UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE SOBRAL

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E COMPUTAÇÃO - PPGEEC

ISMAEL ARAÚJO RAMOS

PREDIÇÃO DE DEFEITOS *JUST-IN-TIME* EM SOFTWARE UTILIZANDO INTELIGÊNCIA ARTIFICIAL

SOBRAL
2020

ISMAEL ARAÚJO RAMOS

PREDIÇÃO DE DEFEITOS *JUST-IN-TIME* EM SOFTWARE UTILIZANDO INTELIGÊNCIA ARTIFICIAL

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Computação - PPGEEC da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Engenharia Elétrica e Computação. Área de concentração: Sistemas de Informação.

Orientador: Prof. Dr. Márcio André Baima Amora.

Sobral-CE
2020

Ismael Araújo Ramos

PREDIÇÃO DE DEFEITOS *JUST-IN-TIME* EM SOFTWARE UTILIZANDO INTELIGÊNCIA ARTIFICIAL

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Computação - PPGEEC da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Engenharia Elétrica e Computação. Área de concentração: Sistemas de Informação.

Aprovada em: ___/___/_____.

BANCA EXAMINADORA

_____
Prof. Dr. Márcio André Baima Amora (Orientador)
Universidade Federal do Ceará (UFC)

_____
Prof. Dr. Alexandre Cabral Mota
Universidade Federal de Pernambuco (UFPE)

_____
Prof. Dr. Iális Cavalcante de Paula Júnior
Universidade Federal do Ceará (UFC)

A HaShem.
Aos meus pais, Israel e Meirelurde.
À minha esposa, Dayane.
À minhas filhas Sarah e Ester.
À minha irmã Lívia e meu cunhado Edwayne.

AGRADECIMENTOS

À FUNCAP, pelo apoio financeiro com a manutenção da bolsa de auxílio.

Ao Prof. Dr. Márcio André Baima Amora, pela excelente orientação.

Aos professores participantes da banca examinadora Alexandre Cabral Mota e Iális Cavalcante de Paula pelo tempo, pelas valiosas colaborações e sugestões.

Aos colegas da turma de mestrado, pelas reflexões, críticas e sugestões recebidas.

“Não fui eu que ordenei a você? Seja forte e corajoso! Não se apavore nem desanime, pois o Senhor, o seu Deus, estará com você por onde você andar”, Josué 1:9.

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. Modelos de predição de defeitos em software podem ser utilizados para isso. Os principais objetivos deste trabalho são realizar um estudo e apresentar uma proposta de modelo de predição de defeitos *Just-In-Time (JIT)* em software. Algumas vantagens da abordagem *JIT* são mais rapidez na análise, melhor aproveitamento de recursos, facilidade de identificação de possíveis áreas do código que estejam defeituosas e facilidade de encontrar o(s) autor(es) das modificações. Nesta dissertação é apresentada uma proposta para a solução do problema de identificação de erros *JIT* utilizando rede neural artificial (*Artificial Neural Network - ANN*) e árvore de decisão (*Decision Tree – DT*). As bases de dados utilizadas como treino, teste e validação apresentam no total 227417 *commits* divididos em seis projetos de software livre (Bugzilla, Columba, JDT, Mozilla, Platform e Postgres). Os resultados obtidos tanto com a ANN quanto com a DT são em média superiores aos trabalhos de comparação. Serão apresentadas as técnicas utilizadas no desenvolvimento do trabalho, bem como suas similaridades e diferenças com as abordagens anteriores.

Palavras-chave: Qualidade de Software. Predição de Defeitos. *Just-In-Time*. Redes Neurais Artificiais. Árvore de Decisão.

ABSTRACT

In the development or modification of a software, the product must have least amount of possible errors. Methods of predicting defects in software could be used for this. The principal objectives about this are performance a study and propose a defect prediction software *Just-In-Time* (JIT). Some advantages of the *JIT* approach are faster analise, better team utilization, easier identification of possible areas of code that are defective, and ease of finding the author (s) of modifications. In this work we present a proposal of the use of *Just-In-Time* (JIT) error identification using Artificial Neural Network (ANN) and decision tree (DT). The databases used as training, testing and validation have 227417 commits in total divided into six open source projects (Bugzilla, Columba, JDT, Mozilla, Platform and Postgres). The results obtained with techniques, ANN and DT, are on average higher than the works of comparation. The techniques used in the work development, as well as their similarities and differences with the previous approaches will be presented.

Keywords: Quality Assurance Sofware. Defect Prediction. *Just-In-Time*. Artificial Neural Network. Decision Tree.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

JIT	*Just-In-Time* – No mesmo instante
DT	*Decision Tree* - Árvore de decisão
RD	*Random Forest* – Floresta aleatória
IA	Inteligência Artificial
ANN	*Artificial Neural Network* – Redes neurais artificiais
CBS	*Classify Before Sorting* – Classificação antes da ordenação sorting
NS	*Number of Modified Subsystems* - Quantidade de subsistemas
ND	*Number of Modified Directories* - Quantidade de diretórios
NF	*Number of Modified Files* - Quantidade de arquivos
LA	*Lines of Code Added* - Linhas adicionadas
LD	*Lines of Code Deleted* - Linhas deletadas
LT	*Lines of Code in File Before the Change* - Linhas totais
FIX	Propósito da mudança
NDEV	*Number of Developers that Changed the Modified Files* - Quantidade de desenvolvedores que modificaram o arquivo
AGE	*Average time interval between the last and the current change* - Tempo médio entre alterações
NUC	*Number of unique changes to the modified files* - Quantidade de alterações únicas
EXP	*Developer experience* - Experiência do desenvolvedor
REXP	*Recent developer experience* - Experiência recente do desenvolvedor
SEXP	*Developer experience on a subsystem* - Experiência do desenvolvedor no subsistema
ABC	*Artificial Bee Colony* - Colônia de abelhas artificial
RBF	*Radial Basis Function* - Função de base radial
CNN	*Convolutional Neural Network* - Redes neurais convolucionais
BBF	*Bilief Bayes Network* - Redes bayesianas de Bilief
MLP	*Multi Layer Perceptron* - Perceptron multicamadas
AC	Árvore de Classificação
AR	Árvore de Regressão

CART	*Classification And Regretion Tree* - Algoritmo de árvore de classificação e regressão
API	*Aplication Program Interface* - Interface de programação de aplicativos
TP	*True Positive* - Verdadeiros p*ositivos*
FP	*False Positive* - Falsos positivos
TN	*True Negative* -Verdadeiros negativos
FN	*False Negative* - Falsos negativos

LISTA DE SÍMBOLOS

V	Complexidade Ciclomática de McCabe
E	Quantidade de Aretas de um Grafo
R	Número de nós de um Grafo
Ev	Complexidade Essencial de McCabe
M	Todos os subgrafos de um Grafo que possuem entrada unitária e nós de saída
P	Programa sugerido
AC	Número de partes testadas
Gi	Grafo
iv	Complexidade de Projeto de McCabe
n_1	Número de operadores distintos
n_2	Número de operandos distintos
N_1	Total de ocorrência de operadores
N_2	Total de ocorrência de operandos
n_1^*	Número de operadores potenciais
n_2^*	Número de operandos potenciais
a	Variável representando número inteiro
b	Variável representando número inteiro
$>$	Símbolo comparativo maior que
N	Tamanho
n	Vocabulário
V	Volume
V^*	Volume mínimo
L	Nível do Programa
L^*	Nível Estimado
I	Conteúdo Inteligente
E	Esforço de programação
T	Tempo de programação necessário
D	Dificuldade de Implementação
loc	Linhas de código
P_k	Probabilidade de k ocorrer

K	Arquivo modificado
μ	Número de arquivos envolvidos
A	Arquivo envolvido no *commit*
B	Arquivo envolvido no *commit*
C	Arquivo envolvido no *commit*
XP	Suposto desenvolvedor
YP	Suposto desenvolvedor
$FS1$	Suposta funcionalidade do sistema
$FS2$	Suposta funcionalidade do sistema
f	Representação de funções
$\forall$	Símbolo matemático que significa para todo
$=$	Símbolo matemático de igualdade
$\leq$	Operador comparativo menor igual
$<$	Operador comparativo menor
$senh$	Seno Hiperbólico
$cosh$	Cosseno Hiperbólico
y_i	Saída do neurônio i
nt	Total de entradas para o neurônio i
x_j	j-ésima entrada
W_{ij}	Peso que interage da conexão entre o neurônio i e a j-ésima entrada
θ_i	Valor de tendência do neurônio i
f_i	Função de ativação do neurônio i
$tanh$	Função tangente hiperbólica
u	Função de ativação
x_i	Atributo avaliado ou vetor de treinamento
β	Operação lógica
$\propto$	Valor limite
c_1	Caminho
c_2	Caminho
$\neq$	Operador de desigualdade
$\geq$	Operador maior igual
$\in$	Símbolo de pertence

R^n	Conjunto
i	Número pertencente ao conjunto R^n
l	Maior índice pertencente ao conjunto R^n
y	Vetor de rótulos
R^l	Conjunto
m	Nó de uma árvore
Q	Dados de um nó
j	Recurso a ser comparado
t_m	Limite de comparação
θ	Divisão dos recursos de treinamento
Q_{esq}	Recursos que ficam à esquerda do nó
Q_{dir}	Recursos que ficam à direta do nó
$\|$	Tal que
\	Exceto
$G(Q,\theta)$	Impureza dos recursos
θ^*	Argumento mínimo da impureza
n_{esq}	Observações do lado esquerdo
N_m	Total de observações
n_{dir}	Observações do lado direito
H	Gini, Entropia ou *Misclassification*
p_{mk}	Probabilidade de k ocorrer no nó m
$\%$	Porcentagem
o	Valores de entrada da rede neural
W_{oi}	Peso que interage da conexão entre os neurônios de i e as entradas de o
W_{ij}	Peso que interage da conexão entre os neurônios de j e as entradas de i
i	Neurônios da camada de entrada
j	Neurônios da camada oculta
W_{jk}	Peso que interage da conexão entre os neurônios de k e as entradas de j
k	Neurônios da camada de saída
$F(Y)$	Função não linear
Y	Argumento da função não linear
X	Saída da função não linear

R^n	Conjunto
R^m	Conjunto
Y'	Valor estimado de Y
X'	Valor que se aproxima de X
ε	Erro mínimo aceitável
Δ	Pequeno incremento no valor estimado Y'
J	Jacobiano, derivada parcial de X em relação a Y
$\|\| \|\|$	Operador normal
J^T	Transposta de J
Y_r'	Ajuste no valor estimado
λ	Valor de ajuste proposto por Levenberg
In	Matriz identidade
$diag$	Matriz diagonal
Z	Normalização
x	Valor do atributo
med	Média aritmética da métrica
s	Desvio padrão

SUMÁRIO

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 e eficazes no software, alocando o mínimo de recursos possíveis, maximizando as falhas encontradas e o mais rápido possível. Os métodos de predição de defeitos podem ser usados para otimizar esse processo. Sua atuação é na indicação de áreas onde podem haver erros.

Vários autores na literatura apresentam estudos e métodos relacionados com a predição de defeitos em softwares. Alguns desses trabalhos serão comentados na Seção 2.4 do Capítulo 2. Em geral, os autores utilizam dois tipos de abordagem para predição de defeitos em software: 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 introdução de defeitos no momento do *commit* (conjunto de mudanças criadas no sistema para implementação de uma nova versão).

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

1.1 Motivação

Uma forma de garantir que um software fabricado ofereça qualidade aos consumidores é adotar padrões de desenvolvimento que apresentem boas metodologias (KOSCIANSKI; SOARES, 2007). As etapas de testes também são uma delas. Assim, o emprego de boas estratégias podem melhorar o desempenho das equipes de testes e consequentemente melhorar a qualidade do produto. Visando oferecer uma estratégia para otimização das etapas de testes, essa dissertação busca analisar algumas opções oferecidas na literatura para predição de defeitos em software.

Muitos sistemas preditivos são baseados na análise de arquivos, que podem ser um tanto quanto extensos a depender de quão distribuído está o sistema e também do tamanho dos arquivos que foram desenvolvidos. Observando essa questão, sistemas preditivos que adotam esta maneira de análise, podem demandar muito esforço e tempo de análise das equipes de testes. Outro ponto negativo que pode ser observado é o caso no qual o software analisado possui muitos desenvolvedores e seja difícil de encontrar quem realizou a implementação do código que possivelmente esteja defeituoso. As predições JIT apresentam uma proposta que busca eliminar esses problemas.

Este trabalho procura criar um sistema preditivo para ajudar na etapa de testes de software. O foco é na análise instantânea de *commits*, que permite verificações com foco em possíveis partes alteradas ou criadas no código fonte. Isso pode melhorar o tempo de execução dos testes, deixando a etapa mais rápida e direta.

1.2 Objetivos

Os objetivos gerais desse trabalho são o estudo e a proposta de um sistema preditivo de defeitos *Just-In-Time* em software utilizando inteligêcia artificial. A finalidade é oferecer as equipes de desenvolvedores/*testers* uma ferramenta preditiva que informe se um determinado *commit* realizado insere ou não um defeito ao sistema.

Os objetivos específicos são três:

1. Utilização de duas técnicas de inteligência computacional aplicada para a criação de métodos de diagnóstico preditivo de defeitos em softwares.
2. Comparação das tecnicas desenvolvidas com os trabalhos produzidos por duas referências de destaque: Kamei *et al.* (2013) e Yang *et al.* (2017). Utilizando, inclusive, bases de dados em comum para comparação.
3. Refinamento das técnicas de inteligência computacional utilizadas com o objetivo de simplificar ao máximo o tamanho e aumentar a capacidade de generalização.

1.3 Desenvolvimento da pesquisa

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 o desenvolvimento dos métodos propostos e validação 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 trabalho 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 trabalho 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 trabalho utiliza modificações no tratamento de dados, em comparação com (KAMEI *et al.,* 2013) que também adota a mesma técnica.

