Universidade Federal Rural de Pernambuco
Departamento de Estatística e Informática

Comparação experimental entre algoritmos de aprendizado de máquina e fatores de exposição ao risco em testes de software

Marina Siqueira Viana

Recife

Dezembro de 2015

Marina Siqueira Viana

Comparação experimental entre algoritmos de aprendizado de máquina e fatores de exposição ao risco em testes de software

Orientador: Rodrigo Gabriel Ferreira Soares

Coorientador: Lenildo José de Morais

Monografia apresentada ao Curso Bacharelado em Sistemas de Informação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Recife

Dezembro de 2015

A Deus,
ao meu querido e amado esposo
Bruno José da Silva,
à toda minha família,
à equipe da Ustore,
aos meus orientadores
e aos meus amigos.

Agradecimentos

Agradeço primeiramente a Deus.

Agradeço ao meu querido e amado esposo Bruno José da Silva, pela paciência e compreensão e por ficar madrugadas sem dormir me dando forças. Ao meu pai Carlos Henrique Viana de Andrade e a minha mãe Ana Maria Siqueira pelo apoio e conselhos. E também a minha sogra Maria Rosa José da Silva, pelo carinho que a mesma tem por mim e em memória ao meu sogro Severino Celestino da Silva. E ainda a família Sobreira, pelo exemplo de família que me passaram ao decorrer da minha vida.

Agradeço ao meu orientador Rodrigo Soares que me conduziu na conclusão desta etapa importante da minha vida. E ao meu coorientador Lenildo Morais pela ajuda e divisão de conhecimentos.

Agradeço a todos meus amigos e em particular Jéssica Nunes, pela amizade e experiência profissional que a mesma vem me passando.

Agradeço a Universidade Federal Rural de Pernambuco.

Resumo

A Prática de testes é fundamental no ciclo de desenvolvimento de sistemas computacionais, para garantir uma qualidade mínima ao produto final. No entanto, a atividade de teste é muito custosa e muitas vezes demanda muito tempo, para resolver esse problema é necessário priorizar os teste. A abordagem de testes baseados em riscos prioriza áreas do software que possuem uma probabilidade maior de ter defeito. Outra abordagem que busca encontrar defeito em áreas de um software é predição de defeitos em softwares

Este trabalho tem por objetivo analisar o desempenho dos algoritmos de aprendizado de máquina (AM) dos paradigmas: Máquinas de vetores suporte(SVM), Redes neurais artificiais(RNA) e Comitês de classificadores(EBS). Aplicados no reconhecimento de defeitos em softwares. Comparamos os desempenhos dos algoritmos de AM com o desempenho da probabilidade do risco em funcionalidades do software P(f), método aplicado em testes de software baseados em riscos e em métricas. O método do teste de hipótese teste t de Student foi utilizado para comparar os algoritmos de AM e a probabilidade do risco.

Mostramos que os algoritmos de aprendizado de máquina, Multilayer Perceptron (MLP) do paradigma RNA e o Bagging do paradigma EBS obtiveram desempenho similares ao da probabilidade de riscos P(f) quando analisado o *Recall*. E obtiveram desempenho superior ou de P(f) quando comparado à precisão dos mesmos. Analisamos também o MAP (Mean average precision) de cada algoritmos e da probabilidade do risco P(f), que é uma medida de desempenho para avaliar o quão bom foi o resultado de uma consulta realizada no nosso caso era para obter defeitos em software. Os algoritmos de AM obtiveram um melhor desempenho do MAP e concluímos que isso foi devido à probabilidade P(f) obter um fraco desempenho em sua precisão.

Palavras-chave: aprendizado de máquina, testes de software baseados em riscos, probabilidade do risco, defeito, software.

Abstract

Testing is critical in the development cycle of computer systems to ensure a minimum quality of the final product. However, the testing activity is very costly and often takes a lot of time to solve this problem it is necessary to prioritize the test. The risk-based approach tests the software prioritizes areas that are more likely to have defect. Another approach which seeks to find fault in areas of software is predicting defects in software.

The objective of this study was analyze the performance of algorithms for Machine Learning of paradigms: Support Vector Machine, artificial neural networks and Ensemble based systems. Applied in the recognition of defects in software. We compared the performances of the AM algorithms with the performance of the probability of risk in software features P (f), the method used in software testing based on risk and metrics. The hypothesis testing method Student t-test, we were used to compare the Machine Learning algorithms and risk probability.

We have shown that machine learning algorithms, Multilayer Perceptron (MLP) paradigm RNA and the EBS paradigm Bagging obtained similar performance to the likely risks of P(f) when analyzing the *Recall*. And they achieved superior performance or P(f) when compared to the accuracy thereof. We also analyze the MAP (Mean average precision) of each algorithms and risk probability P(f), which is a performance measure to assess how good was the result of a consultation in our case was for defects in software. The AM algorithms obtained a better performance of MAP and concluded that this was due to the probability P(f) obtaining a weak performance in its accuracy.

Palavras-chave: machine learning, risk based testing, probability of risk, default, software.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Testes de software é uma atividade vital no processo de Qualidade de Software, e é uma das atividade mais complexas e dispendiosa realizada durante o desenvolvimento e manutenção de um sistema [Burnstein, 2000], Vários autores de livros, guias e padrões na área de engenharia de software como [Pressman, 2009], [SWEBOK, 2004] e [IEEE, 1998] incluem o teste de software no processo de desenvolvimento do mesmo. A industria também tem adotado o processo de teste para garantir ou melhorar a qualidade de seus produtos em empresas como google e nokia, por exemplo, que incluíram o processo de teste de software no desenvolvimento de um produto (software) e acabaram por servir de exemplo e incentivo para outras empresas do mesmo setor a adotar a prática de testes em software, no processo de desenvolvimento dos seus produtos.

Testes rigorosos no software e em suas documentações podem reduzir os riscos de ocorrência de problemas no ambiente de produção, e contribui para a qualidade dos sistemas de softwares, se os defeitos encontrados forem corrigidos antes de liberados para [BSTQB, 2012]. Em muitos casos, executar todos os casos de testes não é possível dado ao tempo disponível para a atividade, deixar de executar todos os casos de testes é assumir um risco de falhas, por isso deve-se adotar técnicas nas escolhas de casos de testes que serão executados para diminuir o risco de falha e que satisfaça a qualidade do produto. Segundo [BSTQB, 2012], para decidir o quanto de teste é suficiente, deve-se levar em consideração o nível do risco: o risco técnico, do negócio e do projeto. Além das restrições do projeto como tempo e orçamento.

1.1 Apresentação

Este trabalho propôs a realização de um experimento para comparar a eficiência de dois modelos. Um na área de predição de defeitos em software utilizando algoritmos de AM e métricas de código para prever defeitos em software. E o outro na área de testes baseados em riscos, que tem por objetivo priorizar casos de testes que possuem uma probabilidade maior de revelarem defeitos existentes em um software, para o cálculo desta probabilidade essa abordagem utiliza métricas de códigos.

1.2 Motivações

Testes de software custa caro e consomem recursos que muitas vezes as empresas que o desenvolve não possui como por exemplo um tempo necessário para desenvolver o sistema. Porém, são necessários para garantia de uma qualidade mínima para o software poder ser colocado em produção. Por outro, lado testes de software mais criteriosos tendem a ser demorados. Estudos foram realizados para atacar estes problemas, como testes baseados em riscos que tem por objetivo tanto diminuir o escopo dos testes aumentando sua velocidade, quanto ser mais assertivos na escolhas de casos de testes que irão retornar falhas. Outros estudos se empenham a descobrir áreas dos sistemas propensas a falhas, utilizando aprendizado de máquina. Então este trabalho propõe a análise desse dois métodos que podem tornassem complementares.

1.3 Objetivos

Este trabalho tem como principais objetivos:

Realizar um experimento com bases de dados da engenharia de software utilizando os algoritmos SVMreg, MLP, *Bagging* e o método de teste baseados em riscos propostos por [Amland, 1995], para identificar a probabilidade de um erro.

- Construção de um *script* para calcular a probabilidade do erro proposto em [Amland, 1995].

- Compara estatisticamente o desempenho dos algorítimos de AM e a probabilidade de um erro no software através do método de teste de hipótese T-Test.
- Analisar os resultados obtidos com as medidas MAP, *Precision* e *Recall*

1.4 Contribuições

As principais contribuições deste trabalho são:

- Execução de um experimento para avaliar o desempenho da probabilidade de um defeito proposto por alguns estudos da engenharia de teste.
- Uma análise de desempenho da detecção de defeito utilizada na engenharia de teste, com a predição de defeito com algorítimos de AM.

1.5 Organização do trabalho

O capítulo 1 é a introdução deste trabalho.

No capítulo 2 apresentamos alguns estudos relacionados com este trabalho: Que usaram a abordagem de predição de defeitos em software, utilizando aprendizado de máquina e que usaram a abordagem de teste baseados em riscos.

No capítulo 3 apresentamos o problema e a definição de riscos e aqueles de interesse deste trabalho.

No capítulo 4 fizemos uma revisão da literatura sobre testes baseados em riscos e em métricas.

No capítulo 5 foi abordado os algorítimos de aprendizado de máquina utilizado neste trabalho.

No capítulo 6, descrevemos desde as bases de dados, até como foi realizado o experimento e seus resultados. Nele descrevemos as características das bases de dados, como foi realizado o pré-processamento das bases e os ajustes de parâmetros para cada algoritmo de AM. Em seguida mostramos como foi realizado o experimento, a análise e resultado do mesmo.

O capítulo 7 é destinado as conclusões do trabalho.

