônios anteriores, x_j é a j-ésima entrada, w_{ij} é o peso da conexão entre o neurônio i e a j-ésima entrada, Θ_i é o valor de tendência (*bias*) do neurônio e f_i representa a função de ativação do neurônio. Um exemplo de função de ativação pode ser observado em (2), que representa a equação de uma função do tipo tangente hiperbólica:

$$y_i = f_i\left(\sum_{j=1}^{n} \mathrm{x_j}\, w_{ij} + \theta_i\right), \tag{1}$$

$$\tanh(y) = u(y) = \frac{senh(y)}{\cosh(y)} = \frac{e^{y}-e^{-y}}{e^{y}+e^{-y}}\,. \tag{2}$$

No processo de treinamento de uma ANN são utilizados valores de treinamento com dados de entrada e saída conhecidos, formando um conjunto de treinamento. Nesse treinamento da rede é utilizado, normalmente, algum método iterativo e supervisionado como, por exemplo, o algoritmo *Backpropagation* [Haykin 2001], para a determinação dos valores dos pesos e bias da ANN. Nas simulações de ANN realizadas neste trabalho foram utilizadas funções do programa de simulação matemática MATLAB.

5. Árvore de Decisão

Uma DT é um modelo preditivo que pode ser utilizando para representar tanto um modelo de classificação como também um modelo de regressão [Rokach e Maimon 2008]. Quando uma DT é utilizada como classificador é normalmente denominada Árvore de Classificação (AC) e quando utilizada para regressão como Árvore de

Regressão (AR).

Uma AC é utilizada para classificar um objeto ou instância dentro de um conjunto pré-definido de classes, baseados nos atributos da instância. As árvores de classificação são frequentemente utilizadas em problemas nas áreas de finanças, marketing, engenharia e medicina [Rokach e Maimon 2008]. Uma AC representa um sistema de decisão multiestágios onde as classes são sequencialmente rejeitadas até ser alcançada uma classe final de aceitação, durante a apresentação de uma instância (vetor de atributos) a ser classificada. No final, o espaço de entrada é dividido em regiões distintas, correspondendo às classes, de maneira sequencial. Durante a apresentação de um vetor para classificação, a pesquisa da região a ser associada a um parâmetro do vetor é obtida através da pesquisa de uma sequência de decisões ao longo de um caminho de nós, numa árvore apropriadamente construída [Theodoridis e Koutroumbas 2006].

Na Figura 2 é exemplificado uma DT tipo AC típica, separando o espaço de entradas em hiperplanos com retas paralelas aos eixos. A sequência de decisões é aplicada para cada atributo da instância apresentada à árvore, com os testes de decisão associados aos nós sendo na forma:

$$Se\ x_i\ \theta\ \alpha\ Então\ c_1\ Senão\ c_2, \tag{3}$$

onde x_i representa o atributo avaliado; θ a operação lógica testada ($=, \neq, \leq, \geq, <, >$); α é um valor limite; e c_1 e c_2 representam "caminhos" distintos na árvore que levam a outros nós na árvore que podem representar um outro nó de teste ou então um nó de "folha" que representa uma classe de classificação.

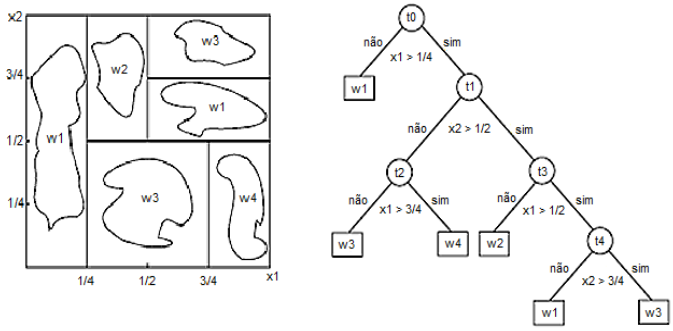

Figura 2: Partição do espaço de variáveis e regras obtidas da AC
Fonte: Theodoridis e Koutroumbas (2006).