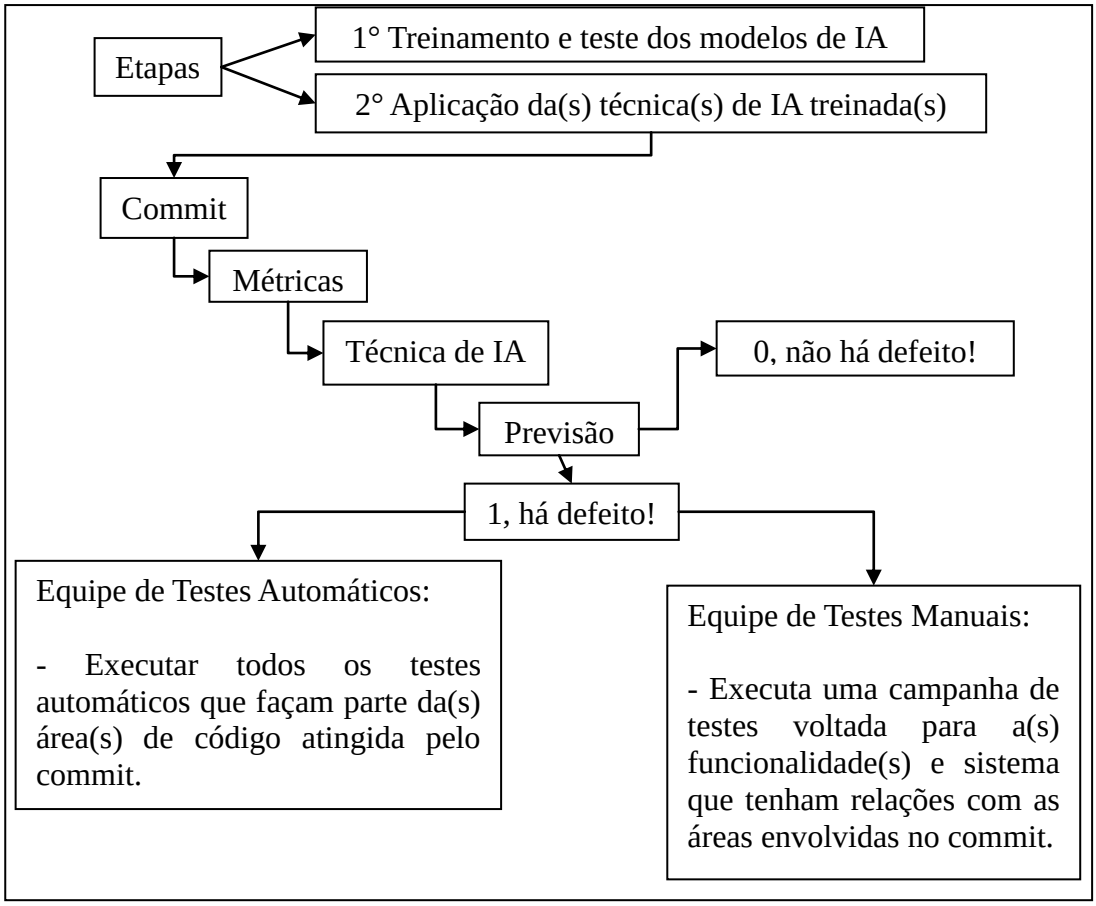
1.4 Metodologia proposta

A Figura 1 mostra um esquemático de como seria a aplicação proposta neste trabalho. Basicamente, sendo dividido em duas etapas. A primeira etapa seria mais metodológica, onde será feita a separação e preparação dos dados utilizados para treinamento e teste da técnica escolhida, que no caso deste trabalho pode ser tanto uma rede neural como uma árvore de decisão. Na segunda etapa seria a aplicação do modelo já treinado e testado. Por exemplo, considerando que uma árvore de decisão foi treinada e testada, durante a modificação de um software são coletadas as métricas utilizadas e as mesmas são aplicadas como entrada na árvore de decisão que serve como preditora. A árvore de decisão então avalia as entradas e retorna uma resposta preditiva, que pode ser 0 ou 1: caso a resposta seja 0, a previsão afirma que o sistema está livre de defeitos; caso contrário, a previsão afirma que existe a possibilidade de defeito. Logo a equipe de testes automáticos poderá executar todos os testes automáticos que tenham haver com as áreas do código que foram envolvidas no *commit*, como também a equipe de testes manuais agirá com uma campanha de testes voltada para as funcionalidades do sistema que também tenham haver com as áreas afetadas pelo *commit*.

Na previsão JIT, como o foco é voltado para o momento do *commit*, normalmente como a(s) alteração(ões) são realizada(s) em uma funcionalidade de cada vez. Portanto, caso a previsão acuse um possível defeito, rapidamente pode-se identificar a(s) área(s) criada(s)/modificada(s), que normalmente não possuem alta granularidade. Sabendo disso, o

esforço envolvido na revisão pode ser muito menor. Como um *commit* normalmente é realizado pelo autor das possíveis mudanças no sistema, caso a predição aponte um possível defeito, será relativamente fácil para o autor executar uma análise e possível correção do sistema.

Figura 1 – Etapas para aplicação da previsão JIT proposta



Fonte: O prórpio autor.

1.5 Produção científica

Durante o desenvolvimento da pesquisa foi gerado o seguinte artigo científico:

1. RAMOS, I. A.; AMORA, M. A. B. Predição Just-In-Time de Defeitos em Software Utilizando Inteligência Artificial. In: XLVI Seminário Integrado de Software e Hardware, Belém, 2019. Anais do XLVI Seminário Integrado de Software e Hardware (46 SEMISH)/ XXXIX Congresso da Sociedade Brasileira de Computação (39 CSBC). Realizado em Belém (PA) entre 14 e 18 de julho de 2019.

1.6 Estrutura do trabalho

O trabalho está divido em capítulos, citados a seguir. No Capítulo 2 são apresentadas informações sobre teste de software, previsões de defeitos feitas com base em arquivos e baseados em JIT. Analisa-se ainda algumas métricas que a literatura adota e é realizada uma revisão bibliográfica de trabalhos de predição de defeitos em softwares de ambas as abordagens. No Capítulo 3 são apresentadas as técnicas de inteligência computacional utilizadas nesse trabalho, ANN e DT.

No Capítulo 4 é exibida a metodologia adotada na pesquisa bem como uma descrição de todos os bancos de dados utilizados. Comentam-se na Seção 4.1 possíveis configurações para as técnicas de inteligência computacional exploradas nas análises. Ainda na mesma Seção, detalhes de treinamento e teste para os sistemas preditivos e escolha criteriosa, baseada na revocação, do melhor produto preditivo gerado são descritos. No Capítulo 5, são comentados os resultados da ANN e da DT desenvolvida. No Capítulo 6 são apresentadas as conclusões deste trabalho e as perspectivas futuras da pesquisa.

2 DIAGNÓSTICO DE DEFEITOS EM SOFTWARE

2.1 Desenvolvimento de software e possibilidade de indicação de defeitos

O processo de fabricação de software está em constante evolução na busca de aperfeiçoamento e aprimoramento das etapas de desenvolvimento e também da metodologia adotada. Pode-se afirmar que nos primórdios, não havia padrões de desenvolvimento e muito menos metodologias. Fato que deixava o produto desenvolvido quase que impossível de realizar uma manutenção ou incremento de funcionalidades, a não ser que o criador original fosse o autor das respectivas ações. Com isso, a maioria dos sistemas antigos tornaram-se o que denomina-se na literatura de “sistemas legados”, pois os autores originais seriam praticamente os únicos a executar ações neles, deixando assim uma espécie de “legado” (SEACORD *et al.*, 2003).

A criação de software evoluiu para algo mais profissional na década de 1960, com o intuito de melhorar a qualidade e padronizar a criação. Um software de qualidade é aquele que atende aos seguintes requisitos: pode ser mantido, é estável, rápido, usável, testável, legível, tamanho considerado aceitável, custo aceitável, seguro, poucas falhas, elegante, conciso, proporciona a satisfação do cliente entre outros. Existem muitas discussões sobre a melhor forma de se criar software de qualidade. Estas discussões possuem inúmeras variáveis, como por exemplo: padrões de projeto, gerenciamento, ambiente de trabalho, contratações e outras. O conceito de engenharia de software abrange todo este conjunto (KING’S, 2018). Assim, como na indústria, observou-se que são necessárias etapas bem definidas e tarefas claras no ciclo de desenvolvimento de um software (BISBAL *et al.*, 1999). Tudo isso alinhado para obtenção de um produto de qualidade e com o menor número de erros possíveis.

Dentre as etapas do desenvolvimento, uma das mais importantes é a de testes. Basicamente temos dois grandes grupos de teste (KOSCIANSKI; SOARES, 2007):

a) Testes de caixa branca, também denominados estruturais;
b) Testes de caixa preta, também denominados funcionais.

Para os testes de caixa branca, é necessário o conhecimento da estrutura interna do software. O foco é voltado para análise do perfeito funcionamento de cada método, classe, objeto e etc. Já nos testes de caixa preta, analisa-se o comportamento do sistema mediante especificações de requisitos de sua funcionalidade, como por exemplo, saber se uma determinada ação gera o resultado esperado (KOSCIANSKI; SOARES, 2007).

Dentro do universo de testes caixa branca, ou seja, que são realizados dentro do sistema totalmente conhecido e analisa-se sua estrutura, a predição JIT (baseada na análise no instante do *commit*) pode ser utilizada no auxílio aos desenvolvedores e/ou testadores do sistema a encontrar possíveis partes do código que possam apresentar erros.

Quando se fala em mudanças no software, logo se pensa em teste de regressão. Os testes de regressão se encaixam em ambos os tipos, tanto nas análises estruturais como nas análises do tipo caixa preta. Sendo aplicados para verificar se as mudanças em determinadas funcionalidades do sistema não alteraram o comportamento correto de outras. Nestes casos, a predição JIT seria usada para que os desenvolvedores e/ou testadores explorassem possíveis funcionalidades novas ou alteradas que possivelmente introduziram um defeito ao sistema. Já no caso do tipo análise estrutural, o teste de regressão seria simplesmente executar toda a suíte de testes realizada pela equipe na etapa de desenvolvimento do código, como por exemplo, testes unitários. Os testes unitários normalmente são criados para verificar o comportamento de cada método desenvolvido. Uma vez criados, podem ser executados sempre que forem necessários para a verificação do correto comportamento da funcionalidade.

A grande motivação em testar o software é buscar evitar possíveis danos financeiros e na reputação da empresa, além de objetivar entregar um produto de qualidade que será bem visto no mercado. Os custos para uma revocação, atualização para correção de um defeito de um produto já entregue aos consumidores, do sistema são elevados e o nome da empresa fica marcado negativamente na mente dos consumidores caso entregue um produto defeituoso no mercado. A grande motivação das predições JIT seria na alocação inteligente de recursos, pois ao se focar em determinadas áreas pode-se minimizar o tempo gasto nas revisões de códigos e/ou funcionalidades. Outro ponto de vantagem seria financeiro, pois com a redução do tempo de revisão é possível diminuir o tempo gasto na fabricação e consequentemente o produto se torna mais barato para se produzir.

2.2 Predições por arquivos

Alguns trabalhos da literatura fazem predições de defeitos utilizando arquivos. Nesse tipo de metodologia, de forma geral, os recursos analisados são buscados dentro do código fonte e, após a extração das métricas, treina-se o preditor com alguma técnica de Inteligência Computacional. Como exemplo dessas métricas, podemos observar na Tabela 1 métricas que são adotadas em: (ARAR; AYAN, 2015), (QUTAISH; ABRAN, 2005) que representam métricas propostas ou modificadas das referências (MCCABE, 1976) e (HALSTEAD, 1977).

Tabela 1 – Exemplo de métricas de predições utilizadas em arquivos

Tipo	Métrica	Definição
McCabe	*Loc*	Total de linhas do código
	v(g)	Complexidade ciclomática
	ev(g)	Complexidade essencial
	iv(g)	Complexidade de projeto
Basic Halstead	n_1	Contagem de operadores sem repetições
	n_2	Contagem de operandos sem repetições
	N_1	Contagem total de operadores
	N_2	Contagem total de operandos
	n_1^*	Número de operadores potenciais
	n_2^*	Número de operandos potencias
Derived Halstead	N	Tamanho do Programa = (N_1+N_2)
	n	Vocabulário = (n_1+n_2)
	V	Volume = $Nlog_2n$
	V^*	Volume Potencial = $V^*=(2+n_2^*)log_2(2+n_2^*)$
	L	Nível de Programa = V^*/V
	D	Dificuldade = $1/L$
	L^*	Nível de Programa Estimado = $2n_2/n_1N_2$
	I	Inteligência = L^**V
	E	Esforço para escrever o programa = V/L
	T	Tempo estimado = $(E/18)$ segundos

Fonte: Adaptado pelo autor a partir de Arar e Ayan (2015) e Al Qutaish e Abran (2005).

A complexidade ciclomática, *v(g)*, é uma métrica responsável por calcular a quantidade de caminhos que são linearmente independentes, ou seja, a quantidade de caminhos que através do fluxograma, um grafo onde os nós representam instruções e as setas um fluxo de controle, de um programa não representam uma combinação linear de quaisquer outros caminhos pertencentes ao conjunto. Matematicamente podemos representar por (1), (MCCABE, 1976):

$$v(g) = e - r + 2p, \quad (1)$$

Onde *e* seria a quantidade de arestas, *r* o número de nós e *p* é o número de componentes conectados.

A complexidade essencial, *ev(g)*, é definida como a diferença entre a complexidade ciclomática de McCabe e a quantidade de decomposição de todos os subgrafos possíveis existentes no fluxograma, (MCCABE, 1976):

$$ev(g) = v(g) - M, \quad (2)$$

sendo *M* a quantidade de todos os subgrafos com entrada unitária e nós de saída.

Outra métrica é a complexidade de projeto, *iv(g)*, utilizada para analisar a importância entre a complexidade dos dados e a complexidade ciclomática de McCabe. Assim como é exemplificado em McCabe (1976), dado um determinado programa *P*, com complexidade ciclomática *v* e com um número de partes testadas igua a *ac*. Caso *ac* seja menor que *v*, pelo menos uma das condições abaixo é atingida (MCCABE, 1976):

a) não foram testadas todas as partes;
b) o grafo do fluxograma pode ser reduzido em complexidade *v* - *ac*;
c) partes do programa podem ser reduzidas em linhas de código (a complexidade aumentou para economizar espaço).