Capítulo 2

Trabalhos Relacionados

Foram realizados vários estudos para identificar possíveis defeitos ou riscos em funcionalidades de um software. Alguns destes como em Amland [1995] e Bach [1999] desenvolveram métodos que identifica e analisa os riscos do software para utilizar na fase de teste do sistema e assim focar na execução de casos de testes das funcionalidades que possuem um maior risco de falha. Neste capítulo serão apresentados alguns destes trabalhos.

Na seção a seguir apresentaremos o primeiro trabalho que utilizou a identificação de riscos do sistema em teste de software, o trabalho proposto por Bach [1999] utilizou um método heurístico. Em seguida ainda na mesma seção, apresentaremos um estudo de caso realizado por Amland [1995] que utilizou métricas de código para priorizar as funcionalidades do sistema que possuíam um risco maior de apresentar um defeito. Na seção seguinte apresentaremos estudos que utilizaram algorítimos de aprendizado e máquina e métricas de software para prever defeitos no sistema.

2.1 Testes baseado em riscos

Teste baseado em risco ou Risk-based Testing (RBT) consiste em um conjunto de atividades que identifica e prioriza os riscos das funcionalidades do sistema. Essa abordagem é útil para que os casos de testes possam ser elaborados com o objetivo de mitigar os riscos identificados e também para priorizar a execução dos casos de teste.

James Bach escreveu o primeiro trabalho de teste baseado em riscos Bach [1999]. O autor

apresentou duas abordagens que utilizam uma análise heurística Bach [1999]: I) Inside-out (de dentro para fora) onde o produto é estudado e é questionado repetidamente, o que pode dar errado nesse produto. II) Inside-in (de fora para dentro) onde o autor sugere 3 listas de riscos: categorias de critérios de qualidade, listas de riscos genéricos e catálogos de riscos, para cada item destas 3 listas é verificado se o item em questão se aplica ou não a cada um dos componentes do sistema. A dificuldade da abordagem utilizada por Bach está na necessidade de conhecimento prévio do produto para que a análise do risco possa condizer com a realidade, vai depender da experiência do engenheiro de teste, analista, gerente ou de quem estiver identificando e analisando os riscos.

Amland [1995] realizou um estudo de caso na fase de teste de um sistema para uma instituição financeira, com requisitos de testes complexos. Tal estudo foi motivado pela falta de recursos suficientes para testar todas as probabilidades de execução do produto. Os resultados mostraram que o método utilizado pelo autor consumiu menor esforço do que a estimativa da abordagem tradicional de testes. O trabalho se concentrou na análise do risco, fase onde é calculada a probabilidade de uma falha ocorrer em uma função do produto. O cálculo da probabilidade é dada pela média ponderada de 4 métricas, que possuem pesos com valores de 1 a 5 e assumem valores entre 1 e 3, sendo 1 bom, isto é, tem uma baixa probabilidade de ocorrer. O principal desafio no cálculo da probabilidade é a atribuição dos valores de 1 a 3, pois eles precisam ser coerentes com as características reais do produto e para isso dependem do conhecimento de quem avalia a função e atribui o valor.

2.2 Método de classificação para predição de defeitos em softwares

Na literatura, há vários estudos que usam algoritmos de classificação para previsão de defeitos em funções de software, alguns desses estudos foram elaborados com a ajuda do repositório PROMISE Software Engineering Repository, que mantem um conjunto de bases de dados incluindo o repositório da NASA Metrics Data Program.

Um desses estudos Wahono and Herman [2014] utilizou um experimento com 9 bases da NASA MDP e 10 algoritmos de classificação: Classificação estatística (LogisticRegression (LR), Linear Discriminant Analysis (LDA) e Naïve Bayes (NB)), Nearest Neighbors (k-

nearestneighbor (k-NN) e K*), Neural Network (BackPropagation (BP)), Support Vector Machine (SVM), e Decision Tree (C4.5, Classification and Regression Tree(CART), e Random Forest (RF)). Os resultados, do experimento realizado no estudo [1], mostraram que o NB e LR aparentemente tiveram o melhor desempenho e os modelos baseados em árvore de decisão obteve um desempenho menor que os demais classificadores devido ao desequilíbrio das classes. O SVM também aparentou um desempenho menor, embora SVM tenha excelente capacidade de generalização da situação dos pequenos dados de exemplo, como o conjunto de dados da NASA MDP.

Um outro estudo Gray et al. [2014] realizou um experimento com 11 bases de dados também da NASA MDP, utilizando apenas um classificador o Support Vector Machine (SVM). Esse experimento mostrou um rendimento com precisão de 70% do SVM. O estudo analisou ainda que embora outros estudos (Elish and Elish [2008], Lessmann et al. [2008], Li and Reformat [2007]) também tenham utilizados os dados da NASA MDP e o classificador SVM, alguns deles fizeram uma citação rápida do pré-processamento destes dados e outros como o caso do estudo Elish and Elish [2008], cujo a média mínima previu corretamente para o total de módulos defeituosos 84%, não fez nenhuma citação sobre o tratamento dos dados. O estudo Gray et al. [2014] considerou o pré-processamento dos dados da NASA MDP uma parte muito importante, uma vez que existem instancias redundantes e isso pode elevar a taxa de acertos do algoritmo.

2.3 Conclusão

Apesar de existirem vários trabalhos na literatura que abordam riscos em teste de software, a maioria realiza esta análise de risco com base na experiência das pessoas envolvidas no projeto. Como em Amland [1995] que calculou a probabilidade de falhar uma função a partir de métricas do sistema, cujo os valores para cada métrica era informado pelos integrantes do projeto de acordo com sua experiência. Em outro estudo Bach [1999], foi utilizada uma abordagem heurística para resolver o problema.

Como algorítimos de inteligência artificial são indicados para resolver problemas que simulam a experiência humana, nosso trabalho propõe uma comparação experimental entre o método utilizado por Amland [1995] na etapa de análise dos riscos e algorítimo de inteligência artificial.

Capítulo 3

Riscos em testes de software

Na etapa do teste de software, um dos principais problemas são erros não detectados pela falta de testes dos seus requisitos. Tais erros podem causar algum dano para o cliente, usuário ou empresa que desenvolveu o sistema. O objetivo do teste é encontrar *bugs* no sistema antes dele ser colocado em produção. Priorizar os riscos do sistema é importante para encontrar os *bugs*, sem ter que testar todos os cenários de testes possíveis.

A atividade de teste tem um custo muito alto. Existem estimativas que ele pode chegar a 45% do valor inicial do projeto Kaner et al. [2004]. Além disso, em muitas situações, executar todos os casos de testes não é possível devido ao tempo disponível para a atividade. Deixar de executar todos os casos de testes é assumir um risco de falhas.

3.1 Testes de software baseados em riscos

O teste baseado em riscos consiste em atividades para identificação, análise e mitigação de fatores de riscos associados aos requisitos do produto de software. Esse tipo de teste prioriza esforços e aloca recursos para os componentes de software que necessitam ser testados mais cuidadosamente. Com testes baseados em riscos, a atividade de testes é melhorada a partir da gerência dos riscos técnicos de software. Os riscos podem ser mitigados através da priorização dos casos de testes Morais [2012].

3.1.1 Identificar

O objetivo desta fase é identificar os riscos técnicos. Esta etapa produz um documento com uma lista de riscos técnicos do sistema Souza [2008]. O passo inicial, geralmente preferido pelas equipes de projeto, é a revisão da documentação: documento de requisitos, plano de projeto, arquivos de projeto anteriores entre outros. Em seguida com base nas informações levantadas um mediador pode elaborar um questionário para solicitar ideias à equipe do projeto, sobre os riscos mais importantes do software. Outra técnica adotada são as reuniões de *brainstorm*. Os passos descritos são ilustrados na figura a seguir.

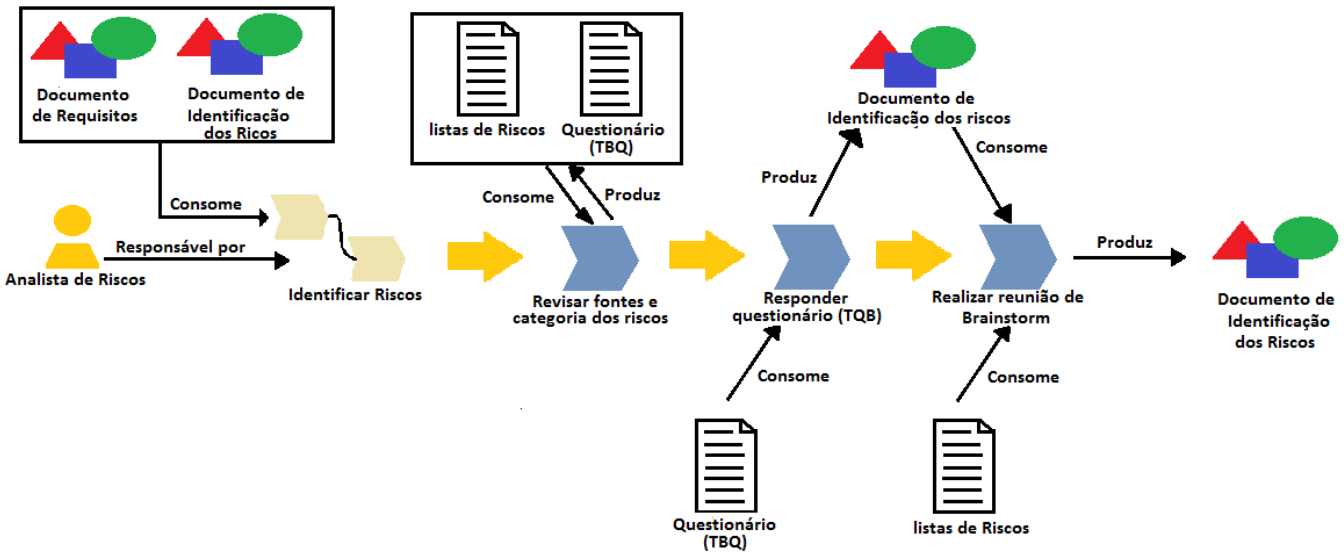

Figura 3.1: Etapa de identificação dos Riscos.