Existem vários algoritmos já conhecidos e amplamente utilizados para construção de uma DT, entre eles podemos citar o C4.5 [Quinlan 2014] e o CART [Breiman 1996]. Em [Kamei *et al.* 2013] é utilizada uma DT para previsão de defeitos em softwares. Nas simulações de DT realizadas neste trabalho foi adotado o algoritmo CART, utilizando métodos da API sklearnig do Python versão 0.19.2 [Scikit-Learn 2019a].

6. Metodologia e Bancos de Dados Utilizados

Os bancos de dados adotados neste trabalho: Bugzilla, Columba, JDT, Mozilla, Platform e Postgresql, podem ser obtidos e verificados em [Kamei 2019] e foram os mesmo utilizados em [Kamei *et al.* 2013] e [Yang *et al.* 2017]. Esses bancos representam casos de *commits*, que por sua vez representam casos de alterações

realizadas nos softwares relacionados. Cada banco de dados é vinculado a um software de desenvolvimento aberto.

Todos os bancos de dados apresentam 14 parâmetros (métricas) de entrada para cada *commit*, e apresentam um diagnóstico com detecção de erro ou não. Os parâmetros são comentados na Tabela 1. Alguns desses parâmetros de entrada indicam métricas focadas na qualidade do código fonte do software, outras métricas são focadas no processo de desenvolvimento. Essas métricas são baseadas no trabalho de [Kamei *et al.* 2013] e, também, utilizadas por [Yang *et al.* 2017].

A Tabelas 2 descreve o total de *commits* em todas as bases de dados utilizadas neste trabalho, informando os casos com ocorrência de erros e os com ausência. No total são 227417 *commits*, sendo que destes apenas 27876 são *erros*, o que corresponde aproximadamente a 12,25% do total de dados.

Para a realização das simulações de teste para o diagnóstico preditivo de falhas em software, inicialmente, todos casos de erros foram separados para serem utilizados num processo de validação cruzada tipo 10-*fold*, numa tentativa de minimizar situações de *overfitting*. Feito isto, selecionou-se aleatoriamente de todas as bases juntas uma quantia igual para casos de *commits* que eram considerados sem erro. Essa abordagem também é utilizada em [Kamei *et al.* 2013].

Num passo seguinte, os métodos de IA (ANN ou DT) utilizados neste trabalho para o diagnóstico foram implementado utilizando os dados de validação cruzada para o treinamento, teste e validação dos métodos.

Tabela 2: Bases de Dados: Geral de *commits*, porcentagem de erros ou não

Base	**Erro (%)**	**Não Erro (%)**	**Total (*commits*)**
Bugzilla	36,71	63,29	4620
Columba	30,55	69,45	4455
JDT	14,38	85,62	35386
Mozilla	5,24	94,76	98275
Platform	14,71	85,29	64250
Postgresql	25,05	74,95	20431

Fonte: Adaptado de Kamei *et at.* (2013)

Uma normalização individual foi realizada em cada um dos parâmetros de entrada para a rede neural MLP desenvolvida neste trabalho:

1. Para todos os parâmetros de entrada, o valor de um parâmetro especifico é normalizado em função do maior valor encontrado para este parâmetro especifico nos casos de treinamento e teste.
2. Semelhante a [Kamei *et al.* 2013], os valores de LA e LD (Tabela 1) foram divididos por LT. Também, os parâmetros LT e NUC foram divididos por NF.

Na utilização da DT para a previsão de defeitos, além da etapa 2 descrita acima, foi utilizada a normalização padrão da biblioteca *scikit learn* da linguagem Python, conhecida por *Standard Scaler* [Scikit-Learn 2019b]; como pode ser visto em 4. Essa

nova normalização na DT é utilizada, porém, sem balancear os dados entre casos sem erro e com erro:

$$z = \frac{x-u}{s}, \tag{4}$$

sendo *z* o novo valor do atributo, *u* a média aritmética da métrica e *s* o desvio padrão.