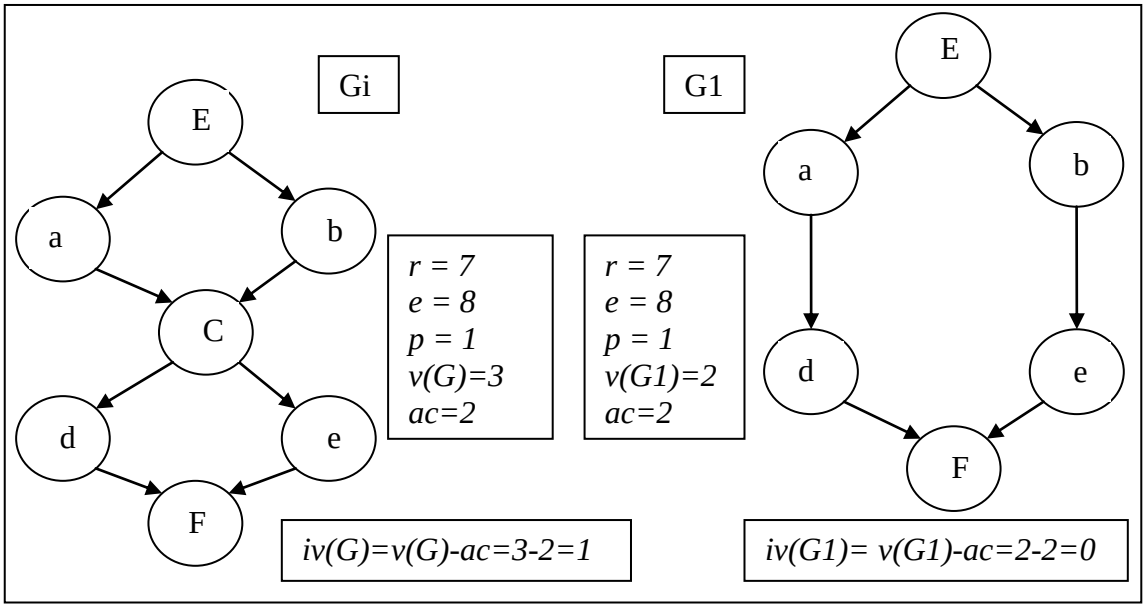
Para entender melhor, vejamos o caso do grafo *Gi* descrito na Figura 2, onde a complexidade atual *ac* é igual a 2, ou seja, os testes do programa percorreram dois caminhos, por exemplo [E,a,C,e,F] e [E,b,C,d,F].

Percebe-se que os testes não atingiram todos os caminhos possíveis, mas todos os nós foram pecorridos. Restaram então os caminhos [E,a,C,d,F] e [E,b,C,e,F] para serem explorados. Analisando o caso, nota-se que *ac* < *v(G)*, satisfazendo a condição do item (b), e então o grafo *Gi* pode ser reduzido retirando o nó C, como resultado exibido no grafo G1.

Outras métricas bastante utilizadas são as propostas por (HALSTEAD, 1977). Essas métricas são baseadas em cálculos realizados com seis variáveis:

a) n_1 : número de operadores distintos;
b) n_2 : número de operandos distintos;
c) N_1 : número total de ocorrência de operadores;
d) N_2 : número total de ocorrência de operandos;
e) n_1^*: número de operadores potenciais;
f) n_2^*: número de operandos potenciais.

Figura 2 – Exemplo de cálculo da complexidade de projeto de McCabe



Fonte: Adaptado de McCabe (1976).

Um operador pode ser o nome de uma função, o símbolo de atribuição ou agrupamento. Já os operandos podem ser os parâmetros passados por uma função ou procedimento. O número de operadores potenciais, n_1^*, assim como os operadores potenciais, n_2^* são definidos como o mínimo possível para um programa ou módulo funcionar. A Figura 3 nos ajuda a entender o caso descrito. Nela uma função simples de comparação é implementada, onde se recebe como argumento dois números inteiros *a* e *b* e se retorna o maior valor entre eles.

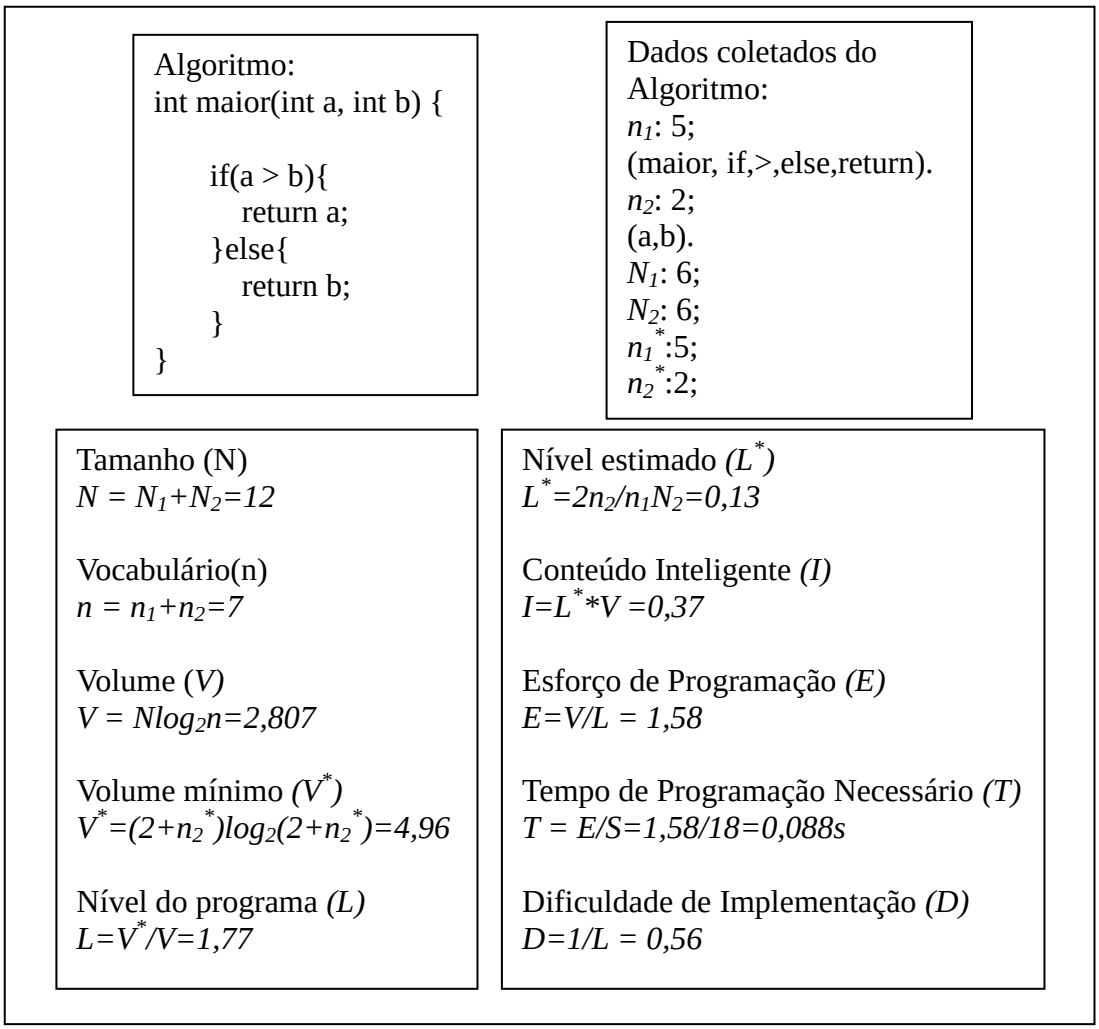
A Figura 3 nos mostra um exemplo de cálculo das métricas de Halstead, as mesmas apresentadas na Tabela 1. Podemos observar que elas analisam a estrutura do código, definindo atributos como número de operandos e operadores como base e calculando outras medidas, como por exemplo, tamanho do programa *(N)*, volume *(V)*, vocabulário (*n*), volume mínimo (V^*), nível do programa (*L*), nível estimado (L^*), conteúdo inteligente (*I*), esforço de programação (*E*), tempo de programação necessário (*T*) e dificuldade de implementação (*D*).

Como é possível observar na Tabela 1, temos algumas métricas consagradas e muito utilizadas pela literatura na predição de defeitos com arquivos. Estas baseiam-se em elementos extraídos dentro do código fonte da aplicação desenvolvida ou em fabricação. Podemos notar que o esforço dos programadores, as dificuldades em desenvolver e as complexidades do programa também são analisadas e levadas em consideração. Algumas questões de projeto como histórico de mudanças, perfil técnico dos desenvolvedores e objetivo da escrita (se é para correção um defeito ou inserir algum módulo novo) não são

abordadas.

Neste tipo de predição, análise por arquivos, normalmente é realizada quando é findado o ciclo de desenvolvimento e se chega à parte dos testes. Vale ressaltar que ela pode ser bem demorada e trabalhosa, pois ao depender da quantidade de classes, métodos e etc. que os arquivos tenham e, principalmente, o quão acoplados estão as estruturas, a revisão pode demandar muito tempo e esforço da equipe que for realizá-la.

Figura 3 – Exemplo de aplicação das métricas de Halstead



Fonte: O próprio autor.

O grande desafio neste tipo de predição seria exatamente como conseguir aplicar a predição quando se têm muitos arquivos e o sistema é muito grande e muito acoplado. Uma revisão poderá levar muito tempo, principalmente se levarmos em conta que algo novo poderá alterar o comportamento de funcionalidades antigas e que nem sempre será possível saber em

qual ponto do sistema a predição está acusando um risco de defeito.

Outro ponto a se levar em consideração é que nem sempre estará recente na memória do desenvolvedor as possíveis áreas que foram realizadas mudanças. Todos estes fatores citados podem ser um desafio quando se trabalha com predições de defeitos que usam arquivos. O uso da abordagem JIT é proposto para solucionar problemas de otimização de recursos e tempo que seriam necessários para testar o software.

2.3 Predições JIT

Just-In-Time (JIT) é uma técnica baseada na análise instantânea do conjunto de mudanças realizadas no software (*commits*). Faz-se simplesmente uma análise do *commit* e é avaliado se ele possui ou não possibilidade de defeitos. Nesse tipo de abordagem, são utilizadas métricas diferentes se compararmos com as utilizadas na análise de arquivos. A Tabela 2 nos mostra exemplos dessas métricas.

Para implementar uma metodologia desse tipo, existem quatro etapas, como descrito em (KAMEI *et al.*, 2013):

a) 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;

b) 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 2, 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;

c) 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 rótulos previamente conhecidos (etapa a);

d) 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.

Tabela 2: 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).

2.3.1 Métricas utilizadas em JIT

A primeira dimensão descrita nas métricas é a dimensão de difusão. Ela busca analisar a forma de distribuição do código fonte em relação a estrutura do projeto. Assim como descrito em Kamei *et al. (*2013), se uma mudança/implementação for muito distribuída, torna-se mais complexo o entendimento e dificulta o controle dos locais modificados/criados. Essa dimensão é composta por quatro métricas. Cada métrica é uma contagem simples das seguintes características, com exceção da Entropia:

a) Denominação dos diretórios/pasta (ND);
b) Nome do diretório raiz e subsistemas (NS);
c) Designativo do arquivo (NF);
d) Entropia.

A Entropia é calculada com base na equação de similaridade de Hassan (KAMEI *et al.*, 2013), descrita em (3).

$$H(P) = -\sum_{k=1}^{n} P_k log_2 P_k, \quad (3)$$

Onde a probabilidade P_k é:

$$P_k \geq 0, \forall k \epsilon \{1,2, \dots, \mu\}, \quad (4)$$

No qual μ é o número de arquivos envolvidos em uma mudança/implementação; *P* é o conjunto das probabilidades de P_k (proporção em que um arquivo *k* é modificado), onde o somatório de todas as proporções de todos os arquivos *k* possui valor unitário.

Por exemplo, consideremos que em um *commit* resultou em alteração/implementação de 30 linhas de um arquivo A, 20 linhas de B e 10 linhas de C. O cálculo da Entropia é exibido em (KAMEI *et al.*, 2013):

$$H(P) = \left(-\frac{30}{60} log_2 \frac{30}{60}\right)\left(-\frac{20}{60} log_2 \frac{20}{60}\right)\left(-\frac{10}{60} log_2 \frac{10}{60}\right) = 1{,}46.$$

A dimensão de tamanho é uma das mais simples de ser contabilizada. Ela consiste em três métricas de fácil quantização. O número de linhas adicionadas (LA), quantidade de linhas

deletadas (LD) e a contabilização das linhas totais envolvidadas em um *commit* (LT) (KAMEI *et al.*, 2013).

A dimensão de propósito foi definida como apresentando apenas dois valores possíveis: nulo ou unitário. Será nulo caso o motivo do *commit* não seja para corrigir um defeito e será unitário se seu propósito for para correção de um *erro* (KAMEI *et al.*, 2013).

A dimensão de Histórico é motivada por buscar analisar se determinado *commit* pode ser causa de defeitos futuramente (KAMEI *et al.*, 2013). Ela está subdividida em três métricas (KAMEI *et al.*, 2013):

a) NDEV – Quantidade de desenvolvedores que mudaram um arquivo;
b) AGE – Média de tempo entre as mudanças no arquivo atual e outros arquivos;
c) NUC – Quantidade de modificações únicas em um arquivo.

A métrica NDEV pode ser exemplificada da seguinte maneira: Sejam A, B e C três arquivos envolvidos em um *commit*. Considere que o arquivo A foi manipulado apenas pelo desenvolvedor XP, porém os arquivos B e C sofreram atuação dos desenvolvedores XP e YP. Como no *commit* temos dois desenvolvedores envolvidos, o NDEV será 2 (KAMEI *et al.*, 2013).

O AGE é uma média e pode ser calculada da seguinte maneira: considere que os mesmos arquivos anteriores, A, B e C que foram envolvidos no *commit*, e que em seus históricos definem que A foi modificado três dias atrás, B cinco dias atrás e C dois dias atrás, então AGE será (KAMEI *et al.*, 2013):

$$AGE = \frac{3+5+2}{3} = 3{,}333.$$

O NUC pode ser exemplificado da seguinte maneira: digamos que os mesmos três arquivos A, B e C no mesmo *commit* sofreram as seguintes mudanças: o arquivo A recebeu uma alteração na funcionalidade FS1 e os arquivos B e C tiveram uma mudança em uma funcionalidade comum a eles FS2. Logo, temos duas modificações, FS1 e FS2. Assim, o NUC será 2 (KAMEI *et al.*, 2013).