3.1.2 Analisar

Com os riscos identificados, é feito o cálculo da exposição Souza [2008] para cada risco. O objetivo deste cálculo é obter uma lista priorizada dos riscos, conforme descrito na figura (3.2).

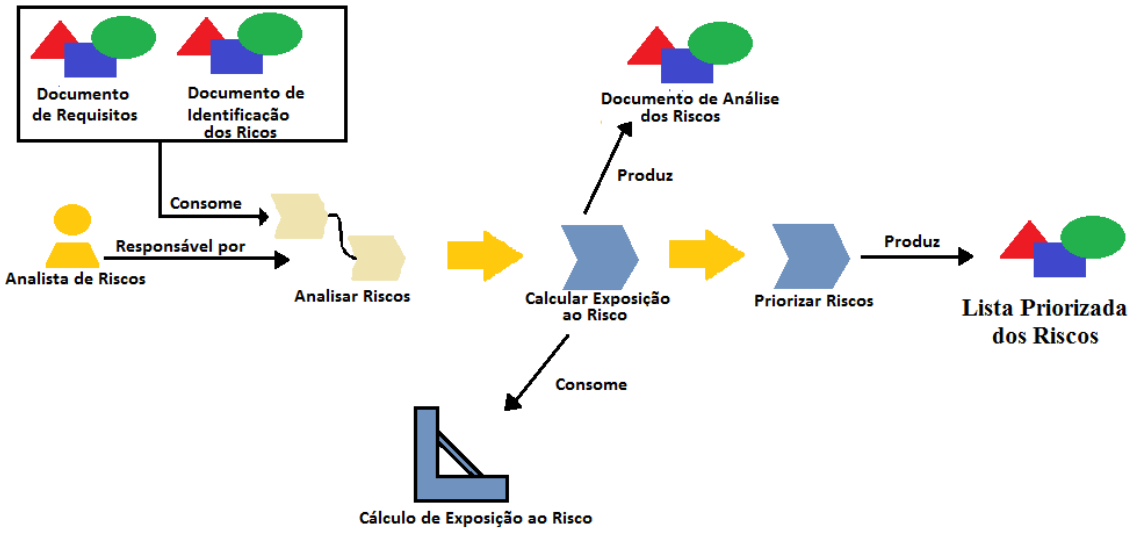

Figura 3.2: Etapa de Análise dos Riscos.

A etapa de analisar pode ser difícil, por ser uma análise subjetiva e por depender da experiência do profissional que estiver realizando a análise. O modelo mais tradicional para analisar um risco é a matriz de impacto *versus* probabilidade.

Para analisar o risco é feito o cálculo da exposição ao mesmo. Onde são atribuídos valores da probabilidade de uma falha ocorrer para cada risco identificado. E atribuído valores para o impacto caso o risco se materialize em uma falha. As atribuições dos valores podem ser feitas de acordo com a matriz demonstrada na figura anterior. Nessa matriz os valores estão no intervalo de 1 a 3. Onde o valor 1 indica baixa probabilidade de ocorrência de uma falha ou baixo impacto. O valor 3 alta probabilidade de ocorrer uma falha ou alto impacto. Outras abordagens de cálculo da exposição aos riscos são utilizadas. Além de variações do modelo descrito na figura (3.3).

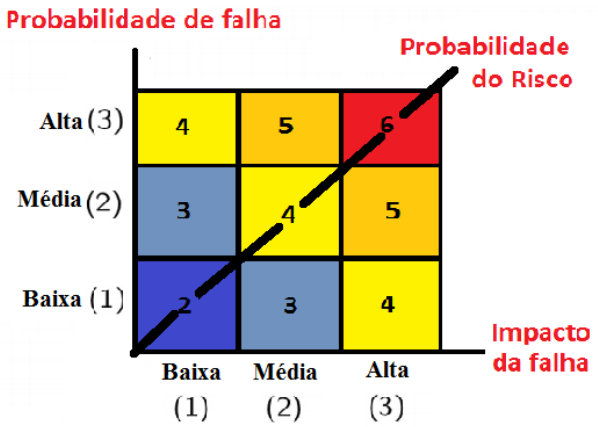

Figura 3.3: Matriz de Impacto x Probabilidade.

3.2 Classificação dos riscos

Risco é um perigo e pode ser considerado também como a probabilidade de perda ou dano. Para um projeto de desenvolvimento de software é a possibilidade de sofrer uma perda, um impacto na qualidade do produto final, um atraso do cronograma, um aumento nos custos ou falha do projeto. Um risco nunca é uma certeza, pois se houver a certeza 100% da ocorrência de um dano, perda ou falha isso deixa de ser um risco e passa a ser um problema.

Quando falamos de risco em projetos de desenvolvimento de software o seu significado fica claro para a maioria das pessoas envolvidas durante o ciclo de desenvolvimento. No ciclo de desenvolvimento, o risco é tudo aquilo que pode atrapalhar o sucesso de um projeto, ou seja, qualquer evento que impeça a entrega do projeto com uma qualidade aceitável. Em testes de software riscos é a possibilidade de uma falha ocorrer durante a execução do sistema. Segundo Rios [2008], os riscos podem ser divididos em: i) Riscos de projeto ii) Riscos técnicos iii) Riscos para o negócio do cliente.

Riscos do projeto são aqueles que estão ligados diretamente ao plano do projeto, por exemplo: problemas com orçamento, cronograma, pessoal, recursos, cliente e requisitos. Riscos técnicos estão relacionados à qualidade do software que será desenvolvido como: defeitos no software, problemas com implementação, interface, manutenção do sistema (ambiguidades na especificação, incertezas tecnológicas, obsolescência tecnológica, tecnologia de ponta). Os riscos para o negócio do cliente afetam as organizações envolvidas, seja por mudanças no orçamento, prazo de desenvolvimento ou problemas de mercado. Exemplo: desenvolver um

produto do qual ninguém precisa (risco de mercado) ou um produto que não esteja alinhado com a estratégia de venda da empresa (risco estratégico) entre outros.

Para este trabalho só interessa os riscos técnicos. Pois estes identificam as áreas de potencial falha do produto (software). Ao identificar os riscos que podem se materializar em falhas durante a execução do sistema, pode-se elaborar casos de testes focados nos riscos. Um outro ganho é que testes baseados em riscos diminui o escopo dos testes, aumentando a velocidade da execução dos casos de testes e consequentemente diminui o custo do mesmo Morais [2012].

3.3 Conclusão

Neste capítulo foi descrito duas etapas de testes baseados em risco. A primeira para identificar o risco e a segunda para análise do mesmo. Descrevemos como pode ser realizado a identificação dos riscos e que a análise dos riscos pode ser feita tanto com análise da documentação e pelo conhecimento da equipe envolvida no projeto quanto pode ser calculada, cálculo esse que será descrito na próxima seção.

Capítulo 4

Testes de software baseados em riscos e em métricas

Amland publicou um trabalho intitulado *Risk Based Testing and Metrics: Risk analysis fundamentals and metrics for software testing* Amland [1995]. Contendo um estudo de caso na fase de teste de um sistema para uma instituição financeira. O objetivo do trabalho era diminuir o escopo de teste, ou seja, diminuir a quantidade de testes a ser executado na fase de teste do produto. Porém era necessário garantir que isso não afetaria a qualidade do sistema.

4.1 Estratégia utilizada

Amland utilizou a abordagem de testes baseados em riscos. No trabalho Amland [1995], foi concordado com o cliente 6 pontos para cobrir todas as atividades de identificação dos riscos por meio de relatórios, listados a seguir:

1. O fornecedor do sistema iria testar todas as funcionalidades da aplicação para "um nível mínimo"(além de todas as interfaces, e todos os testes não-funcionais). Isso não foi documentado antes da execução dos testes, mas o registro de detalhes sobre todos os testes (isto é, entrada, saída esperada e saída real), foram documentados após a execução do teste, isso comprova um "nível mínimo de teste"(Estratégia de Risco).

2. Todos os casos de teste ("o que testar") foram documentados antes de começar a testar

e estava disponível para o cliente avaliar (Estratégia de Risco).

3. Só testadores altamente qualificados, ou seja, analistas de sistemas com experiência na área da aplicação, foram utilizados para testar, e os testadores foram responsáveis pelo planejamento de todos os casos de testes, incluindo o fornecimento de dados para o mesmo e documentar os que foram executados. (Ferramentas estavam disponíveis para o testador documentar os testes) (Estratégia de Risco).

4. O fornecedor do sistema fez uma análise dos riscos em conjunto com o cliente, para identificar as áreas de maior risco, tanto para o cliente quanto para o fornecedor (Identificação de Riscos e Avaliação de Riscos).

5. Com base na análise dos riscos, o fornecedor irá focar "teste extra"nas áreas de maior risco (Mitigação de Risco).

6. Os testes extras foram planejados e executados por um especialista na área da aplicação, que não estava envolvidos no "nível mínimo de teste"(Mitigação de Riscos e Reportar Riscos).

No ponto 4 desta lista, verifica-se que a identificação das áreas com maior risco foi feita a partir de uma análise em conjunto do vendedor e cliente do sistema.

O estudo de caso realizado por Amland Amland [1995] se concentrou na etapa de Análise dos riscos identificados. O sistema utilizado no estudo de caso possuía dois módulos, um on-line e outro que processa em lote (batch). O módulo on-line teve uma técnica de análise dos riscos diferente da realizada no módulo batch. Neste trabalho será descrito apenas o modelo de análise dos riscos utilizado no módulo de processamento em lote (batch), descrito na seção a seguir.