Para análise estatística dos resultados obtidos, foram adotados os parâmetros: Exatidão, Precisão, *Recall* e a média harmônica - F1 (entre precisão e *recall*). Para calculá-los, os resultados foram primeiramente analisados e rotulados com os seguintes nomes: Verdadeiro Positivo - *TP* (casos em que o alvo se trata de um erro e o método preditivo acertou), Falso Positivo - *FP* (casos em que o alvo não se trata de um erro e a previsão disse que era erro), Verdadeiro Negativo - *TN* (casos em que as mudanças não resultaram em erro e o método utilizado acertou) e Falso Negativo - *FN* (casos em que as mudanças resultaram em erro, porém o diagnóstico informou não se tratar de erro).

A Exatidão representa a divisão entre a soma dos casos corretos, ou seja, que a predição acertou, pela soma geral de todos os casos, como pode ser representado matematicamente pela equação (5) [Kamei *et al.* 2013]:

$$Exatidão = \frac{(TP+TN)}{(TP+TN+FP+FN)}. \tag{5}$$

A Precisão é a divisão entre casos que eram erros e a previsão estava correta (TP), pela soma dos casos TP e os casos considerados como falsos positivos (FP), equação (6), [Kamei *et al.* 2013]:

$$Precisão = \frac{TP}{TP+FP}. \tag{6}$$

O *Recall* é a divisão entre casos que foram erros e a previsão estava correta (TP), pela soma dos casos TP com os casos falsos negativos (FN), equação (7), [Kamei *et al.* 2013]:

$$Recall = \frac{TP}{TP+FN}. \tag{7}$$

A média harmônica - F1 representa a média harmônica entre Precisão e *Recall,* equação (8), [Kamei *et al.* 2013]:

$$F1 = \frac{2*RECALL*PRECISÃO}{RECALL+PRECISÃO}. \tag{8}$$

Na próxima seção, são exibidos os resultados obtidos com a ANN e a DT desenvolvidas neste artigo, e comparados com os trabalhos de [Kamei *et al.* 2013] e [Yang *et al.* 2017].

7. Resultados

Para a realização do diagnóstico preditivo de erros em um software que sofre modificação, abordagem JIT, inicialmente é proposto neste trabalho uma ANN do tipo MLP com 14 parâmetros de entrada, um neurônio de saída que indica se o *software* apresenta erro ou não. Duas camadas ocultas são adotadas, com 28 neurônios cada. A função tangente hiperbólica foi escolhida como a função de ativação nos neurônios e para saída uma função linear. O algoritmo de treinamento da rede utilizado foi o método de Levenberg–Marquardt [Yu e Wilamowski 2011], implementado utilizando funções do MATLAB. Também foi desenvolvido neste trabalho uma DT para o diagnóstico de

softwares, adotando o algoritmo CART através do PYTHON. Os dados de treinamento e validação foram tratados conforme descrito na seção 6.

Na Tabela 3 são apresentados os resultados obtido por [Kamei *et al.* 2013] utilizando o mesmo banco de dados de teste adotado neste artigo, e utilizando uma DT no diagnóstico de falhas em software. Os resultados obtidos da ANN proposta neste trabalho são indicados na Tabela 4, e os resultados da DT desenvolvidas pelos autores são apresentados na Tabela 5. Para essas duas tabelas, os dados apresentados na cor azul representam valores obtidos das métricas que são melhores que os valores indicados em [Kamei *et al.* 2013] e exibidos na Tabela 3, na cor vermelha são métricas com resultados inferiores e na cor preta são resultados considerados semelhantes.