A última dimensão descrita é a de Experiência. Nessa dimensão temos três métricas abordadas por (KAMEI *et al.*, 2013):

a) EXP – Experiência do desenvolvedor;
b) REXP – Experiência recente do desenvolvedor;

c) SEXP – Experiência do desenvolvedor no subsistema.

Exemplificando, considere que um desenvolvedor XP tenha realizado quinze mudanças no sistema anteriormente. Dessas 15 mudanças, considere que XP tenha feito dez mudanças no ano atual, duas no ano passado e três no ano anterior a este. Considere ainda que as mudanças foram todas dentro de um mesmo subsistema que foi solicitado nova alteração. Neste cenário, teremos os seguintes valores para as métricas: EXP será de valor quinze, pois XP teve quinze alterações anteriores que contam como experiência. SEXP também será quinze, pois consideramos que as mudanças foram realizadas no mesmo subsistema. REXP será calculado da seguinte forma (KAMEI *et al.*, 2013):

$$REXP = \frac{10}{1} + \frac{2}{2} + \frac{3}{3} = 12.$$

As grandes vantagens em utilizar o JIT são (KAMEI *et al.*, 2013):

a) Rapidez na revisão do código, pois são analisadas as áreas modificadas recentemente;
b) Poderá ser identificado o programador da equipe que realizou a mudança de forma precisa facilitando assim uma possível conversa entre os membros da equipe de desenvolvimento e de testes;
c) A modificação recente na memória de quem a realizou, fato de grande valia na correção de possíveis problemas.

Como podemos perceber, ao optar pelo JIT a empresa terá uma redução de custos, pois o tempo de revisão e a quantidade de pessoas envolvidas na revisão do sistema poderão ser menores com relação à técnica que utiliza apenas a predição com arquivos. Fato que é de grande valia em relação à economia de tempo de projeto. Os desenvolvedores do sistema poderão ajudar ou até mesmo realizar as possíveis revisões em primeira instância, realizando um “filtro” para que quando cheguem ao time de testes as revisões sejam mais breve possíveis e com a menor quantidade de erros possíveis.

Nos casos de testes de regressão, onde todo o sistema deve ser analisado profundamente para saber se as novas funcionalidades não alteraram o comportamento correto que as anteriores apresentavam, a predição JIT é fundamental como ponto de alerta para o time de testes, pois poderão dedicar um esforço maior para possíveis áreas e funcionalidades

que estão relacionadas com os locais de risco informados.

2.4 Revisão bibliográfica

O estado da arte da predição de defeitos em software, basicamente se subdivide em dois tipos de pesquisa: as que utilizam os arquivos para executar um diagnóstico, que pode ser um tanto quanto trabalhosa caso hajam muitos arquivos para serem verificados; e por fim as que se baseiam nos *commits* para executar uma previsão menos granulosa, pois possui um foco na última alteração realizada.

2.4.1 Previsões baseadas em arquivos

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 à 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).

2.4.2 Previsões baseadas em commits – abordagem just-in-time, JIT

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 revocação, revocação, (verdadeiros positivos divididos pela soma entre os verdadeiros positivos e os falsos negativos) 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 revocação em média de quase 76%.

No trabalho de Wei Fu e Tim Menzies (2017) é realizado um estudo sobre as técnicas de IA supervisionadas e não supervisionadas para saber qual tipo de abordagem melhor se encaixaria para previsões. Eles afirmam que os preditores supervisionados são melhores, porém não muito do que os não supervisionados. Por fim incentiva a outros pesquisadores combinarem métodos supervisionados e não supervisionados para esse tipo de abordagem.

Xiang *et al.* (2018) executam uma pesquisa baseada na predição de defeitos através do reconhecimento de esforço com múltiplos objetivos. Utilizando esse cenário, conseguem resultados melhores do que o apresentado na literatura nos casos, (validação cruzada conseguiram uma *accuracy* (verdadeiros positivos somados com os verdadeiros negativos e divididos pelo total de casos, ou seja, a soma dos verdadeios e falsos negativos e positivos) de média de 0,638 e na validação cruzada do tipo *timewise* obtiveram uma *accuracy* de 0,623 e de 0,730 no cenário de validação cruzada combinando de dois em dois projetos. No tipo validação cruzada pura, eles executam o mesmo balanceamento dos dados que Kamei *et al*. (2013) e após isso fazem o *10-fold* (técnica de validação cruzada). Já para o tipo *timewise*, analisam uma ordem cronológica das mudanças por projeto analisando a data do *commit*. Após isto, ou seja, depois de ranquearem todos os projetos pela data de cada *commit*, agrupam as mudanças que foram realizadas dentro do mesmo mês. Após o agrupamento, dividem os dados em cinco partes iguais, executando com as partes uma espécie de *5-fold*.

Huang, Xia e Lo (2017) fazem uma análise global de esforço de modelos

supervisionados e não supervisionados que utilizam a predição JIT. Eles criticam a métrica LT (Linhas Totais), afirmam que deve ser ignorada, pois para coletá-la, é preciso fazer inspeções em todos os arquivos, e para sistemas muito grande isto é um problema. Os mesmos ainda executaram uma técnica supervisionada que denominaram classificação antes da ordenação *sorting* (*Classify Before Sorting* – CBS). Com o CBS, conseguiram os valores: uma revocação média de 45,8%, uma precisão (verdadeiros positivos divididos pela soma entre os verdadeiros positivos e os falsos positivos) de 28,7% e uma média harmônica (média hamônica entre a precisão e a revocação) de 32,9%, verificando 20% das linhas de código.

Yang *et al.* (2016) fazem um estudo onde investigam se modelos supervisionados podem ser melhores que modelos não supervisionados na abordagem JIT. O principal objetivo deles foi investigar modelos supervisionados simples para analisar suas eficácias em predizer defeitos JIT com consciência de esforço. Eles concluem afirmando que muitos modelos simples do tipo não supervisionados possuem desempenho melhor que os modelos supervisionados mais modernos.

Fukushima *et al.* (2014) fazem um estudo empírico direcionado a previsão JIT que foca em modelos utilizados anteriormente. A motivação deles é que para se treinar um preditor JIT é necessário se ter uma grande quantidade de *commits* já preparada e coletada de um histórico anterior do projeto, porém para projetos novos, é impossível de se ter um histórico anterior, sendo então necessário usufruir de projetos antigos para coletar seus históricos. Eles realizam esse estudo com onze projetos de código aberto. Concluem que modelos de projetos que precisam de alta performance, raramente o preditor apresenta bom desempenho; modelos em que apresentam correlação entre as variáveis preditoras, normalmente apresentam bom desempenho no preditor; técnicas de aprendizado em conjunto normalmente possuem bom desempenho de predição.

Yang *et al.* (2015) fazem um preditor JIT utilizando uma *Deep Learning* (apredizagem profunda). A motivação que apresentam é preencher uma lacuna que ainda não havia sido levada em consideração em preditores JIT, que é a utilização de uma técnica de aprendizagem profunda. Nos resultados, conseguem uma precisão média de 35,64%, uma revocação média de 69,03% e um F1 médio de 45,06%.

Como comentado anteriormente, ao optar-se pela técnica de predição que usufrui do JIT, pode-se ganhar tempo de revisão de código, economizando recursos e consequentemente obtendo possivelmente mais chances de êxito na correção, pois como a mudança é recente será mais facilmente identificada. As métricas escolhidas neste trabalho são as mesmas adotadas e compiladas por Kamei *et al.* (2013) e também utilizadas em parte por Yang *et al.*

(2017). Estas métricas apresentam fatores tanto de código fonte, quanto de experiência de desenvolvedores, como de objetivo da mudança e também histórico.

2.5 Conclusão do capítulo

Neste capítulo, expomos o estado da arte e motivações que levam trabalhos como este a serem realizados. Mostrou-se uma breve introdução a testes de software, mostrando os dois universos possíveis que são caixa branca e preta. Exibiu-se ainda um breve estudo sobre métricas de previsões por arquivos e também previsões JIT. Descreveu-se como as métricas criadas por Kamei *et al.* (2013) são computadas. Argumentou-se o motivo de escolha para predições do tipo JIT.

Nos estudos realizados nesta pesquisa, foram adotadas como técnicas de inteligência computacional: ANN e DT. Essas técnicas serão comentadas no capítulo seguinte. Os bancos de dados utilizados como teste neste trabalho são comentados com mais detalhes no capítulo 4.

3 TÉCNICAS DE INTELIGENCIA ARTIFICIAL UTILIZADAS

3.1 Redes neurais artificiais

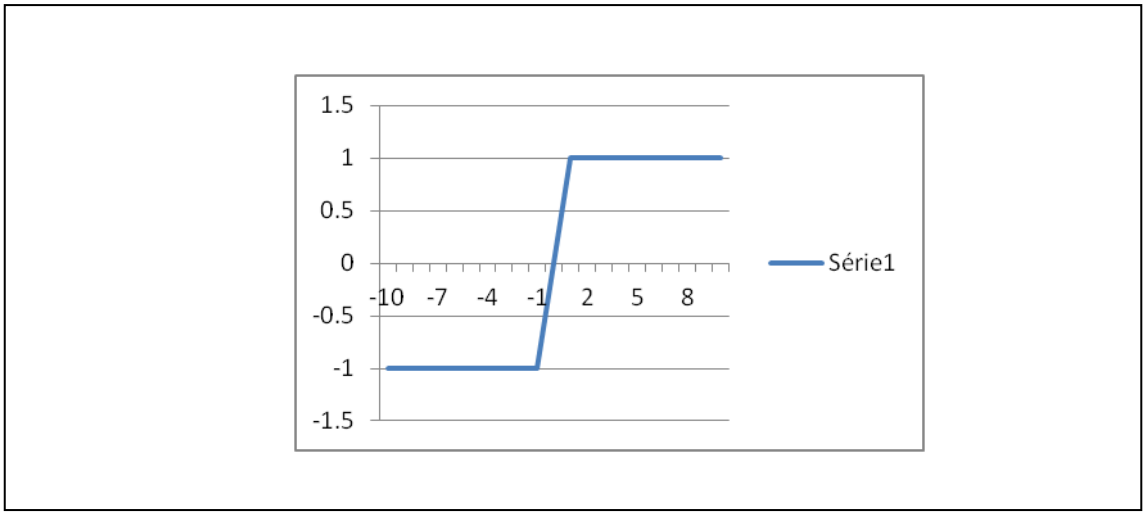
Uma ANN (*Artificial Neural Network*), redes neurais artificiais, é um modelo de inteligência computacional baseado no cérebro humano. Na técnica, os neurônios são um conjunto de unidades que operam como chaves de processamento, transmitindo informações quando um limiar de operação é alcançado.

Existem várias funções matemáticas que são utilizadas para determinar o limiar de ativação de um neurônio como, por exemplo, função linear (Equação 5) (Figura 4).

$$f(x) = \begin{Bmatrix} -1, \forall x \leq -1 \\ x, se -1 < x < 1 \\ 1, \forall x \geq 1 \end{Bmatrix} \quad (5)$$

A Figura 4 nos mostra um gráfico para a solução de (5). Observa-se que para valores de *x* menores iguais a -1, a equação nos retorna uma constante como resultado de valor -1. Caso o argumento da função pertença ao intervalo fechado entre -1 e 1, o retorno da função será o próprio argumento. Caso a entrada da função seja um número maior ou igual a 1, seu retorno será unitário.

Figura 4 – Função linear

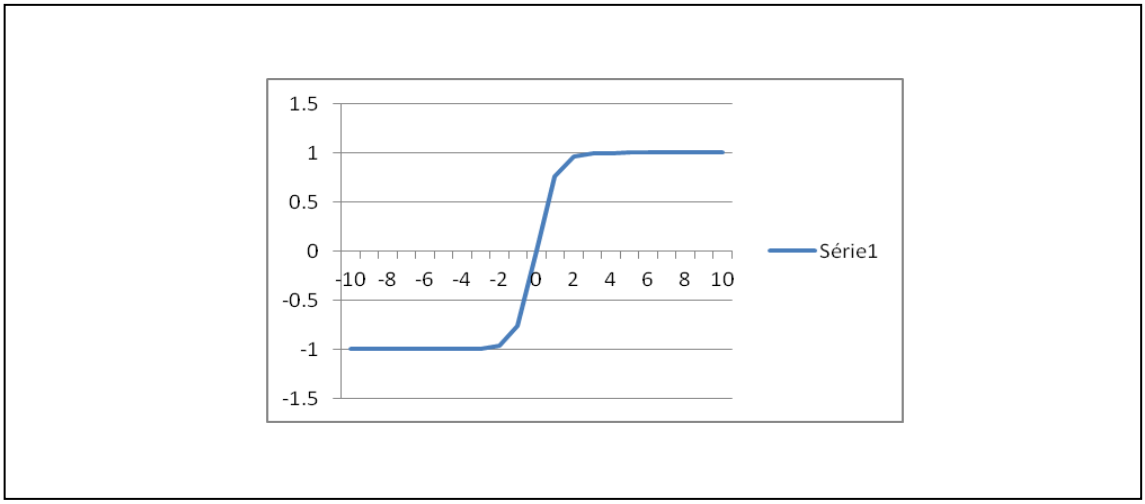


Fonte: O próprio autor.

Em (6) observamos a equação que define o comportamento da função tangente hiperbólica. Seu gráfico pode ser observado na Figura 5. A exemplo da função linear, descrita anteriormente, a função tangente hiperbólica também pode ser utilizada como função de ativação dos neurônios de uma rede neural artificial.

$$\tanh(y) = u(y) = \frac{senh(y)}{\cosh(y)} = \frac{e^{y} - e^{-y}}{e^{y} + e^{-y}}. \quad (6)$$

Figura 5 – Função tangente hiperbólica



Fonte: O próprio autor.