4.2 Análise de riscos

Na análise dos riscos, foi calculada a exposição do risco com base em: i) Probabilidade de um erro, ii) Custo (consequência) de um erro na função correspondente, tanto para o vendedor quanto para cliente (em produção).

A probabilidade do erro está relacionada com a qualidade da função (área) a ser testada, ou seja, a qualidade de um programa. Isto foi usado como uma indicação da probabilidade de

uma falha P(f). O pressuposto é que uma função que sofre de má concepção (programador inexperiente, funcionalidade complexa etc.) está mais exposta a falhas do que as funções com base em uma melhor qualidade de *design*, programador mais experiente etc.

A consequência para o cliente é tido como o custo C(c) cujo cliente que comprou o sistema pode ter, caso uma falha o corra na fase de produção. Alguns exemplos de custos seriam: a probabilidade de uma ameaça legal, perca de mercado, não cumprimento das regulamentações governamentais entre outros.

A consequência para o vendedor é o custo C(v), e refere-se ao custo que a empresa fornecedora do sistema poderia ter caso uma função falhasse. Um exemplo desse custo seria a publicidade negativa e o alto custo de manutenção de software.

A exposição ao risco de uma funcionalidade é calculada pela formula:

$$Re(f) = P(f) \cdot \frac{C(c) + C(v)}{2} \tag{4.1}$$

Onde, P(f) é a probabilidade de falha da funcionalidade analisada, C(v) é o custo para o vendedor e C(c) é o custo para o cliente.

A empresa que desenvolveu o sistema convidou a gestão do projeto, especialistas de produto, designers, programadores, pessoas que trabalham com manutenção de aplicativos e gerenciamento de teste, para uma reunião. Esta equipa teve de concordar com o custo de uma falha e a probabilidade de uma falha para cada função.

O custo de uma falha tanto para o vendedor do software, quanto para o cliente que comprou o sistema, foi atribuído os seguintes valores: 1 , 2, ou 3. Em que 1 representa um custo mínimo no caso de uma falha nesta função e 3 o custo máximo. A probabilidade de uma falha foi indicado por 4 métricas (nova funcionalidade, qualidade, tamanho e complexidade) cada um desses quatro indicadores, assim como no custo, podem assumir valores entre 1 e 3. A seguir descrevemos as 4 métricas:

- Nova funcionalidade - o autor justificar que o projeto foi uma reengenharia de um aplicativo existente e que a alteração da funcionalidade variou de função para função.
- Qualidade do projeto ou (Design Qualidade) - Esta métrica foi medida pela contagem de números de solicitações de alteração ao projeto.

- Tamanho - Assumiu-se que o número de subfunções dentro de uma função afeta o número de defeitos introduzidos pelo programador.
- Complexidade - A habilidade do programador para compreender a função que ele estava programando.

Foram dados pesos para cada indicador descritos acima que variam de 1 a 5, sendo a classificação 5 como o mais importante indicador de uma função com má qualidade. Os pesos utilizado no estudo de caso Amland [1995] foram:

- Nova funcionalidade - 5
- Design Qualidade - 5
- Tamanho - 1
- Complexidade - 3

A seguir descreveremos um exemplo para calcular R(f) descrito na equação (4.1):

	Probabilidade					
	Nova funcionalidade	Design de Qualidade	Tamanho	Complexidade	Média Ponderada	P(f)
Pesos	5	5	1	3		
Fechar conta	2	2	2	3	2,21	0,736

Tabela 4.1: Probabilidade do risco.

	Custo		
	C(v)	C(c)	Média do Custo
Fechar Conta	1	3	2

Tabela 4.2: Média do custo.

Como descrito na equação (4.1), o risco para a funcionalidade de fechar uma conta é dada pelo produto de P(f) e da média do custo neste caso 0,736 multiplicado por 2 cujo resultado é 1,47.

4.3 Conclusão

Neste capítulo, foi apresentado como o método de testes baseados em riscos e em métricas foi estruturado e como calcular o risco R(f) de uma funcionalidade. Para aplicar na análise do risco baseado. E descrevemos ainda as quatro métricas utilizadas para realizar o cálculo da probabilidade de uma funcionalidade conter erro.

Capítulo 5

Aprendizado de Máquina

5.1 Máquinas Vetores Suporte

Máquinas de Vetores Suporte ou *Support Vector Machine (SVM)* é um algoritmo de aprendizado de máquina(AM) supervisionado que pode ser utilizado tanto para classificação, quanto para regressão Soares [2008]. Esta técnica originalmente desenvolvida para classificação binária, busca a construção de um hiperplano como superfície de decisão, de forma que a separação entre os exemplos seja máxima.

SVMs lineares são bons para conjunto de dados linearmente separáveis. No entanto, existem muitos casos em que não é possível dividir o conjunto de treinamento linearmente por um hiperplano. Nesses casos, podemos mapear os dados para um espaço de dimensão mais alta, no qual os dados passam a ser linearmente separáveis.

Rótulos ou classe são os fenômenos que desejamos realizar uma previsão. Para SVMs de classificação estes rótulos possuam valores discretos (1,..,n). Nos casos que estes valores sejam contínuos temos o SVM para regressão.

Alguns exemplo para aplicações do SVM são: categorização de textos, análise de imagens e em Bioinformática Lorena and Carvalho [2007]. Os resultados da aplicação desta técnica de Aprendizado de máquina (AM) tem se mostrado satisfatório e em alguns casos superam os resultados de outros algoritmos de inteligencia artificial (IA) Cristianini and Shawe-Taylor [2000].

5.1.1 Margens Rígidas

O modelo mais simples, que é a base para SVMs mais complexo, são os SVMs lineares com margens rígidas que separam os dados com um hiperplano ótimo, ou seja, de margens máximas. Estes SVMs separam dados linearmente separáveis.

Para exemplificar os dados linearmente separáveis, assuma que S é um conjunto de treinamento que desejamos realizar uma predição. S possui n dados $x_i \in$ X. Cada x_i possuem seus respectivos rótulos $y_i \in$ Y onde Y = { +1 , -1 }. O conjunto de dados é dito linearmente separáveis se existir um hiperplano que divida os dados em +1 e -1. A equação do hiperplano é apresentada a seguir Soares [2008].

$$f(x) = w \cdot x + b = 0 \tag{5.1}$$

Onde w ∈ X e é a normal(perpendicular) ao hiperplano, x.w é o produto escalar desses dois vetores. A equação (5.1) divide o espaço em duas regiões: w · x + b > 0 e w · x + b 4 < 0.

5.1.2 Margens Suaves

Como a maioria dos dados reais não são linearmente separáveis, devido a diversos fatores como por exemplo ruídos, o SVM com margem suave introduz uma variável de folga ϵ. Esta variável permite que alguns dados possam violar as restrições do SVM com margem rígida.

$$y_i(w. \cdot x_i + b) > 1 - \xi_i, \xi_i > 0, \forall i = 1, ..., n \tag{5.2}$$

5.1.3 Máquinas de Vetores Suporte não Lineares

SVM com margens suaves toleram apenas alguns ruídos. Para dados que não podem ser separados por um hiperplano com margem rígida ou suaves, temos o SVM não linear. A técnica adotada aqui é aumentar a dimensionalidade do espaço amostral dos dados, isto é, caso os dados estejam em R^d passamos os dados para o espaço R^x, em que x > d, onde os dados possam ser separados por um hiperplano. Para isso é necessário uma função kernel Soares [2008].

5.1.4 Máquinas de Vetores Suporte para Regressão

SVMs também podem ser aplicado para regressão. O conceito de margem máxima é aplicado quando se quer classificar os dados Soares [2008]. No caso da regressão SVR *(Support Vector Regression)* ao invés de lida com saídas Y = +1, -1 o SVR está preocupado com estimar funções reais. Para gerar um algoritmo SVR é reformulada a teoria do SVM com margens suaves. Da mesma forma que a classificação o SVR gera um hiperplano separador e podem ser aplicados a problemas não lineares com o auxilio de uma função kernel. A ideia do SVR é encontra uma função que construa a melhor interpolação das instâncias Soares [2008].

Como descrito em Soares [2008] a dificuldade para o SVM é a escolha da função kernel e seus parâmetros e do valor da constante C (parâmetro responsável pela capacidade de generalização da SVM e a largura das marges). Essas confinações são importantes pois elas tem impacto no desempenho obtido pelo algoritmo. Por outro lado as SVMs são robustas as instâncias com características atípicas.

5.2 Redes Neurais Artificiais

5.2.1 Conceitos de Redes Neurais Artificiais

A estrutura de um neurônio artificial é similar a de um biológico. Desenvolvido por McCulloch e Pitts Braga et al. [2000], ele possui terminais com n entradas, realizando uma alusão ao neurônio biológico essa entradas seriam os dendritos. Neurônios artificiais também possui terminal de saída para que o neurônio biológico seja o axônio. São associados pesos w aos terminais de entradas do neurônio artificial. Estes pesos são ajustados conforme o aprendizado da rede. Em um neurônio biológico, ocorre uma saída (pelo axônio) sempre que a soma do impulso de um neurônio atingiu seu limite. Em neurônio artificial a saída ocorro através da aplicação de uma função de ativação.

Um dos principais aspectos das redes neurais artificiais é a utilização de uma função de ativação. Ela ativa ou não a saída, dependendo do valor ponderado nas entradas do neurônio. Existe várias funções de ativação: função de ativação linear, função degrau, função rampa, função sigmóide entre outras Braga et al. [2000]. Em neurônios MCP (modelo de neurônio artificial de McCulloch e Pitts), a ativação de um neurônio é feita pela função de ativação.