Tabela 3: Resultados DT

Base	Exatidão	Precisão	Recall	F1
Bugzilla	67%	54%	69%	60%
Columba	70%	51%	67%	58%
JDT	69%	26%	65%	37%
Mozilla	77%	13%	63%	22%
Platform	67%	27%	70%	38%
Postgresql	74%	49%	65%	56%
Média	70,67%	36,67%	66,5%	45,2%
Mediana	69,50%	38%	66%	47%
Desvio Padrão	4,03%	16,88%	2,66%	15,2%

Fonte: Adaptado pelos autores de Kamei *et al.* (2013).

Tabela 4: Resultados ANN desenvolvida

Base	Exatidão	Precisão	Recall	F1
Bugzilla	69,11%	56,02%	73,8%	63,7%
Columba	62,47%	44,05%	84,6%	58,0%
JDT	69,32%	28,04%	72,3%	40,4%
Mozilla	79,21%	16,01%	69,9%	26,0%
Platform	68,47%	28,69%	77,0%	41,8%
Postgresql	73,00%	47,38%	70,2%	56,6%
Média	70,26%	36,70%	74,6%	47,7%
Mediana	69,22%	36,37%	73,1%	49,2%
Desvio Padrão	5,55%	14,89%	5,54%	14,1%

Fonte: Os próprios autores.

Tabela 5: Resultados DT desenvolvida

Base	Exatidão	Precisão	Recall	F1
Bugzilla	83%	92%	75%	83%
Columba	84%	86%	82%	84%
JDT	77%	78%	75%	77%
Mozilla	70%	64%	77%	70%
Platform	78%	79%	76%	78%
Postgresql	84%	87%	81%	84%
Média	79%	81%	78%	79%
Mediana	80%	83%	77%	80%
Desvio Padrão	5%	9%	3%	5%

Fonte: Os próprios autores.

Comparando os resultados das tabelas 3 e 4. Os valores de *Recall* obtidos da rede neural proposta foi melhor em todas as bases de dados. Já o valor de Precisão foi melhor em quatro das bases de dados e pior em duas. A média harmônica foi melhor em quatro

bases e semelhante em duas. Exatidão foi melhor em três base de dados, pior em duas e semelhante em uma. Dos 24 valores possíveis, distribuídos em quatro métricas nas seis bases de dados, foram obtidos resultados melhores em 17 valores (marcados em azul).

Como verificação estatística, foram calculados três medidas nas tabelas: média, mediana e desvio padrão. Como pode ser observado na comparação das tabelas 3 e 4, foram obtidos valores de média e mediana melhores do *Recall* na ANN desenvolvida, mas um desvio padrão maior se comparado a [Kamei *et al.* 2013]. Foi verificado ainda que na média harmônica *F1,* é obtido uma média e mediana melhores e um desvio padrão mais baixo. Foi obtido empate na média da Precisão e um resultado ligeiramente inferior na mediana, porém o desvio padrão apresentado é menor. Já na Exatidão, foi observado empate na média e mediana e um desvio padrão pior.

Percebe-se que os resultados da nova DT proposta (Tabela 5) são bem superiores ao de [Kamei *et al.* 2013]. Comparando os resultados das Tabelas 3 e 5, os valores de *Recall*, Precisão e média harmônica F1 obtidos da DT proposta foram melhores em todas as bases de dados. A Exatidão foi melhor em cinco bases de dados e pior em apenas uma, Mozilla. Dos 24 valores possíveis, distribuídos em quatro métricas e em seis bases de dados, foram obtidos resultados melhores em 23 valores (marcados em azul). Os resultados expostos nas tabelas 4 e 5, mostram que das 24 medidas possíveis, a ANN desenvolvida só superou a DT desenvolvida em apenas 3 casos. Nas medidas estatísticas, a DT superou a ANN em todas.

Na Tabela 6 são apresentados resultados obtido por [Yang *et al.* 2017], que representa um estudo que também faz comparação com [Kamei *et al.* 2013] e utiliza os mesmos bancos de dados de teste. Já a Tabela 7 representa os mesmos resultados da DT desenvolvida nesse trabalho (Tabela 5), entretanto a codificação de cores é feita em comparação com os resultados da Tabela 6: resultados melhores na DT desenvolvida são apresentados em azul, valores semelhantes em preto e valores piores são indicados em vermelho. Vale destacar que não é feita a comparação com a métrica Exatidão, pois os autores em [Yang *et al.* 2017] não utilizam esse valor.