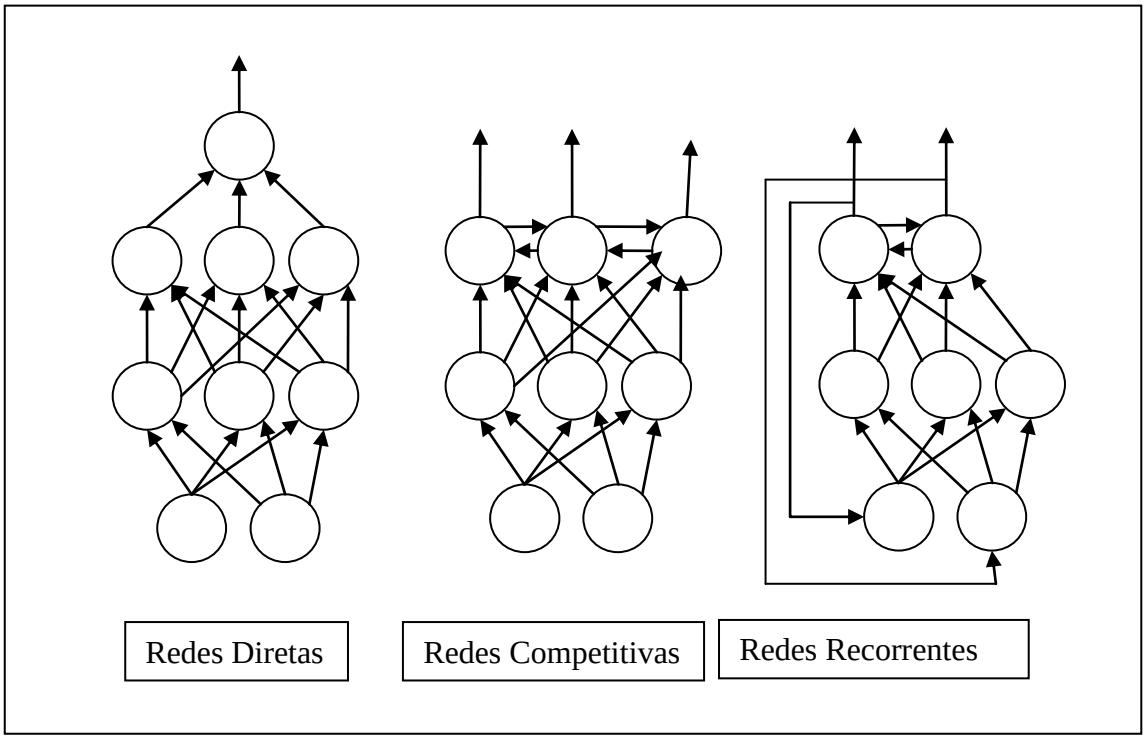
Na literatura existem muitos tipos de redes neurais artificiais. Historicamente os primeiros a desenvolverem um modelo matemático de um neurônio foram McCulloch e Pitts (1943). Rosenblatt (1958) criou o *perceptron*, uma rede neural de única camada, no qual mostrava que com ajustes de pesos, ou seja, sinapses ajustáveis, o modelo era capaz de aprender bem certos padrões linearmente separáveis de dados.

Uma década após, mais precisamente no ano de 1969, Minsky e Papert (1969) fizeram um trabalho crítico onde mostrava que a *perceptron* não seria capaz de resolver problemas não linearmente separáveis (MINSKY; PAPERT, 1969). Após isto, como mencionado em Lima, Pinheiro e Santos (2016) muitos pesquisadores ficaram frustrados e redirecionaram suas pesquisas para outros rumos, porém alguns pesquisadores perseveraram na pesquisa, entre eles temos Teuvo Kohonen, Stephen Grossberg, James Anderson e Kunihiko Fukushima.

A grande retomada das pesquisas com redes neurais se deu na década de 80, quando os pesquisadores Rumelhant, Hinton e Williams (1986), inspirados no trabalho de Werbos (1974), estabeleceram uma forma de retropropagar o erro através de camadas em uma rede neural. O algoritmo do *backpropagation* abriu novamente os olhos de pesquisadores para as redes neurais, surgindo assim várias pesquisas com aplicações e novos modelos de redes.

Uma rede neural pode apresentar topologias diferentes. Caso a comunicação (ligação) entre os neurônios seja apenas em um único sentido, ou seja, um neurônio só propaga informações com neurônio(s) de uma camada posterior, temos as redes chamadas *feedfoward*, que podem possuir uma ou mais camadas. Caso a comunicação entre as unidades seja em vários sentidos, ou seja, mais de uma direção, a rede neural é denominada cíclica. As redes com ciclo podem ser recorrentes ou competitivas (LIMA; PINHEIRO; SANTOS, 2016). A rede será recorrente quando houver ligações entre a camada de saída e a camada de entrada. A rede será competitiva quando nos existir mais de um neurônio de saída com ligações entre eles bidirecional. Exemplos de topologias de redes neurais podem ser observadas na Figura 6.

Figura 6: Exemplos de topologias de redes neurais



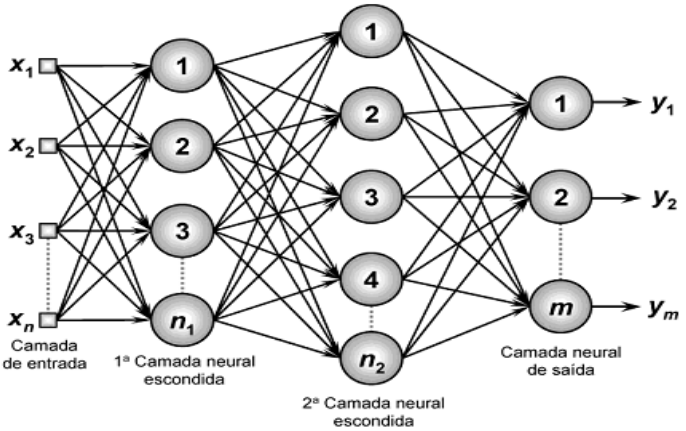
Fonte: O próprio autor.

Uma ANN do tipo MLP (*MultiLayer Perceptron*), segundo Haykin (2001), possui pelo menos três camadas (Figura 7): 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 nas conexões e dos valores de *bias* (uma constante que é associado ao peso com o intuito de deslocar a função de ativação no eixo) nos neurônios (ARAR; AYAN, 2015) (GAYATHRI; SUDHA, 2014).

Em (7) é demonstrado o modelo de cálculo para um neurônio artificial (ARAR; AYAN, 2015), onde y_i é a saída do neurônio *i*, *nt* é o número total de entradas para este neurônio provenientes das entradas da ANN ou então provenientes das saídas de uma camada de neurônios anterior, 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 (6), que representa a equação de uma função do tipo tangente hiperbólica.

$$y_i = f_i\left(\sum_{j=1}^{nt} \mathrm{x_j}\, w_{ij} + \theta_i\right) \tag{7}$$

Figura 7: Exemplo de uma ANN - MLP



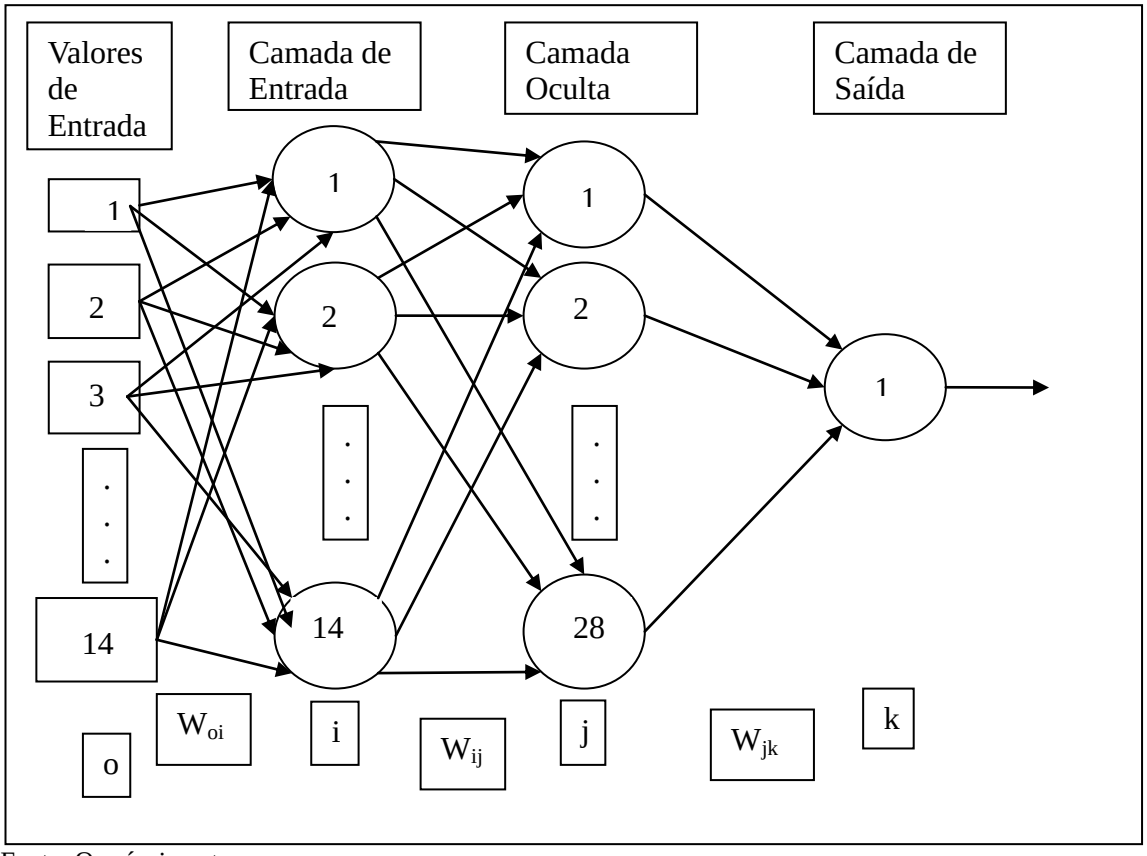
Fonte: Batista, (2012).

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*, para a determinação dos valores dos pesos e *bias* da ANN (HAYKIN, 2001).

A entrada da ANN desenvolvida neste trabalho é composta de 14 entradas, relacionadas com as métricas descritas na Tabela 2. A arquitetura da rede proposta possui um neurônio de saída que indica se o software apresenta erro ou não. Uma camada oculta é utilizada, com 28 neurônios. A função tangente hiperbólica foi escolhida como a função de ativação nos neurônios da camada escondida e para saída uma função linear. A escolha da função tangente hiperbólica para os neurônios da camada oculta garante a capacidade da rede neural de resolver problemas não lineares. O algoritmo de treinamento da rede utilizado foi o método de Levenberg–Marquardt (YU; WILAMOWSKI, 2011). Diferente do *backpropagation*, o método de Levenberg-Marquardt trabalha com a matriz da segunda derivada do erro (matriz hessiana) o que garante em diversas situações a convergência mais rápida na determinação de pesos e *bias* da rede em treinamento (SILVA, L. N. D. C, 1998). A Figura 8 exibe um esboço da arquitetura escolhida.

Figura 8: Arquitetura da rede neural proposta



Fonte: O próprio autor.

Realizaram-se ainda experimentos com outra arquitetura de rede neural do tipo MLP. Nela, utilizamos duas camadas ocultas com 14 e 28 neurônios cada, uma camada de entrada de 14 neurônios e uma camada de saída com apenas um neurônio. As funções de ativação nas camadas de entrada e oculta foram do tipo tangente hiperbólica e na camade de saída uma função linear.

O algoritmo de treinamento da rede neural escolhido é o *backpropagation* para ambos os cados, utilizando o método de Levenberg-Marquardt. Inicialmente Levenberg (1944), propôs uma melhoria do método de Newton para que problemas não lineares conseguissem convergir mais rápido. Seja (8) uma função não linear, na qual (LEVENBERG, 1944):

$$F(Y) = X, \tag{8}$$

onde (9), (LEVENBERG, 1944):

$$X \epsilon R^M \; e \; Y \epsilon R^N \tag{9}$$

Sabendo que X e Y são vetores e que M é maior ou igual a N. Precisamos estimar um valor Y' que se aproxime ao máximo de X' constatado. Uma reformulação plausível para o caso seria (10), (LEVENBERG, 1944):

$$X' = F(Y') + \varepsilon \tag{10}$$

No qual ε seria um erro mínimo aceitável para a aproximação. Como descrito por Press *et al. (*1992), o método de Newton melhora (10), a partir das seguintes suposições descritas em (11) e (12), (LEVENBERG, 1944):

$$F(Y' + \Delta) = F(Y') + J\Delta \tag{11}$$

$$J = \frac{\partial X}{\partial Y} \tag{12}$$

J é o Jacobiano e Δ é um pequeno incremento de Y'. Para diminuir o erro o máximo possível, deve-se aplicar (13), (LEVENBERG, 1944):

$$||\varepsilon - J\Delta|| \quad (13)$$

Que resulta em resolver (14), (LEVENBERG, 1944):

$$J^T J\Delta = J^T \varepsilon \quad (14)$$

A solução de (14) melhorada é sugerida por (15), podendo se ajustar mais precisamente a cada iteração, (LEVENBERG, 1944):

$$Y'r = Y' + \Delta \quad (15)$$

Levenberg (1944) sugeriu a equação (14) para acelerar a convergência de (14), onde *In* representa a matriz identidade, (LEVENBERG, 1944):

$$(J^T J + In\lambda)\Delta = J^T \varepsilon \quad (16)$$

Marquardt (1963) propôs uma melhoria da equação (16), pois constatou-se que caso o λ crescesse muito, o método apresentava instabilidades. A solução que foi sugerida trabalha na ponderação de cada componente do gradiente de acordo com a sua curvatura (17).

$$(J^T J + diag(J^T J)\lambda)\Delta = J^T \varepsilon \quad (17)$$

Um exemplo de onde a técnica de treinamento é aplicada seria nos ajustes de pesos (18). Para os pesos, um instante de tempo futuro é calculado a partir do valor do peso atual diminuído pela multiplicação do Jacobiano e erro, dividido pela soma do produto entre o Jacobiano transposto pelo Jacobiano e o produto entre o valor de ajuste pela matriz identidade (FAUSETT, 1994).

$$w(t+1) = w(t) - (J^T J + \lambda I)^{-1} J\varepsilon \quad (18)$$

3.2 Árvore de decisão

Uma DT (*Decision Tree*) é um modelo preditivo que pode ser utilizado para

representar tanto um modelo de classificação como também um modelo de regressão (ROKACH; MAIMON, 2008). Quando uma DT é utilizada como classificador é normalmente denominada Árvore de Classificação (AC) e quando utilizada para regressão é denominda Árvore de Regressão (AR).

Uma AC é utilizada para classificar um objeto ou instância (vetor de atributos) 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; 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 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; KOUTROUMBAS, 2006).