O modelo MCP resolve apenas problemas que são linearmente separáveis e sua função de ativação descrita na equação 5.3 tem como saída apenas os valore 0 e 1, além disso seus pesos w são fixos.

$$\sum_{i=0}^{n} x_i w_i \geqslant \theta \tag{5.3}$$

Onde n é o número de entradas do neurônio, x_i é a entrada e w_i é o peso associado a entrada. θ é o limite *threshold* do neurônio artificial Braga et al. [2000].

As redes neurais artificiais (RNA), além da função de ativação do neurônio, possuem a arquitetura (topologia da rede). Essa arquitetura pode ser com múltiplas camadas, onde existem mais de um neurônio entre a entrada e a saída da rede , ou com uma única camada onde existe um único nó entre a entrada e saída Braga et al. [2000].

A saída e entrada dos valores em um RNA (conexões), podem ser de duas formas. *Feed-forward* ou acíclica: onde a saída de um neurônio não pode ser utilizado em nenhum outro neurônio que esteja em uma camada anterior da arquitetura. E a conexão *Feedback* ou cíclica: em que a saída de um neurônio pode ser utilizado como entrada para outro neurônio de uma camada anterior.

A utilização do RNA se inicia com a aprendizado do algoritmo, nela a rede vai ajustando seus parâmetros que são seus pesos. A aprendizado pode ser supervisionada ou não-supervisionada. O modelo mais comum é o supervisionado, nele existe a figura de um supervisor (professor) que informa a entrada e a saída desejada para a rede. No modelo não-supervisionado a figura desse supervisor não existe o aprendizado da rede, assim o aprendizado não-supervisionado é feito com padrões de entrada, esse método é bom apenas quando existe redundância nos dados.

5.2.2 Redes MLP (MultiLayer Perceptron)

Para resolver problemas não separáveis linearmente foram criadas as redes MLP, pois esse tipo de rede possui pelo menos duas camadas permitindo a aproximação de qualquer função contínua. As redes MLP deriva de um modelo proposto por Frank Rosenblatt em 1958

chamado de *perceptron* Braga et al. [2000], esse modelo resolve apenas problemas linearmente separáveis, ao adicionar camadas escondidas o poder computacional foi aumentado. A função de ativação do MLP mais empregada é a sigmoidal logística.

O algoritmo mais utilizado para o treinamento de redes MLP é o algoritmo *backpropagation* outro algoritmo usado é o *Levenberg-Marquardt*, o primeiro otimiza sua função objetivo através de equações de primeira ordem e o segundo treina com funções de segunda ordem Soares [2008].

O MLP é um algoritmo com paradigma supervisionado que pode realizar tanto classificação, quanto regressão Soares [2008]. Seu treinamento ocorre em duas fases: fase forward e a fase backward Braga et al. [2000]. Na fase forward a entrada é apresentada à primeira camada da RNA, que calcula seus sinais de saída e passa os valores para a camada seguinte. Esta camada calcula seus sinais de saída e passa para a próxima camada, isto vai acontecendo até a camada de saída obter as saídas da RNA, as quais são comparadas com as saídas desejadas. A fase backward percorre o caminho inverso. A partir da camada de saída até a camada de entrada os pesos dos neurônios vão sendo ajustados de forma a diminuir seus erros (os erros dos neurônios das camadas intermediárias são calculados utilizando o erro dos neurônios da camada seguinte ponderado, pelo peso da conexão entre eles). Este processo é repetido até atingir algum critério de parada.

Como descrito em Soares [2008] as redes MLPs são robustas a ruidosos e tem a capacidade de representar funções lineares ou não-lineares, além de possuírem a capacidade de lidar com instâncias de alta dimensão, onde os valores dos atributos podem ser contínuos ou discretos. As principais dificuldades encontradas para se trabalhar com as MLPs é a dificudade da definição dos seus parâmetros, como por exemplo definir o número de neurônios em suas camadas escondidas entre outras configurações. Além disso existe a dificuldade da compreensão dos conceitos aprendidos pela rede, codificados nos valores finais dos pesos da rede.

5.3 Comitê de Classificadores

Ensemble based systems (EBS), também conhecido como comitê de classificadores, é um método da AM que utiliza a saída de diferentes classificadores chamados de classificadores

base para conseguir uma classificação mais exata, com menos erro. Caso ao escolher um único classificador e ele não cometer nenhum erro não é necessário a construção de *Ensemble*. Caso o classificador escolhido cometa erros, pode-se construir um comitê de classificação com classificadores que não cometam o mesmo erro Kuncheva [2004]. desta forma a diversidade na saída dos classificadores bases do *Ensemble* é muito importante. Esta diversidade pode ser realizada de várias formas, como por exemplo utilizando parâmetros diferentes para o mesmo algoritmo, aumentando a quantidade de classificadores bases, variando os dados utilizados na construção do classificador entre outros.

Comitê de classificadores tem mostrado maior desempenho e confiabilidade do que sistemas individuas. Sua dificuldade esta em construir um comitê que possuem classificadores bases com a diversidade necessária. Os principais pontos que deve-se levar em consideração ao combinar os classificadores bases são: i) Identificar como realizar a combinação de cada um, ii) Criar os classificadores membro e iii) Escolher os métodos mais efetivos para o multi-classificador. O comitê tem a habilidade de corrigir erros de seus membros.

5.3.1 Bagging

O método *baggin* Breiman [1999] é bastante utilizado para a construção de comitês, onde os classificadores bases são formados a partir de padrões diferentes. A implementação do *bagging* é simples, ele foi o primeiro algoritmo construído para implementação de EBS. Nele a saída dos classificadores são combinadas por meio de votos e o classificador que obtiver o mair numero de votos para uma determinada instância será a resposta. A diversidade no *bagging* é obtida com o uso de diferentes subconjuntos de dados criados aleatoriamente. E cada subconjunto é utilizado para treinar um classificador do mesmo tipo.

5.3.2 Conclusão

Neste capítulo apresentamos 3 algoritmos de aprendizado de máquina, suas definições e características. Mostramos que o SVM pode ser aplicado tanto para regressão quanto para classificação. Apresentamos o conceito do algoritmo de rede neural MLP. Por fim apresentamos o *Bagging* do paradigma EBS. Os três método apresentados nesta seção serão utilizados para regressão dos dados no experimento deste trabalho.

Capítulo 6

Experimentos

6.1 Bases de dados

As bases de dados utilizadas para os experimentos foram obtidas através do site *Promise Software Engineering Repository* **?**, que possui um conjunto de dados gratuitamente disponibilizados para servirem aos pesquisadores na construção de modelos de previsão de software e a comunidade de engenharia de software em geral.

- CM1 - É uma base que contém métricas de código de um software. Esse software é uma sonda da NASA escrito em "C".
- JM1 - Também escrito em "C", as métricas de código contidas nesta base de dados é de um programa cujo objetivo é realizar predições utilizando simulações.
- KC1 - É uma base de dados que contem métricas de código de um software para gerenciar o armazenamento de dados. Foi escrito em "C++".
- KC2 - É uma base de dados que contém métrica de código de um sistema para processamento de dados; O software de onde foi inferida as métricas contidas em KC2 faz parte do mesmo projeto do software em que foi retirada as métricas da base KC1. Porém, usou pessoas diferentes . Compartilhou algumas bibliotecas de software de terceiros, assim como o KC1, mas não realizou sobreposição no software.
- PC1 - É uma base de dados que contém métricas de código de um software de voo para órbita terrestre por satélite e foi escrito em "C".

6.1.1 Atributos

As bases CM1, JM1, KC1, KC2 e PC1 possuem 22 atributos que são na verdade métricas de código. Por isso a seguir descreveremos cada uma das 22 métricas utilizadas no experimento.

Tipos de Métricas	**Tipos de Métricas**
McCabe	CC,EC,DC,LOC
Medidas derivadas (Halstead)	N,V,D,I,E,B,L,T
Medidas de linhas de código (Halstead)	lOCode, lOComment, lOBlank,,lOCodeAndComment
Medidas de base (Halstead)	uniq Op, uniq Opnd , total Op, total Opnd
Branch count	BC

Tabela 6.1: Resumo dos 22 atributos das instância.

As métricas utilizadas no experimento pelas bases de dados são as métricas de McCabe, Halstead e Branch count. As métricas de McCabe e Halstead foram definidas na década de 70, são medidas de código estático, que podem ser extraídas do código fonte de forma automática, mesmo para software muito grande Nagappan [2005] e são baseadas em módulo. Na linguagem de programação C, módulo é considerado uma função.

Medidas de código estático dificilmente é uma caracterização fiel de uma função. Segundo Fenton [1996] uma mesma funcionalidade construída em linguagem de programação diferente, resulta em diferentes medições estáticas. Um argumento ao pensamento de Fenton é que métrica estática do código não é uma certeza 100% da presença de um defeito, mas a probabilidade de ocorrer uma falha em uma função do sistema.

As métricas de McCabe utilizadas neste trabalho são: complexidade ciclomática, complexidade essencial, complexidade de design e linhas de código. Elas serão Descritas a seguir:

Complexidade ciclomática ou v(G) - calcula o número de caminho linearmente independente. Um conjunto de caminhos é dito linearmente independente se nenhum caminho no conjunto é uma combinação linear de quaisquer outros caminhos no conjunto através do fluxograma de um programa. Um fluxograma é um grafo orientado em que cada nó corresponde a uma

Instrução do programa, e cada seta indica o fluxo de controle de uma declaração para outra. A complexidade ciclomática de um grafo pode ser calculada através da fórmula da teoria dos grafos descrita a baixo:

$$V(G) = e - n + 2 \tag{6.1}$$

onde e é o número de arestas e n é o número de nós do grafo.