Tabela 6: Resultados RD

Base	Precisão	Recall	F1
Bugzilla	62,39%	75,92%	68,50%
Columba	51,22%	74,33%	60,65%
JDT	29,34%	73,48%	41,94%
Mozilla	15,79%	77,75%	26,25%
Platform	31,42%	77,48%	44,71%
Postgresql	49,86%	76,97%	60,52%
Média	40,00%	75,99%	50,43%
Mediana	40,64%	76,44%	52,61%
Desvio Padrão	17,30%	1,75%	15,63%

Fonte: Adaptado de Yang *et al.* (2017).

Tabela 7: Resultados DT desenvolvida

Base	Precisão	Recall	F1
Bugzilla	92%	75%	83%
Columba	86%	82%	84%
JDT	78%	75%	77%
Mozilla	64%	77%	70%
Platform	79%	76%	78%
Postgresql	87%	81%	84%
Média	81%	78%	79%
Mediana	83%	77%	80%
Desvio Padrão	9%	3%	5%

Fonte: Os próprios Autores.

Ao compararmos os resultados da Tabela 7, DT desenvolvida neste artigo, com os da Tabela 6, RD de [YANG *et al.* 2017], é verificado que a Precisão e a média

harmônica F1 da DT proposta são melhores em todas as bases de dados e também nos dados estatísticos. Ao compararmos a Recuperação, *Recall*, podemos considerar dois empates técnicos: Bugzilla da Tabela 6 com 75,92% e da DT com 75%; Mozilla da Tabela 6 com 77,75% e da DT com 77%. Já para a base de dados Platform, o resultado na Tabela 6 foi superior em 1,48%. Em relação aos valores estáticos para o *Recall*, a DT proposta só perde para o quesito desvio padrão.

8. Conclusão e Trabalhos Futuros

Neste trabalho, foi possível mostrar que para as seis bases de dados indicando casos de alterações em softwares de código aberto: Bugzilla, Columba, JDT, Mozilla, Platform e Postgresql, utilizadas neste artigo como teste para o diagnóstico preditivo de falhas em *softwares*, foram obtidos taxas de Recuperação média de 74,6% e de 78%, adotando, respectivamente, uma ANN e uma DT no processo de detecção de erros. Resultados estes que são superiores aos exibidos em trabalhos anteriores, [KAMEI *et al.* 2013] em ambas as técnicas, ANN e DT desenvolvidas, e [YANG *et al.* 2017] na DT desenvolvida. Nas outras métricas adotadas: Precisão, Exatidão, média harmônica F1, também em média a DT desenvolvida foi superior.

Neste artigo foram desenvolvidas uma ANN que não é adotada nos trabalhos de comparação, e uma DT mas com diferenças no tratamento dos dados de treinamento.

Vale destacar que os bancos de dados apresentam apenas casos de alterações em trechos específicos dos códigos dos softwares livres citados, garantindo um menor esforço de revisão. Portanto, as análises são feitas apenas nessas instâncias e não em todos os códigos dos programas, demonstrando a utilidade da abordagem JIT apresentada neste artigo, junto com o diagnóstico utilizando ANN e DT.

Em trabalhos futuros, os autores pretendem aprimorar a DT desenvolvida, diminuindo o número de nós e melhorando os resultados. Pretende-se, ainda, testar outras técnicas inteligentes de diagnóstico, como por exemplo redes RBF. Também, planeja-se fazer uma análise sobre a importância das métricas de entrada adotadas no diagnóstico, verificando a viabilidade de aumentar ou diminuir o número de entradas para aumentar a eficácia no diagnóstico de defeitos nos softwares analisados.