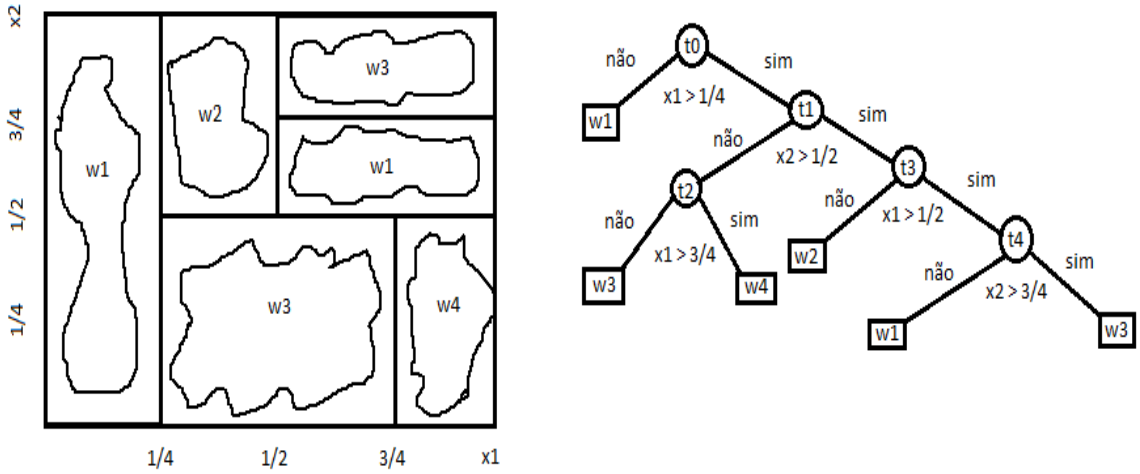
Na Figura 9 é exemplificado uma DT tipo AC, separando o espaço de entradas em hiperplanos com retas paralelas aos eixos. A sequência de decisões é aplicada para cada atributo da instância apresentada à árvore, com os testes de decisão associados aos nós sendo na forma:

$$Se\ x_i\ \beta\ \alpha\ Então\ c_1\ Senão\ c_2, \tag{19}$$

onde x_i representa o atributo avaliado; β 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.

Existem vários algoritmos já conhecidos e amplamente utilizados para construção de uma DT, entre eles podemos citar o C4.5 mostrado em Quinlan (2014) e o CART (*Classification and Regression Tree*) exibido em Breiman (1996) e formulado em Breiman *et al.* (1984). 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 em Scikit-Learn (2019a).

Figura 9: Partição do espaço de variáveis e regras obtidas da AC



Fonte: Adaptado pelo autor de Theodoridis e Koutroumbas (2006).

O CART trabalha na construção de árvores de acordo com os recursos e considerando o limite que produz o maior ganho de informação. A particularidade da biblioteca utilizada é que ainda não existe suporte para variáveis discretas, (TREE ALGORITHMS, 2019).

Considerando a formulação matemática do CART, baseado em (BREIMAN *et al.*, 1984), em (20) o x_i representa um vetor de treinamento, n representa a quantidade de atributos e l representa a classe em que pertence (*label*); onde (TREE ALGORITHMS, 2019):

$$x_i \in R^n, i = 1, \dots, l \tag{20}$$

onde (21) y é um vetor de rótulos (TREE ALGORITHMS, 2019) e (BREIMAN *et al.*, 1984):

$$y \in R^l. \tag{21}$$

Uma DT particiona recursivamente o espaço amostral, agrupando amostras com o mesmo rótulo. Admitindo que os dados em um determinado nó m são representados por Q. θ é a divisão dos recursos de treinamento. Cada atributo do vetor de treinamento participará da

divisão (22) que consiste em um recurso *j* e um limite t_m, particionando logo em seguida os dados em dois subconjuntos conforme (23) e (24) (TREE ALGORITHMS, 2019) e (BREIMAN *et al.*, 1984).

$$\theta = (j, t_m) \tag{22}$$

$$Q_{esq}(\theta) = (x, y)|x_j \leq t_m \tag{23}$$

$$Q_{dir}(\theta) = Q \backslash Q_{esq} \tag{24}$$

A impureza (25) de um nó *m* é calculada selecionando os parâmetros que a minimizam (26) e depende da escolha da função de impureza, que pode ser a de Gini (27), Entropia (28) ou *Misclassification* (29), ambas para as árvores do tipo classificação (TREE ALGORITHMS, 2019) e (BREIMAN *et al.*, 1984).

$$G(Q, \theta) = \frac{n_{esq}}{N_m} H\left(Q_{esq}(\theta)\right) + \frac{n_{dir}}{N_m} H\left(Q_{dir}(\theta)\right) \tag{25}$$

$$\theta^* = argmin_\theta G(Q, \theta) \tag{26}$$

$$H(X_m) = \sum_k p_{mk}(1 - p_{mk}) \tag{27}$$

$$H(X_m) = -\sum_k p_{mk} \log\left(p_{mk}\right) \tag{28}$$

$$H(X_m) = 1 - \max\left(p_{mk}\right) \tag{29}$$

Onde X_m representa os dados de treinamento para o nó *m* e p_{mk} (30) é a proporção de observação de uma determinada classe *k* em um determinado nó *m*, onde se possui uma região R_m com N_m observações, (TREE ALGORITHMS, 2019) e (BREIMAN *et al.*, 1984).

$$p_{mk} = 1/N_m \sum_{x_i \in R_m} I(y_i = k) \tag{30}$$

3.3 Conclusão do capítulo

Neste trabalho utilizou-se como técnicas de inteligência computacional redes neurais artificiais do tipo MLP e árvore de decisão DT do tipo classificação. Ambas as ferramentas

são aplicadas para treinamento, teste e validação de um sistema preditivo de defeitos em software. Essas técnicas foram discutidas nesse capítulo.

O banco de dados utilizados e o tratamento dos dados serão comentados no próximo capítulo. A base de dados é a mesma desenvolvida e utilizada por Kamei *et al. (*2013) e aplicada também por Yang *et al. (*2017).

4 DADOS UTILIZADOS NO TREINAMENTO E MODELO DE APLICAÇÃO

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 mesmos 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. Estas alterações podem ter sido motivadas por correção de erros, implementações de novas funcionalidades ou alteração de alguma parte do software que se ache necessária. 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 de saída com detecção de erro ou não. Os parâmetros são comentados no capítulo 2 (Tabela 2). Alguns desses parâmetros de entrada indicam métricas focadas na qualidade do código fonte do software, entre eles temos as dimensões de difusão e tamanho. Têm-se ainda medidas baseadas no histórico do sistema, propósito da mudança e experiência dos programadores.

A Tabela 3 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.

Tabela 3: 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).

Para a realização do treinamento para o diagnóstico preditivo de falhas em software,

inicialmente, todos os casos de erros foram separados para serem utilizados na etapa. Feito isto, selecionou-se aleatoriamente de todas as bases uma quantia igual para casos de *commits* que eram considerados sem erro, prática semelhante à Kamei *et al.* (2013), sendo justificado em Kamei *et al.* (2007) e Khoshgoftarr, Yuan e Allen (2000).

Essa abordagem se faz necessária devido ao grande desbalanceamento de casos de erro em relação aos casos de não erro. O desbalanceamento causa piora significativa nos modelos de predição (KAMEI *et al.*, 2013), (KAMEI *et al.*, 2007) e (KHOSHGOFTARR, YUAN e ALLEN 2000).

Escolheu-se aplicar a validação cruzada tipo 10-*fold*, numa tentativa de minimizar situações de *overfitting*. Essa abordagem também é utilizada em (KAMEI *et al.*, 2013). A técnica é descrita em Efron (1983), onde consiste em subdivir o espaço amostral dos dados em dez partes iguais. Utilizam-se nove das dez partes para treinamento e a restante para testes. O ciclo do *10-fold* possui dez iterações, fazendo com que todas as partes sejam utilizadas em treinamento e testes. No final, verificam-se os dez resultados de exatidão e escolhe-se o melhor caso. Essa abordagem foi utilizada durante o desenvolvimento dos métodos de predição de erro em software adotando ANN ou DT.

Após treinado, testado e validado com *10-fold*, e escolhido o melhor caso, que nos entregou uma melhor revocação, durante o desenvolvimento do método de diagnóstico, e aplicamos o modelo selecionado em todos os dados de cada base (Bugzilla, Columba, JDT, Mozilla, Platform e Postgres) para observarmos e coletarmos os dados estatísticos e de desempenho que o método preditivo foi capaz de entregar.

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

a) 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;

b) Semelhante a (KAMEI *et al*., 2013), os valores de LA e LD (Tabela 2) 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 b 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 (31). Essa nova normalização na DT é utilizada, porém, sem balancear os dados entre casos sem erro e com erro:

$$z = \frac{x - med}{s}, \quad (31)$$

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

Para análise estatística dos resultados obtidos, foram adotados os parâmetros: Exatidão, Precisão, Revocação e a média harmônica - F1 (um tipo de média entre precisão e revocação). 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 no software analisado não resultaram em erro e o método de diagnóstico 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 uma proporção geral de acertos. Matematicamente 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 (KAMEI *et al.*, 2013):

$$Exatidão = \frac{(TP+TN)}{(TP+TN+FP+FN)}. \quad (32)$$

A Precisão avalia os casos de previsões corretas de determinada classe, no nosso caso os classificados como positivos, defeitos, previsões que foram classificadas como defeito. Em linhas gerais, seria o quão a previsão pode evitar os casos de falsos positivos, FP. Matematicamente é a divisão entre casos que foram classificados corretamente como defeitos (TP), pela soma dos casos TP com os casos classificados incorretamente como defeitos (FP) (KAMEI et al., 2013):

$$Precisão = \frac{TP}{TP+FP}. \quad (33)$$

A Revocação avalia se os casos classificados como de uma determinada classe realmente estão corretos. Indica o quão foi boa a classificação em relação à escolha de casos corretos. Está interessada na taxa de elementos relevantes. Em linhas gerais, o quão a previsão é capaz de evitar os falsos negativos, FN, levando em consideração os casos classificados como verdadeiros positivos, TP. É calculada levando em consideração os casos de determinada classe em relação ao total de casos que pertencem àquela classe. Para a abordagem realizada no trabalho, é considerada a mais importante. No nosso caso, são levados em conta os casos que o classificador disse que era defeito e estava correto (TP) e os

casos que o classificador disse que não era defeito, porém era defeito (FN). Matematicamente é a divisão entre casos classificados acertadamente como defeitos (TP), pela soma dos casos TP com os casos classificados erroneamente como não defeitos (FN) (KAMEI *et al.*, 2013):

$$Revocação = \frac{TP}{TP+FN}. \quad (34)$$

A medida F1, no nosso caso, leva em consideração que os casos de verdadeiros positivos (TP) e os casos de falsos negativos (FN) são os mais importantes na classificação. Matematicamente representa a média harmônica entre Precisão e Revocação (KAMEI *et al.*, 2013):

$$F1 = \frac{2*REVOCAÇÃO*PRECISÃO}{REVOCAÇÃO+PRECISÃO}. \quad (35)$$

A média aritmética simples, a mediana e o desvio padrão foram utilizadas como medidas estatísticas relacionadas com exatidão, precisão, revocação e média harmônica F1 de cada resultado obtido por base de dados. A média aritmética simples (TRIOLA, 1999):

$$med = \frac{\sum x}{n}, \quad (36)$$

onde *x* é o valor da variável utilizada para representar os dados que serão somados e *n* é a quantidade de sua totalidade.

A mediana, segundo Triola (1999), para um conjunto de valores ordenados é considerado o valor do meio desse conjunto. Para efeitos de cálculo, caso a quantidade de valores seja impar, o valor do meio será considerada a mediana, caso contrário, será uma média aritmética simples dos dois valores do meio.

O desvio padrão, definido em Pearson (1894):

$$s = \sqrt{\frac{\sum(x-med)^2}{n}} \quad (37)$$

Onde *x* é o valor da variável e *n* é a quantidade.

4.1 Modelos de configurações para treinamento e teste

O processo de treinamento e testes das técnicas de inteligência computacional aplicadas neste trabalho, basicamente se subdivide em cinco partes:

1. Balanceamento dos dados: todos os casos que são *commits* de defeito juntos com igual quantidade de casos que não são defeitos, sendo que os casos que não são defeitos devem ser selecionados aleatoriamente.

2. Configuração da técnica de inteligência computacional aplicada.
    a. Um exemplo para a rede neural artificial: 14 neurônios na camada de entrada, 28 neurônios na camada oculta e 1 neurônio na camada de saída. Na camada oculta é utilizada a função de ativação do tipo tangente hiperbólica para garantir a capacidade de não linearidade da técnica. Para a camada de saída pode-se aplicar como função de ativação, uma função linear. Foi adotado uma taxa de aprendizado com valor inicial de 0,3, e ajuste automatico durante o treinamento. O número de épocas de treinamento foi de 5000. Foi utilizado o algoritmo de treimento de Levenberg-Marquardt.
    b. Para o caso da árvore de decisão: critério de divisão de nós baseado no método de Gini. Máxima profundidade da árvore de 30 níveis. Amostragem mínima para divisão será dois. Amostragem mínima para folha será um. Ativar o retreino e utilização de revocação como principal pontuação.
3. Aplicação da téncica de validação cruzada 10-*fold* para treino e teste da árvore de decisão ou da rede neural artificial.
4. Validar o resultado da técnica com todos os dados da base, ou seja, a base de dados original sem o balanceamento descrito na parte 1.
5. Verificar e salvar a rede neural ou árvore de decisão que apresentou o melhor resultado de revocação e utilizar como preditor para os próximos *commits* do projeto.

4.2 Conclusão do capítulo

Foi exposto neste capítulo a maneira como os dados são apresentados, tratados e treinados. Apresentamos uma nova forma de separação de amostras para treinamento como retratada em Kamei *et al.* (2013), sendo justificado em Kamei *et al.* (2007) e Khoshgoftarr, Yuan e Allen (2000). Essa forma se faz necessária para melhorar o desempenho do preditor em casos que os dados apresentados são fortemente desbalanceados em relação à classe pertencente. Tratamos ainda de equacionar as medidas utilizadas para cálculo de desempenho das técnicas comparadas.

Tambem apresenta-se na Seção 4.1 um modelo de configuração para treinamento e testes das técnicas de inteligência computacional aplicada nesta dissertação. Todos os

pormenores de configuração e escolha estão descritos.