Complexidade Essencial ou ev(G)- é a medida em que o fluxograma pode ser reduzido pela decomposição de todos os subgrafos. A complexidade essencial é calculada pela formula a seguir:

$$ev(G) = v(G) - m \tag{6.2}$$

Onde v(G) é a complexidade ciclomática descrita anteriormente e "m"é o número de subgrafos próprios com única entrada e nós de saída.

Complexidade de Design ou iv(G) - é a complexidade reduzida do grafo. A redução é realizada para eliminar qualquer complexidade que não faz influencia na relação entre as funções do projeto. Para mais detalhe de como realizar a redução consulte McCabe and Butler [1989].

LOC - é número de linhas de código contadas com base nas convenções de McCabe. McCabe defendeu que o código com caminhos mais complexos são mais propensos a erros. Por isso, suas métricas indicam os caminhos dentro do código fonte de uma função.

Por outro lado, Halstead defendeu que códigos fonte que são difíceis de ler estão mais propenso a falhas. As métricas de Halstead estimam a complexidade da leitura através da contagem do número de conceitos em uma função. As métricas de Halstead se dividem em: medidas de base, medidas derivadas e medidas de linhas de código.

Medidas de base utilizadas neste trabalho são: Operadores Distintos (uniqOp), Operandos

Distinto (uniqOpnd), Total de Operadores (totalOp) e Total de Operandos (totalOpnd). Um exemplo do cálculo desses operadores seria dada a expressão:

$$\text{return } max(w + x, x + y) \tag{6.3}$$

obtemos os seguintes valores para cada métrica:

- uniqOp = 4, são eles: return, max e os dois operadores de soma (+,+).
- uniqOpnd = 4 , são as quatro variáveis: "w", "x", "x"e "y".
- totalOp = 3 (return, max,+).
- totalOpnd = 3 (w,x,y).

Medidas derivadas - Compostas pelas métricas descritas a seguir:

Volume (V) é o número de comparação mental para escrever um programa de tamanho N. obtida por:

$$V = N \cdot \log_2(uniqOp + uniqOpnd) \tag{6.4}$$

onde N = totalOp + totalOpnd.

Tamanho do programa (L) dada por:

$$L = \frac{((2 + uniq_Opnd) \cdot \log_2(2 + uniqOpnd))}{totalOp + totalOpnd} \tag{6.5}$$

Dificuldade (D), calculada pela expressão:

$$D = \frac{1}{L} \tag{6.6}$$

Inteligência (i), que é o inverso da dificuldade 1/D vezes o volume V:

$$I = \frac{1}{D} \cdot V \tag{6.7}$$

$$I = \frac{1}{\frac{1}{L}} \cdot V \tag{6.8}$$

$$I = L \cdot V \tag{6.9}$$

$$I = \frac{(2 + uniqOpnd) \cdot \log_2(2 + uniqOpnd)}{totalOp + totalOpnd}. \tag{6.10}$$

Esforço para escrever o programa (E) que é o volume dividido pelo tamanho do programa :

$$E = \frac{V}{L} \tag{6.11}$$

Estimativa de Erros (B) calculada por:

$$B = \frac{V}{3000} \tag{6.12}$$

Tempo de Programação (T)obtida pela equação:

$$T = \frac{E}{18} \tag{6.13}$$

Onde 18 é dado em segundos.

As Medidas de linhas de código de Halstead utilizadas neste trabalho são quatro: Linhas de Código (lOCode), Linhas de Comentário (lOComment), Linhas em Branco (lOBlank) e Linhas de Código e Comentário (lOCodeAndComment).

6.1.2 Instâncias

As medidas McCabe e Halstead extraídas em "módulo". Como já dito anteriormente em C um módulo é chamado de função e para este trabalho sempre usaremos o termo função ou

instância quando necessário. As bases originais estudadas variam o número de instâncias entre 498 e 1885 como exibido na tabela 6.2. Cada instância possui 22 atributos já descritos e um rótulo que informa se a instância(função) possui ou não defeito. Os valores do rótulo para CM1, JM1, KC1 e PC1 assumem *true* ou *false*, sendo *true* quando existe defeito e *false* caso não haja defeito. A base KC2 rotula *yes* para instâncias com defeito e *no* para quando não existe defeito na função.

Projeto (Bases)	Funções (Instâncias)	Quantidade de Instâncias com defeito	Quantidade de Instâncias sem defeito	Linguagem
CM1	498	49	449	C
JM1	10885	2106	8779	C
KC1	2109	326	1783	C++
KC2	522	107	415	C++
PC1	1109	77	1032	C

Tabela 6.2: Bases Estudadas

6.2 Método

6.2.1 Pré-processamento

Foi realizado os seguintes procedimentos nas bases de dados para se realizar o experimento:

- **Remover instâncias que possuem dados faltosos.** Apenas a base JM1 possuiu dados faltosos, foi retirado da base 5 instâncias que possuía atributos faltosos. A base agora possui 10880 instancia sendo 8777 da classe sem defeito e 2103 com defeito. Nesta etapa do pré-processamento as demais bases não foram alteradas.
- **Remover instâncias duplicadas das bases**. Nesta etapa foram removidas 56 instâncias da base CM1, 1973 de JM1, 897 de KC1, 252 de KC2 e 225 de PC1. A seguir é exibido um quadro com a quantidade de instâncias totais, com defeito e sem defeito após a remoção das instâncias duplicadas.

Projeto (Bases)	Sem Defeito	Com Defeito	Total
CM1	394	48	442
JM1	6903	2004	8907
KC1	897	315	1212
KC2	270	105	375
PC1	884	70	954

Tabela 6.3: Bases após remoção de dados duplicados.

- **Balancear**. Foi necessário pois existia uma quantidade muito superior de instância da classe sem defeito e isso poderia interferir tanto no experimento, quanto na análise do mesmo. Nessa etapa a base CM1 ficou com 48 instâncias com defeito, pois a mesma só possuía esta quantidade e 48 sem defeito totalizando 96 instâncias. As demais bases, por uma questão de desempenho e limitação de recursos computacionais para o processamento do experimento, ficaram com 70 instância em cada classe totalizando cada uma 140 instâncias.

- **Embaralhar as Instâncias**. Após ter balanceado as duas classes, as instâncias ficaram agrupadas em cada base. Além disso a ordem das instâncias para alguns algoritmos podem interferir em sua aprendizado.

- **Padronizar** as bases para detecção de instâncias atípica, que estão nos extremos (com valores muito alto ou baixo).

- **Alterar rótulos** das bases para 0 quando a instância não possuía defeito e 1 para as instâncias com defeitos, foi necessário, pois queríamos aplicar algoritmos de regressão para obter um *rankin* e poder comparar com o método proposto por Amland [1995].

6.2.2 Ajuste de Parâmetros Para os Algoritmos de Aprendizado de Máquina

Para nos auxiliar no experimento dos algoritmos de AM, foi utilizando o software wek. O algoritimo para a execução do SVM de regressão escolhido foi o SVMreg diponível no Weka.

O algoritmo utilizado para RNA foi o MLP e para o comitê de classificação foi o bagging ambos disponíveis no Weka.

- SVMreg. Foi utilizado o kernel polinomial. Variamos o parâmetro C em $\{10^{-3}, ..., 2^{9}\}$ e Kernel foi variado de 1 a 3.
- MLP. O Learning rate variou de $\{10^{-4}, ..., 10^{-1}\}$ e a quantidade de neurônios escondidos variaram de 2 a 50.
- Bagging. Foi escolhido o algoritmo MLP e o numero de interações variou de 10 a 75.

6.2.3 Execução do Experimento para a Probabilidade do Risco P(f)

A probabilidade do risco P(f) é calculada através da média ponderada das métricas: Nova funcionalidade (que possui peso 5), Design de Qualidade (cujo peso também é 5), Tamanho (peso é 1) e Complexidade (de peso igual a 3).

- Complexidade assumiu o valor do atributo v(g) que é a "cyclomatic complexity"de McCabe explicada na equação 6.1.
- O Design de Qualidade assumiu os valores do atributo "design complexity"que também é uma métrica de McCabe.
- O tamanho assumiu os valores do atributo "program length"que é uma métrica de Halstead.
- Por fim a métrica nova funcionalidade, como o autor do estudo de caso Amland [1995] nos informa que o projeto estudado foi uma reengenharia do sistema já existente e que a métrica foi utilizada para avaliar alterações usamos o atributo "time estimato"de Halstead's, pois quando houve alteração, existia uma quantidade de horas trabalhadas diferente de 0 e quando a função não foi alterada a quantidade de horas trabalhada foi zero.

Em seguida os valores de cada uma dessas 4 métricas foram divididos em 3 partes. Onde os valores mais baixos assumiram valor igual a 1, médios 2 e valores mais altos assumiram

valores 3, assim como descrito na seção 4.1 deste trabalho. Em seguida os valores de cada métrica foram multiplicados pelos seus respectivos pesos, como descrito também na seção 4.1 e por fim foi calculado a probabilidade da media ponderada dessas 4 métricas. Para realizar este experimento foi necessário a construção de um *script* para calcular a probabilidade P(f) para todas as bases. O *script* foi construído em python.

6.3 Validação

Após a seleção do melhor conjunto de parâmetros dos algoritmos de AM, etapa onde foi utilizado o método de validação cruzada para a escolha dos melhores parâmetros para cada base de dado, os melhores resultados obtidos foram comparados com o método P(f). Para realizar esta comparação utilizamos o MAP *Mean average precision* que é a precisão média, para média obtida de um conjunto de dados de uma consulta realizada. Seu resultado é obtido pela equação a seguir:

$$\text{MAP} = \frac{\sum_{q=1}^{Q} Avg(P)}{Q} \tag{6.14}$$

onde, Q é o número de consultas.

O resultado do MAP de cada algorítimo de AM foi comparado com o resultado do MAP de P(f). Para essa comparação foi utilizado o teste de hipótese. Os resultados estão na tabela a seguir.