9. Agradecimentos

Os autores agradecem a FUNCAPE - Fundação Cearense de Apoio ao Desenvolvimento Científico e Tecnológico, pelo financiamento da bolsa de mestrado.

Referências

Arar, O. F., and Ayan, K. (2015) "*Software* Defect Prediction Using Costsensitive Neural Network". Applied Soft Computing, v. 33, p. 263-277.

Batista, B. C. F.; (2012) "Soluções de Equações Diferenciais Usando Redes Neurais de Múltiplas Camadas Com os Métodos da Descida Mais Íngreme e Levenberg-Marquardt". (Doctoral dissertation, Dissertação de mestrado, PPGME-ICEN-UFPA).

Breiman, L.; (1996) "Some Properties of Splitting Criteria". *Machine Learning*, *24*(1), 41-47.

Fenton, N. E. and Neil, M.; (1999) "A Critique of *Software* Defect Prediction Models". IEEE Transactions on *software* engineering, v.25, n. 5, p. 675-689.

Figueiredo Filho, D. B., & Silva Júnior, J. A. D. (2009). Desvendando os Mistérios do Coeficiente de Correlação de Pearson (r).

Gayathri, M., and Sudha, A.; (2014) "*Software* Defect Prediction System Using Multilayer Perceptron Neural Network With Data Mining". International Journal of Recent Tecnology and Engineering (IJRTE). v. 3, n. 2, p. 54-59.

Haykin, S.; (2001) "Redes Neurais: Princípios e Prática". Bookman Editora.

Okutan, Ahmet; Yildz, Olcay Taner. (2014) "*Software* Defect Prediction Using Bayesian Networks", Empirical *Software* Engineering, v. 19, n. 1, p. 154-181.

Jindal, R., Malhotra, R., and Jain, A. (2014). "*Software* defect prediction using neural networks". In: Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions), 3rd International Conference on, pages 16. IEEE.

Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., And Ubayashi, N.; (2013) "A Large-Scale Empirical Study of Just-In-Time Quality Assurance". IEEE Transactions on *Software* Engineering, v. 39, n. 6, p. 757-773.

Kamei, Y.; JIT Databases. Disponível em: http://research.cs.queensu.ca/~kamei/jittse/jit.zip. Acesso em: 18 mar. 2019.

Li, J., He, P., Zhu, J., And Lyu, M. R.; (2017) "*Software* Defect Prediction Via Convolutional Neural Network". In *Software* Quality, Reliability and Security (QRS), 2017 IEEE International Conference on, pages 318328. IEEE.

Pearson, Karl; Fisher, Ronald & Inman, Henry F. (1994), "Karl Pearson and R. A. Fisher on Statistical Tests: A 1935 Exchange from Nature". The American Statistician, 48,1: 2-11.

Quinlan, J. R.; (2014) "*C4.5:* Programs For Machine Learning". Elsevier.

Rokach, L., and Maimon, O. Z.; (2008) "Data Mining With Decision Trees: Theory and Applications" (Vol. 69). World scientific.

Scikit-Learn. Machine Learning in Python. Disponível em: http://scikit-learn.org/stable/index.html. Acesso em: 18 mar. 2019a.

Scikit-Learn. Machine Learning in Python. Disponível em: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html. Acesso em: 18 mar. 2019b.

Theodoridis, S.; Koutroumbas, K.; (2009) "Pattern Recognition". 3. ed. San Diego: Academic Press.

Yang, X.; (2015) "Deep Learning for Just-In-Time Defect Prediction". In QRS. p.17-26.

Yang, X., LO, D., XIA, X., and SUN, J.; (2017) "Tlel: A Two-Layer Ensemble Learning Approach for Just-In-Time Defect Prediction". Information and *Software* Technology, v.87, p. 206-220.

Yu, Hao, and Wilamowski, B. M.; (2011) "Levenberg–Marquardt Training" Industrial Electronics Handbook, vol. 5 – Intelligent Systems, 2nd Edition, chapter 12, pp. 12-1 to 12-15, CRC Press.