No próximo capítulo, são exibidos os resultados obtidos com as ANN e as DT desenvolvidas neste artigo, e comparados com os trabalhos de (KAMEI *et al.,* 2013) e (YANG *et al.*, 2017).

5. 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, (ANN1) com 14 parâmetros de entrada, um neurônio de saída que indica se o *software* apresenta erro ou não. Duas camadas ocultas são adotadas, com 14 e 28 neurônios cada. O mesmo tratamento de dados adotado em Kamei *et al.* (2013) e enfatizado no Capítulo 4 é ultilizado. Considerando ainda as mesmas configurações para treinamento e funções de ativação, foi desenvolvida também outra ANN, a ANN 2, do tipo MLP, com 14 neurônios na camada de entrada, apenas uma camada oculta com 28 neurônios e um único neurônio na camada de saída. Vale ainda ressaltar que na segunda ANN foi realizado um balanceamento dos dados por base, garantindo que tenhamos iguais quantidades de casos de erros e sem erros para cada base dentro da amostra de treinamento e testes que são utilizados na validação cruzada.

Tabela 4: Resultados DT de Kamei *et al*. (2013)

Base	**Exatidão**	**Precisão**	**Revocação**	**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 pelo autor de Kamei *et al.* (2013).

Também foi desenvolvido neste trabalho três árvores de decisão (DT) para o diagnóstico de softwares, adotando o algoritmo CART através do PYTHON. Os dados de treinamento e validação foram tratados conforme descrito no capítulo 4. Na DT 1, os dados não foram balanceados e foi escolhida a árvore que apresentou o melhor resultado de Revocação na validação cruzada; na DT 2 os dados foram balanceados e optou-se na escolha da árvore que apresentou a melhor Exatidão na validação cruzada; na DT 3 também os dados foram balanceados porém a escolha foi a árvore que apresentou a melhor Revocação na

validação cruzada.

Na Tabela 4 são apresentados os resultados obtido por (KAMEI *et al.*, 2013) utilizando o mesmo banco de dados de teste adotado neste trabalho, e utilizando uma DT no diagnóstico de falhas em software.

Os resultados obtidos da ANN 1 proposta neste trabalho são indicados na Tabela 5, e os resultados da DT 1 desenvolvidas pelo autor são apresentados na Tabela 6. 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 4, na cor vermelha são métricas com resultados inferiores e na cor preta são resultados considerados semelhantes.

Tabela 5: Resultados ANN 1

Base	Exatidão	Precisão	Revocação	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: O próprio autor.

Tabela 6: Resultados DT 1

Base	Exatidão	Precisão	Revocação	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: O próprio autor.

Comparando os resultados das Tabelas 4 e 5. Os valores de Revocação obtidos da rede neural proposta foram melhores em todas as bases de dados. Já os valores de Precisão foram melhores em quatro das bases de dados (Bugzilla, JDT, Mozilla e Platform) e piores em duas (Columba e Postgres). As médias harmônicas F1 foram melhores em quatro bases (Bugzilla, JDT, Mozilla e Platform) e semelhantes em duas (Columba e Postgres). As Exatidões foram melhores em três bases de dados (Bugzilla, Mozilla e Platform), piores em duas (Columba e Postgres) e semelhante em uma (JDT). 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 4 e 5, foram obtidos valores de Média e Mediana melhores na Revocação da ANN 1 desenvolvida, mas um Desvio Padrão maior se comparado a (KAMEI *et al.*, 2013). Foi verificado ainda que na média harmônica F1, foram obtidos 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 DT 1 proposta (Tabela 6) são bem superiores ao de (KAMEI *et al.*, 2013). Comparando os resultados das Tabelas 4 e 6, os valores de Revocação, 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). Dos 12 valores estatísticos, apresentou-se inferioridade apenas no Desvio Padrão da Revocação.

Comparando os resultados entre as Tabelas 5 e 6, mostram que das 24 medidas possíveis, a ANN 1 só superou a DT 1 em apenas 3 casos. Nas medidas estatísticas, a DT 1 superou a ANN 1 em todas.

Na Tabela 7 são apresentados os resultados obtidos por (Yang *et al.*, 2017). O mesmo faz um estudo e também faz comparações com (KAMEI *et al.*, 2013), utilizando os mesmos bancos de dados de teste. Já a Tabela 8 representa os mesmos resultados da DT 1 desenvolvida neste trabalho (Tabela 6), entretanto a codificação de cores é feita em comparação com os resultados da Tabela 7: resultados melhores na DT 1 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.

Ao compararmos os resultados da Tabela 8, DT 1 desenvolvida neste trabalho, com os da Tabela 7, que utiliza uma RD (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. Na Revocação, podemos considerar dois empates técnicos: Bugzilla da Tabela 7 com 75,92% e da DT 1 com 75%; Mozilla da Tabela 7 com 77,75% e da DT 1 com 77%. Já para a base de dados Platform, o resultado na Tabela 7 foi superior em 1,48%. Em relação aos valores estatísticos, para a Revocação, a DT 1 proposta só perde no quesito Desvio Padrão, apresentado superioridade em todas as outras medidas.

Tabela 7: Resultados RD

Base	Precisão	Revocação	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 8: Resultados DT 1

Base	Precisão	Revocação	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: O próprio Autor.

As Tabelas 9 e 10 mostram os resultados obtidos com uma (ANN 2) e uma (DT 2) balanceada por banco de dados individualmente. Esse balanceamento foi feito analisando a quantidade de rótulos do tipo erro que cada base contribuía, resgatando a mesma quantidade para os casos de não erro por base de dados individual. Agrupou-se todos esses dados selecionados e formou-se uma amostra para treinamento e teste também aplicando o *10-fold* como técnica. Após o treinamento, escolheu-se o melhor caso e aplicou-se em todos os dados de cada base para obtenção dos resultados de validação.

Tabela 9: Resultados ANN 2 comparados com Kamei *et al.* (2013)

Base	Exatidão	Precisão	Revocação	F1
Bugzilla	**74,2%**	**64,5%**	**66,1%**	**65,3%**
Columba	**70,9%**	**51,6%**	**74,9%**	**61,1%**
JDT	**73,8%**	**31,3%**	**68,7%**	**43,0%**
Mozilla	**70,1%**	**12,5%**	**78,1%**	**21,5%**
Platform	**70,6%**	**30,1%**	**75,1%**	**42,9%**
Postgresql	**75,1%**	**50,2%**	**74,0%**	**59,8%**
Média	**72,5%**	**40,0%**	**72,8%**	**48,9%**
Mediana	**72,3%**	**40,7%**	**74,5%**	**51,4%**
Desvio Padrão	**2,1%**	**18,8%**	**4,5%**	**16,5%**

Fonte: O próprio autor.

Tabela 10: Resultados DT 2 comparados com Kamei *et al.* (2013)

Base	Exatidão	Precisão	Revocação	F1
Bugzilla	**74%**	**63%**	**73%**	**67%**
Columba	**65%**	**46%**	**83%**	**59%**
JDT	**68%**	**27%**	**75%**	**40%**
Mozilla	**64%**	**11%**	**86%**	**20%**
Platform	**68%**	**29%**	**78%**	**42%**
Postgresql	**73%**	**47%**	**77%**	**59%**
Média	**69%**	**37%**	**79%**	**48%**
Mediana	**68%**	**37%**	**77%**	**50%**
Desvio Padrão	**4%**	**17%**	**5%**	**16%**

Fonte: O próprio autor.

Pode-se observar que a Tabela 9 é superior a Kamei *et al.* (2013) em 30 medidas (marcados em azul), inferior em 4 medidas (marcadas em vermelho) e empate técnico em

duas medida (marcada em preto). Já para a Tabela 10 em relação a Kamei *et al. (*2013), observou-se superioridade em 21 medidas (marcadas em azul), inferioridade em 11 medidas (marcadas em vermelho) e igualdade técnica em 4 medidas (marcadas em preto).

Se analisarmos apenas os fatores estatísticos, pode-se observar na Tabela 9 que para Média e Mediana da Exatidão, Precisão, Revocação e média harmônica F1 são expostos valores superiores que os apresentados em (Kamei *et al.*, 2013). O Desvio Padrão só é superior para o caso da Exatidão.

A Tabela 10 apresenta dados estatísticos superiores nas Médias da Precisão, Revocação e média harmônica F1. Já a Mediana é superior para os casos de Revocação e média harmônica F1. Para o Desvio Padrão, exibiram-se três empates técnicos, Exatidão, Precisão e media harmônica F1. Já o Desvio Padrão da Revocação é inferior. Ao se comparar com (Kamei *et al.*, 2013).

Agora analisaremos os resultados da ANN 2 e da DT 2 em comparação com (YANG *et al.* 2017). Os resultados obtidos por Yang *et al.* (2017) foram exibidos na Tabela 7.

Tabela 11: Resultados ANN 2 comparados com Yang *et al.* (2017)

Base	Precisão	Revocação	F1
Bugzilla	64,5%	66,1%	65,3%
Columba	51,6%	74,9%	61,1%
JDT	31,3%	68,7%	43,0%
Mozilla	12,5%	78,1%	21,5%
Platform	30,1%	75,1%	42,9%
Postgresql	50,2%	74,0%	59,8%
Média	40,0%	72,8%	48,9%
Mediana	40,7%	74,5%	51,4%
Desvio Padrão	18,8%	4,5%	16,5%

Fonte: O próprio autor.

Tabela 12: Resultados DT 2 comparados com Yang *et al.* (2017)

Base	Precisão	Revocação	F1
Bugzilla	63%	73%	67%
Columba	46%	83%	59%
JDT	27%	75%	40%
Mozilla	11%	86%	20%
Platform	29%	78%	42%
Postgresql	47%	77%	59%
Média	37%	79%	48%
Mediana	37%	77%	50%
Desvio Padrão	17%	5%	16%

Fonte: O próprio autor.

Podemos observar na Tabela 11 que a ANN 2 foi superior a Yang *et al.* (2017) em apenas 4 medidas, sendo elas a Precisão no Bugzilla e JDT; e a média harmônica F1 no Columba e JDT. Considerando ainda valores computados nas bases de dados, observamos que ocorreram 7 empates técnicos, sendo dois de medidas estatísticas (Média e Mediana da Precisão), dois na Precisão (Columba e Postgres), dois no Revocação (Columba e Mozilla) e um na média harmônica F1 (Postgres). Temos ainda 16 casos de inferioridade: Precisão (Mozilla, Platform, Desvio Padrão), Revocação (Bugzilla, JDT, Platform, Postgres, Média,

Mediana e Desvio Padrão), e média harmônica F1 (Bugzilla, Mozilla, Platform e Média, Mediana e Desvio Padrão).

Para a Tabela 12, constatamos que a DT 2 foi superior a Yang *et al.* (2017) em 8 medidas, na Precisão (Bugzilla), no Revocação (Columba, JDT, Mozilla, Platform, Postgres, Média e Mediana). Observamos ainda dois empate técnico na Precisão e média harmônica F1 (Desvio Padrão) e 17 valores inferiores: Precisão (Columba, JDT, Mozilla, Platform, Postgres, Média e Mediana), Revocação (Desvio Padrão), e média harmônica F1 (Bugzilla, Columba, JDT, Mozilla, Platform, Postgres, Média e Mediana).

Tabela 13: Resultados DT 3 comparados com Kamei *et al.* (2013)

Base	Exatidão	Precisão	Revocação	F1
Bugzilla	73%	61%	70%	65%
Columba	65%	46%	82%	59%
JDT	68%	27%	73%	40%
Mozilla	65%	12%	84%	20%
Platform	67%	28%	77%	41%
Postgresql	71%	46%	76%	57%
Média	68%	37%	77%	47%
Mediana	68%	37%	76%	49%
Desvio Padrão	3%	16%	5%	15%

Fonte: O próprio autor.

Tabela 14: Resultados DT 3 comparados com Yang *et al.* (2017)

Base	Precisão	Revocação	F1
Bugzilla	61%	70%	65%
Columba	46%	82%	59%
JDT	27%	73%	40%
Mozilla	12%	84%	20%
Platform	28%	77%	41%
Postgresql	46%	76%	57%
Média	37%	77%	47%
Mediana	37%	76%	49%
Desvio Padrão	16%	5%	15%

Fonte: O próprio autor.

Analisaremos agora os resultados da árvore DT 3 comparados com (Kamei *et al.*, 2013) e (Yang *et al.*, 2017). Na Tabela 13 comparamos a DT 3 com (Kamei *et al.*, 2013). É exibido que a DT 3 foi superior em 20 resultados, na Exatidão (Bugzilla e Desvio Padrão), na Precisão (Bugzilla, JDT e Platform), no Revocação (Bugzilla, Columba, JDT, Mozilla, Platform, Postgres, Média e Mediana) e na média harmônica F1 (Bugzilla, Columba, JDT, Platform, Postgres, Média e Mediana). Ocorreram 12 valores inferiores: Exatidão (Columba, JDT, Mozilla, Postgres, Média e Mediana), Precisão (Columba, Mozilla, Postgres e Mediana), Revocação (Desvio Padrão), e média harmônica F1 (Mozilla). Observamos ainda 4 empates técnicos: Exatidão (Platform); Precisão (Média e Desvio Padrão), e média harmônica F1 (Desvio Padrão).

Na Tabela 14 comparamos a DT3 com (Yang *et al.*, 2017). Observamos que a DT 3 foi superior em 4 medidas: Precisão (Desvio Padrão) e Revocação (Columba, Mozilla e Média). Em 19 casos observamos inferioridade: Precisão (Bugzilla, Columba, JDT, Mozilla, Platform, Postgres, Média e Mediana), Revocação (Bugzilla, Postgres e Desvio Padrão), e média

harmônica F1 (Bugzilla, Columba, JDT, Mozilla, Platform, Postgres, Média e Mediana). Em 4 casos observamos empates técnicos: Revocação (JDT, Platform e Mediana) e média harmônica F1 (Desvio Padrão).

A árvore DT 3 apresenta Exatidão média de 68% e Revocação média de 77%. Valores de Revocação superiores a (Kamei *et al.,* 2013) e (Yang *et al.,* 2017). Já para os valores de Exatidão, possuem uma inferioridade de aproximadamente 2,67% em relação a que é apresentada em (Kamei *et al.,* 2013).