	P(f)	**SVMreg**	**MLP**	**Bagging**
CM1	0,43 ± 0,18	0,73 ± 0,15	0,75 ± 0,15	0,76 ± 0,16
JM1	0,56 ± 0,18	0,73 ± 0,11	0,83 ± 0,11	0,82 ± 0,11
KC1	0,38 ± 0,23	0,79 ± 0,15	0,79 ± 0,15	0,77 ± 0,14
KC2	0,59 ± 0,28	0,89 ± 0,09	0,93 ± 0,09	0,96 ± 0,06
PC1	0,48 ± 0,23	0,83 ± 0,12	0,86 ± 0,12	0,91 ± 0,08
	-	(5/0/0)	(5/0/0)	(5/0/0)

Tabela 6.4: T-Teste dos algoritmos de AM comparado com P(f).

6.4 Resultados

Após a execução dos algoritmos de AM e do cálculo da probabilidade P(f) proposto em Amland [1995] as bases de dados foram divididas em 10 parte e foi calculado a média do MAP e o desvio padrão do valor predito por cada método para cada experimento.

O *Mean Average Precision* (MAP) foi escolhido para avaliar o desempenho, pois queríamos analisar quais métodos retornou as classes com defeito nas primeiras posições do seu rankig. O MAP melhora (seu valor ótimo é 1) caso as instâncias com defeito esteja nas primeiras colocações do ranking feito por cada método.

Os resultado obtidos de cada algoritmo de AM foram comparados com os resultados da probabilidade P(f) proposto em Amland [1995] com o T-teste sendo todos rejeitados, isto é, os valores obtido da média do MAP descritos na tabela 6.5 para cada base são diferentes do MAP obtido nas demais tabelas (6.6, 6.7, 6.8) dos algoritmos de AM.

Bases	Média do MAP	Desvio Padrão
CM1	0,43	0,18
JM1	0,56	0,18
KC1	0,38	0,23
KC2	0,59	0,28
PC2	0,48	0,23

Tabela 6.5: Resultados do MAP para os experimento utilizando P(f).

Bases	Média do MAP	Desvio Padrão
CM1	0,73	0,15
JM1	0,73	0,11
KC1	0,79	0,15
KC2	0,89	0,09
PC2	0,83	0,12

Tabela 6.6: Resultados do MAP para os experimento utilizando SVMreg.

Bases	Média do MAP	Desvio Padrão
CM1	0,75	0,15
JM1	0,83	0,11
KC1	0,79	0,15
KC2	0,93	0,09
PC2	0,86	0,12

Tabela 6.7: Resultados do MAP para os experimento utilizando MLP.

Bases	Média do MAP	Desvio Padrão
CM1	0,76	0,16
JM1	0,82	0,11
KC1	0,77	0,14
KC2	0,96	0,06
PC2	0,91	0,08

Tabela 6.8: Resultados do MAP para os experimento utilizando Bagging.

Abaixo segue uma tabela com a precisão de cada método estudado para cada base de dados. Ela nos revela o quanto cada método acertou ao informar o conjunto de instâncias que pertencia a classe defeituosa. Isto significa que quanto mais próximo do valor 1, que é o valor ótimo, Melhor foi o resultado do método. Isto é, caso o valor seja 1 todas as instâncias retornadas pelo método como defeituosa, realmente tinha defeito. Já quanto mais próximo de 0 uma quantidade maior de instância que não havia defeito, o método informou que havia defeito.

	CM1	JM1	KC1	KC2	PC1
P(f)	0,56	0,69	0,51	0,63	0,59
SVMreg	0,94	1,00	0,87	1,00	1,00
MLP	0,84	0,93	0,85	1,00	0,98
BAG	0,88	0,95	0,86	1,00	0,98

Tabela 6.9: PRECISION.

A tabela (6.10) traz os resultados do *Recall* de cada experimento. Estes valores nos informam quanto o método foi bom em resgatar todas as instâncias que eram defeituosas. Quando o valor é "1"significa que o método resgatou todas as instâncias defeituosas, ou seja, o método informou todas as instâncias que havia defeito. Por outro lado quanto mais próximo de zero uma quantidade maior de instâncias que havia defeito, o método não conseguiu retornar (ou prever) que a instância era defeituosa.

	CM1	JM1	KC1	KC2	PC1
P(f)	0,87	0,71	0,80	1,00	0,87
SVMreg	0,68	0,68	0,82	0,87	0,80
MLP	0,79	0,82	0,81	0,92	0,90
BAG	0,77	0,81	0,82	0,95	0,90

Tabela 6.10: RECALL.

6.5 Análise dos resultados

Neste trabalho foi realizado um experimentos com 4 base de dados CM1, JM1, KC1, KC2 e PC1. E utilizamos 3 algoritmos de aprendizado de máquina de paradigmas diferentes, o SVMreg do paradigma SVM, o MLP do paradigma de redes neurais artificiais e o Bagging do paradigma de comitê de classificação, todos aplicados para regressão dos dados. Além desses 3 métodos, utilizamos também o cálculo da probabilidade P(f) de uma função falhar, proposto no trabalho [Amland, 1995]. O valor do MAP para cada base de dados em cada método aplicado pode ser verificados nas tabelas (6.5), (6.6), (6.7), (6.8) e estão resumidos no gráfico da figura a baixo.

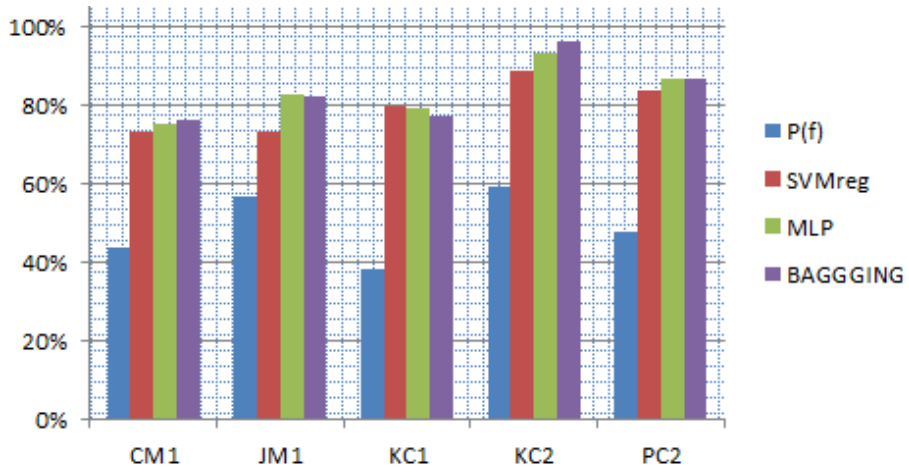

Figura 6.1: Média do desempenho de cada Método analisado.

O melhor desempenho do método P(f) foi com a base KC2, com o resultado de 59%. Isto também se verificou para os demais métodos, que também obtiveram seu melhor resultado com a base KC2. O SVMreg obteve um desempenho de 89% para KC2, MLP teve 94% em KC2 e *Bagging* teve 96% de desempenho para base KC2.

No gráfico verificamos que o método P(f), teve o seu pior desempenho com a base KC1 em que seu resultado foi de apenas 38% e depois com a base CM1 onde o desempenho foi de 44%. O SVMreg teve seu pior desempenho do MAP com a base CM1 com 73% e em seguida com as bases JM1 (74%) e KC1 (80%). O MLP assim como o método SVMreg e o *Bagging*, teve seu pior desempenho com a base CM1 que foi de 76% e o do *Bagging* foi de 77%.

Podemos verificar que o MAP da probabilidade P(f) teve o pior desempenho entre os quatro métodos avaliados. Isso pode ser justificado pelo fato do método ter retornado instâncias que não possuíam defeitos como sendo defeituosas. Para o método P(f) e para os demais métodos consideramos as instâncias defeituosas quando o valor retornado pelo método foi maior ou igual a 0,5, ou seja, tinham 50% de chance ou mais de terem defeitos.

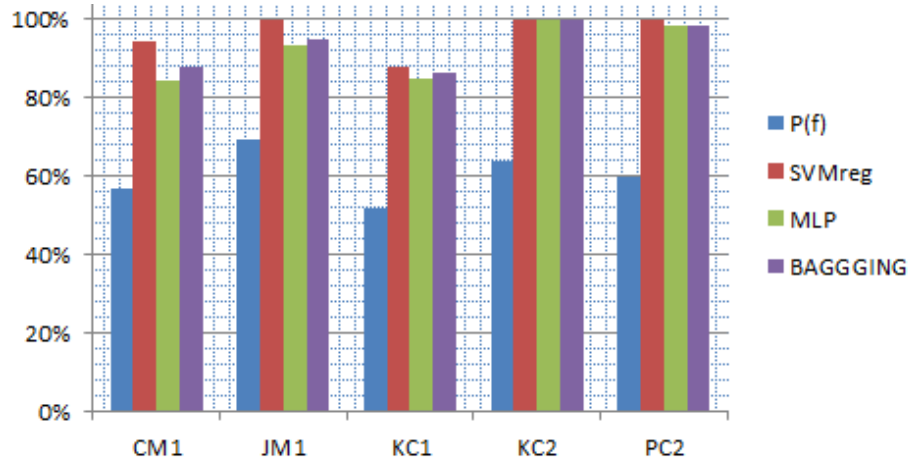

Figura 6.2: Precisão de cada método analisado.

Podemos verificar no gráfico da precisão (figura 6.2) que o método P(f) também teve um fraco desempenho em sua precisão, embora o método tenha atingido valores elevado no seu *recall*. Isso pode ser devido ao método P(f) realizar o cálculo da probabilidade entre a média ponderada de quatro atributos e assumem que não existe correlação entre esses atributos, isto é, o valor de um atributo não influencia o do outro.