5.1 Conclusão do capítulo

Podemos observar que das duas técnicas de inteligência computacional utilizadas neste trabalho, ANN e DT, foi alcançado os objetivos gerais do trabalho. Considerando que a Revocação é considerada a medida mais importante, pois os casos de verdadeiros positivos, ou seja, predições corretas para os casos de defeitos, realmente importam mais do que os casos de predições corretas para os casos de não defeitos, verdadeiros negativos.

A segunda medida mais importante a ser considerada é a Exatidão, pois ela nos oferece uma taxa total de acertos considerando ambos os casos, defeitos ou não. Em ambas as técnicas, ANN e DT, alcançamos bons resultados, em comparação com os trabalhos de referência citados (Kamei *et al.,* 2013) e (Yang *et al.,* 2017).

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 como teste para o diagnóstico preditivo de falhas em *softwares*, foram obtidos taxas de Revocação média de 74,6% e de 78%, adotando, respectivamente, a rede ANN 1 com duas camadas oculta e uma árvore DT 1 com milhares de nós no processo de detecção de erros. Após refinarmos as técnicas, obtivemos taxas de revocação média de 72,8% e 78% para uma ANN 2 e DT 2 respectivamente, onde a ANN 2 tinha apenas uma camada oculta e a DT 2 tinha 1136 nós. 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. Desenvolvemos ainda uma nova DT 3, na qual possui apenas 382 nós e apresenta exatidão média de 68% e revocação média de 77%. Esses valores representam uma revocação superior a (Kamei *et al.,* 2013) e (Yang *et al.,* 2017), porém uma exatidão inferior em aproximadamente 2,67% do que a apresentada em (Kamei *et al.,* 2013). Vale ressaltar ainda que (Kamei *et al.,* 2013) e (Yang *et al.,* 2017) não relatam a quantidade de nós que sua AD e RD respectivamente apresentam.

Neste trabalho foram desenvolvidas duas ANN que não são adotadas nos trabalhos de comparação, (Kamei *et al.,* 2013) e (Yang *et al.,* 2017). Desenvolvemos ainda três árvores de decisão, onde a primeira possui diferenças no tratamento dos dados de treinamento e as duas últimas possuem o mesmo tratamento dos dados. A diferença está no tratamento dos dados, onde na DT 1 os dados não foram balanceados e nas árvores DT 2 e DT 3 foram balanceados, garantindo que os dados utilizados no treinamento e teste possuam iguais quantidades de classes erro e não erro.

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 trabalho, junto com o diagnóstico utilizando ANN e DT.

6. CONCLUSÃO

Neste trabalho realizou-se um estudo e proposta para um sistema preditivo de defeitos *Just-In-Time* em software utilizando inteligência artificial. O objetivo é oferecer para as equipes de desenvolvedores e de *testers* uma ferramenta preditiva para ajudar na detecção de defeitos no momento do *commit*. O estudo realizado mostrou que ainda existem casos a serem explorados na predição JIT, como por exemplo, novas manipulações nas bases de dados para observações dos comportamentos de preditores com o acréssimo/decréssimo de métricas. Observou-se que para uma melhor predição é necessário realizar um balanceamento dos dados dos casos de defeitos e não defeitos, visto que a grande diferença entre as quantidades de *commits* envolvidos pelas duas classes causa uma tendência no preditor a optar pela classe que possui mais amostras, que no caso é a classe dos não defeitos.

Propuseram-se duas opções de técnicas para predição: árvore de decisão e redes neurais artificiais. Em ambas as técnicas alcançou-se bons resultados de predição em comparação com Kamei *et al.* (2013) e Yang *et al.* (2017). Para o foco em Revocação, observa-se que a árvore de decisão entrega melhores resultados do que a rede neural artificial.

Com relação às bases de dados utilizadas, todas são de projetos de código aberto, sendo eles: Bugzilla, Columba, Mozilla, JDT, Platform e Postgres. As métricas foram extraídas por Kamei *et al.* (2013), e são utilizadas por Kamei *et al.* (2013) e Yang *et al.* (2017).

Inicialmente treinaram-se modelos de rede neural artifical com duas camadas ocultas e uma camada de saída. Refinou-se o modelo para utilizar apenas uma camada oculta e uma de saída, propondo ainda bons resultados. Treinou-se também uma árvore de decisão mais geral, que obteve excelentes resultados preditivos ao custo de milhares de nós. Refinou-se duas vezes a árvore, gerando modelos menores, com destaque para a última árvore que tinha apenas 382 nós e capacidade preditiva com Revocação de 77%.

Entende-se que ao refinar as técnicas preditivas, aumenta-se o poder de generalização das mesmas, pois no caso de possuir uma grande quantidade de neurônios e camadas na rede neural artificial ou ainda uma grande quantidade de nós em uma árvore de decisão, esses mecanismos tendem a se sobreajustarem aos dados propostos.

Uma questão a ser discutida é a viabilidade da aplicação de previsão de defeitos JIT em projetos de linha de produção. Entende-se com este trabalho que é totalmente viável a

aplicação. Observando que a ferramenta oferecida eleva o poder dos *testers* para um foco em determinadas áreas que foram recentemente modificadas ou criadas em um *commit*, observa-se um ganho na redução de esforço e tempo na fase de testes tanto para a equipe de testes automáticos quanto para a equipe de testes manuais.

A grande questão a ser abordada seria a quantidade necessária de *commits* para treinar o sistema preditivo, visto que será necessário todo um histórico de projeto com grande quantidade de *commits* para produzir um grande banco de dados e assim um bom sistema preditivo. Para projetos novos, talvez seja difícil a implantação, a não ser que se leve em consideração o treinamento realizado anteriormente para outro projeto, com alguma semelhança, que já possua um grande banco de dados e adote o preditor para o projeto novo.

Portanto, conclui-se que os objetivos propostos neste trabalho foram cumpridos: foram desenvolvidos métodos preditivos de defeitos em sofwares baseados em tecnicas de IA, as técnicas foram comparadas com outros métodos da literatura, apresentando comportamentos em média superiores, e os métodos desenvolvidos foram refinados de forma a diminuir a complexidade e aumentar a capacidade de generalização.

6.1 Trabalhos futuros

Em trabalhos futuros, pretendemos aprimorar a predição utilizando novas técnicas de tratamento de dados e também de *machine learning*, buscando melhorar os resultados:

a) Pretende-se testar outras técnicas inteligentes de diagnóstico, como por exemplo, redes RBF e floresta aleatória.

b) 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.

c) Planeja-se fazer um estudo da viabilidade de denvolver um preditor específico para cada caso de software, durante a sua criação e desenvolvimento. Verificando a utilidade e eficácia dos resultados.

REFERÊNCIAS

AL QUTAISH, Rafa E.; ABRAN, Alain. **An analysis of the design and definitions of Halstead's metrics.** In: 15th Int. Workshop on Software Measurement (IWSM'2005). Shaker-Verlag. 2005. p. 337-352.

ARAR, Ömer Faruk; AYAN, Kürşat. **Software defect prediction using cost-sensitive neural network.** Applied Soft Computing, v. 33, p. 263-277, 2015.

BATISTA, Brigida Cristina Fernandes. **Soluções de Equações Diferenciais Usando Redes Neurais de Múltiplas camadas com os métodos da Descida mais íngreme e Levenberg-Marquardt**. 2012. Tese de Doutorado. Dissertação de mestrado, PPGME-ICEN-UFPA.

BISBAL, Jesús et al. **Legacy information systems: Issues and directions.** IEEE software, v. 16, n. 5, p. 103-111, 1999.

BREIMAN, Leo. **Some properties of splitting criteria.** Machine Learning, v. 24, n. 1, p. 41-47, 1996.

BREIMAN, Leo et al. **Classification and regression trees**. Wadsworth Int. Group, v. 37, n. 15, p. 237-251, 1984.

KING'S, Jared. **"CS302: The History of Software"**. *learn.saylor.org. Retrieved 2018-02-17.*

EFRON, Bradley. **Estimating the error rate of a prediction rule: improvement on cross-validation.** Journal of the American statistical association, v. 78, n. 382, p. 316-331, 1983.

FAUSETT, Laurene. **Fundamentals of neural networks: architectures, algorithms, and applications**. Prentice-Hall, Inc., 1994.

FENTON, Norman E.; NEIL, Martin. **A critique of software defect prediction models.** IEEE Transactions on software engineering, v. 25, n. 5, p. 675-689, 1999.

FIGUEIREDO FILHO, Dalson Britto; SILVA JÚNIOR, José Alexandre da. **Desvendando os Mistérios do Coeficiente de Correlação de Pearson (r)**. 2009.

FU, Wei; MENZIES, Tim. **Revisiting unsupervised learning for defect prediction**. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017. p. 72-83.

FUKUSHIMA, Takafumi et al. **An empirical study of just-in-time defect prediction using cross-project models**. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, 2014. p. 172-181.

GAYATHRI, M.; SUDHA, A. **Software defect prediction system using multilayer perceptron neural network with data mining**. International Journal of Recent Technology and Engineering, v. 3, n. 2, p. 54-59, 2014.

Halstead, M. H.. **Elements of Software Science**. In: Elsevier North-Holland. New York,

1977.

HAYKIN, Simon. **Redes neurais: princípios e prática**. Bookman Editora, 2007.

HUANG, Qiao; XIA, Xin; LO, David. **Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction**. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2017. p. 159-170.

INMAN, Henry F. **Karl Pearson and RA Fisher on statistical tests: A 1935 exchange from Nature.** The American Statistician, v. 48, n. 1, p. 2-11, 1994.

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

KAMEI, Yasutaka et al. **The effects of over and under sampling on fault-prone module detection**. In: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007). IEEE, 2007. p. 196-204.

KAMEI, Yasutaka et al. A large-scale empirical study of just-in-time quality assurance. **IEEE Transactions on Software Engineering**, v. 39, n. 6, p. 757-773, 2013.

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

KHOSHGOFTAAR, Taghi M.; YUAN, Xiaojing; ALLEN, Edward B. **Balancing misclassification rates in classification-tree models of software quality**. Empirical Software Engineering, v. 5, n. 4, p. 313-330, 2000.

KOSCIANSKI, André; DOS SANTOS SOARES, Michel. **Qualidade de Software-2ª Edição: Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software**. Novatec Editora, 2007.

Levenberg, K. **A method for the solution of certain non-linear problems in least squares**. Quarterly of applied mathematics, *2*(2), 164-168, 1944.

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

Lima, I., Pinheiro, C. A., & Santos, F. A. O. **Inteligência artificial** (Vol. 1). Elsevier Brasil. 2016.

MARQUARDT, Donald W. **An algorithm for least-squares estimation of nonlinear parameters**. Journal of the society for Industrial and Applied Mathematics, v. 11, n. 2, p. 431-441, 1963.

MCCABE, Thomas J. **A complexity measure**. IEEE Transactions on software Engineering, n. 4, p. 308-320, 1976.

MCCABE, Thomas J.; BUTLER, Charles W. **Design complexity measurement and**

testing. Communications of the ACM, v. 32, n. 12, p. 1415-1425, 1989.

MCCULLOCH, Warren S.; PITTS, Walter. **A logical calculus of the ideas immanent in nervous activity**. The bulletin of mathematical biophysics, v. 5, n. 4, p. 115-133, 1943.

MINSKY, Marvin; PAPERT, Seymour. **An introduction to computational geometry**. Cambridge tiass., HIT, 1969.

OKUTAN, Ahmet; YILDIZ, Olcay Taner. **Software defect prediction using Bayesian networks**. Empirical Software Engineering, v. 19, n. 1, p. 154-181, 2014.

PEARSON, Karl. On the dissection of asymmetrical frequency curves. **Phil. Trans. Roy. Soc**, v. 185, n. pt 1, p. 71-110, 1894.

PRESS, W. H.; TEUKOLSKY, S. A.; VETTERLING, W. T.; FLANNERY, B. P, **Numerical Recipes in C: The Art of Scientific Computing.** [S.l.]: Cambridge University Press. 1992.

Quinlan, J. R. ***C4.5:*** **Programs For Machine Learning**. In: Elsevier. 2014.

ROKACH, Lior; MAIMON, Oded Z. **Data mining with decision trees: theory and applications**. World scientific, 2008.

ROSENBLATT, Frank. **The perceptron: a probabilistic model for information storage and organization in the brain**. Psychological review, v. 65, n. 6, p. 386, 1958.

RUMELHART, David E.; HINTON, Geoffrey E.; WILLIAMS, Ronald J. **Learning representations by back-propagating errors**. nature, v. 323, n. 6088, p. 533-536, 1986.

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.

SEACORD, Robert C.; PLAKOSH, Daniel; LEWIS, Grace A. **Modernizing legacy systems: software technologies, engineering processes, and business practices**. Addison-Wesley Professional, 2003.

SILVA, Leandro Nunes de Castro et al. **Análise e síntese de estratégias de aprendizado para redes neurais artificiais**. 1998.

THEODORIDIS, S.; KOUTROUMBAS, K. **Pattern Recognition**. 3. ed. San Diego: Academic Press. 2009.

Tree Algorithms: ID3, C4.5, C5.0 and CART, 2019. Disponível em: <https://scikit-learn.org/stable/modules/tree.html#tree-algorithms-id3-c4-5-c5-0-and-cart>. Acesso em: 15 out. 2019.

TRIOLA, Mário F. **Introdução à Estatística. 7a. Ed. Rio de Janeiro: LTC,** 1999.

VIANA, Marina Siqueira. **Comparaçao experimental entre algoritmos de aprendizado de**

máquina e fatores de exposiçao ao risco em testes de software. 2015.

WERBOS, P. **Beyond Regression:" New Tools for Prediction and Analysis in the Behavioral Sciences**. Ph. D. dissertation, Harvard University. 1974.

YANG, Yibiao et al. **Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models**. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2016. p. 157-168.

YANG, Xinli et al. **Deep learning for just-in-time defect prediction**. In: 2015 IEEE International Conference on Software Quality, Reliability and Security. IEEE, 2015. p. 17-26.

YANG, Xinli et al. **TLEL: A two-layer ensemble learning approach for just-in-time defect prediction**. Information and Software Technology, v. 87, p. 206-220, 2017.

YU, Hao; WILAMOWSKI, Bogdan M. **Levenberg-marquardt training**. Industrial electronics handbook, v. 5, n. 12, p. 1, 2011.