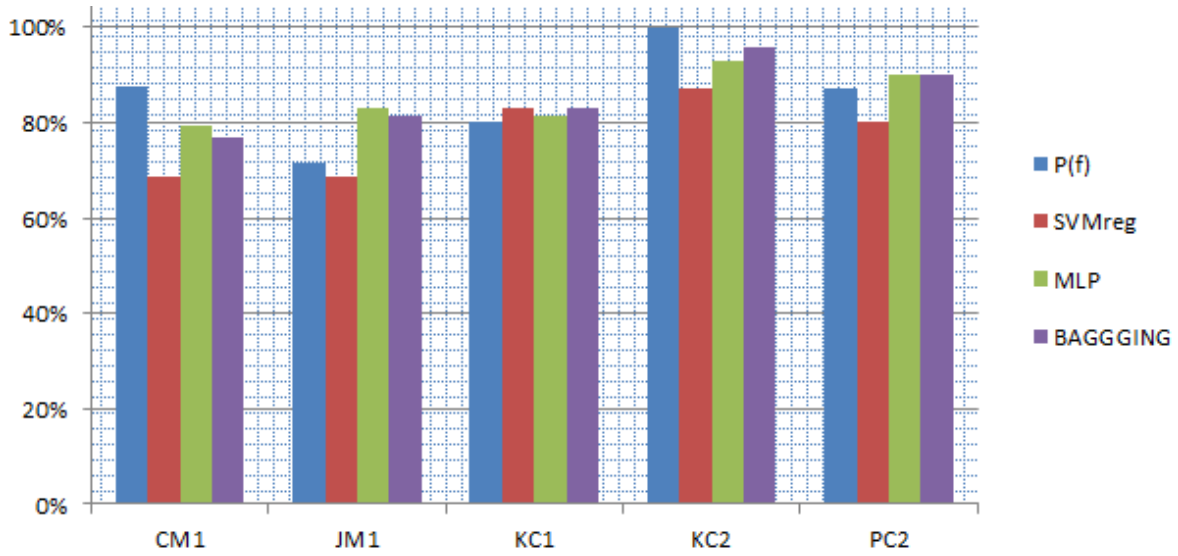

Figura 6.3: Recall de cada método analisado.

O método P(f) teve um ótimo *recall*, ganhou dos outro métodos em duas bases KC2 e CM1. Seu melhor *recall* foi para base KC2 onde atingiu 100% e em seguida para base CM1 com 88%. O MLP ganhou no desempenho do *recall* com a base JM1 (83%). O *Bagging* foi o

método que obteve o melhor desempenho do *recall* na base KC1(83%). O melhor *recall* da base PC2 foi de 90% dos algorítmos MLP e Bagging.

Os métodos P(f), MLP e *Bagging* tiveram o melhor desempenho, em média obtiveram um *recall* de 85%. O pior desempenho dos métodos analisado, no *recall* foi do SVMreg seu melhor desempenho foi de 87% para KC2 e o pior foi de 69% para as bases CM1 e JM1. O fraco desempenho do SVM pode se justificar pelo fato desse método só depender do seu vetor de suporte e não consiga identificar possíveis relações entre os valores de cada atributo.

6.6 Conclusão

Neste capítulo apresentamos como foi desenvolvido os experimentos. Descrevemos as bases de dados utilizadas, seus atributos e suas instâncias. Mostramos como foi realizada a etapa de pré-processamento, vimos que as bases eram desbalanceadas e que era necessário balancear as bases para obter resultados mais precisos. Mostramos os métodos utilizados nos experimentos e os ajustes de parâmetros para cada algoritmo de AM. Foi dito que para a execução do cálculo da probabilidade do risco P(f) foi necessário escrever um *script* em *python*. Apresentamos ainda neste capítulo os métodos usados para validação dos dados, onde foi utilizado o MAP para calcular o desempenho de cada algoritmo de AM e da probabilidade P(f). Foi mostrado também que para comparar o desempenho de cada algoritmo de AM com P(f) foi utilizado o teste de hipótese. Além do MAP como já foi dito analisamos também o valor da precisão e do *recall*. Por fim, exibimos os resultados para as medidas analisadas e concluímos que a probabilidade do risco P(f) teve um baixo desempenho para o MAP e a precisão. Porém P(f) teve um desempenho similar aos desempenhos dos algoritmos de AM quando analisado o *recall*. O que demostra que o método P(f) teve um bom desempenho para encontrar as instâncias que possuíam defeito mas apresentou uma quantidade elevada de falsos positivos.

Capítulo 7

Conclusão

O objetivo deste trabalho foi avaliar o desempenho da detecção de erros em funções de software, entre os algorítimo de AM e da probabilidade P(f) da abordagem de testes baseados em riscos e em métricas. Para isso foi realizado experimentos com bases de dados da engenharia de software. Apresentamos alguns trabalhos na área de predição de defeito e suas propostas para resolver o problema de encontrar funções com defeitos, utilizando algorítimos de AM. Alguns algorítimos de AM foram apresentados neste trabalho como proposta para resolver o problema da detecção de defeitos em funções de software que foram eles: SVMreg, MLP e *Bagging*.

Analisamos os desempenhos dos algoritmos de AM propostos neste trabalho e da probabilidade P(f). Os resultados mostraram que os algoritmos MLP e *Bagging* tiveram desempenho similares a probabilidade P(f) para retornar as funções de um sistema que possuía defeito e que o algoritmo SVMreg teve o desempenho inferior nesta análise. Em contra partida, a abordagem que utiliza probabilidade para prever a chance de uma função ter defeitos retornou um alto número de falsos positivos, ou seja, retornou instâncias como defeituosas quando na realidade elas não possuíam defeitos. Já os métodos da AM tiveram um bom desempenho neste ponto. Todos retornaram uma quantidade menor de falsos positivos em comparação com o método que utiliza a probabilidade. Por isso todos os algoritmos de AM tiveram uma performa melhor do que a de P(f) ao ser comparados com as medidas de desempenho utilizada neste trabalho (MAP). Em resumo podemos dizer que para este experimento os algorítimos MLP e *Bagging* tiveram um desempenho igual ou superior ao do P(f).

7.1 Trabalhos Futuros

Uma possibilidade de estudo que complementaria este trabalho seria a análise dos 22 atributos utilizados nas bases usadas no experimento. Seria interessante analisar o desempenho dos algoritmos de AM em relação ao 22 atributos utilizados e a relevância de cada um para predição de defeitos em softwares.

Como visto nesse trabalho todos os métodos analisados tanto de AM como a probabilidade do risco P(f) tiveram um desempenho melhor para a base KC2, apesar de todas as bases de dados utilizarem os mesmos atributos e serem todas da mesma instituição. Talvez isso se deva a forma como foram interpretadas por quem realizou as medições, ou coma já citado neste trabalhos, uma mesma funcionalidade construída em linguagem de programação diferente, resulta em diferentes medições estáticas. Uma outra possibilidade de trabalho complementar ao nosso estudo seria analisar as formas de extração de métricas estáticas de código para predição de defeito.

Referências Bibliográficas

Weka 3: Data mining software in java. `http://www.cs.waikato.ac.nz/~ml/weka/`. Accessed: 2015-12-11.

S. Amland. Risk based testing and metrics: Risk analysis fundamentals and metrics for software testing including a financial application case study. 1995.

J. Bach. Heuristic risk-based testing. *Software Testing and Quality Engineering Magazine*, pages 23–28, 1999.

A. Braga, A. Carvalho, and L. T. *Redes Neurais Artificiais Teoria e Aplicação*. 10. The name of the publisher, LTC Editora, Rio de Janeiro-RJ, 2000.

L. Breiman. Bagging predictors. *Machine Learning v. 24, n. 2*, pages 123–140, 1999.

BSTQB. *Certified Tester Advanced Level Syllabus Test Manager (TM)*. 9 2012.

I. Burnstein. *Practical Software Testing*. Springer Professional Computing, 2000. ISBN 0-387-95131-8.

N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. The name of the publisher, The address, 3 2000. ISBN 9780521780193.

K. Elish and M. Elish. Predicting defect-prone software modules using support vector machines. *J. Syst. Softw*, pages 649–660, 2008.

N. Fenton. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, Boston, MA, USA ©1996, 2 edition, 1996. ISBN 1850322759.

D. Gray, D. Bowes, N. Davery, Y. Sun, and B. Christianson. Using the support vector machine as a classification method for software defect prediction with static code metrics. *Communications in Computer and Information Science V. 43*, pages 223–234, 2014.

IEEE. *IEEE Standard for Software Test Documentation.* 9 1998.

C. Kaner, J. Falk, and H. Q. Nguyen. *Combining Pattern Classifier: Methods and Algorithms.* New York: Wiley, 2004.

L. Kuncheva. *Combining Pattern Classifier: Methods and Algorithms.* New York: Wiley, 2004.

S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions Science*, pages 485–496, 2008.

Z. Li and M. Reformat. A practical method for the software fault-prediction. information reuse and integration. *IEEE International Conference*, pages 659–666, 2007.

A. Lorena and A. Carvalho. Uma introdução às support vector machines. *Revista de Informática Teórica e Aplicada*, 14(2):43–67, 1 2007. An optional note.

T. McCabe and C. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, 12 1989.

L. Morais. Um processo para priorização de casos de teste de regressão. Master's thesis, Centro de Informática-Universidade Federal de Pernambuco, Recife, 3 2012.

N. Nagappan. Static analysis tools as early indicators of pre-release defect density. *international conference on Software engineering (ICSE 05)*, pages 580–586, 2005.

R. Pressman. *Engenharia de software.* Springer Professional Computing, 6 edition, 2009. ISBN 9788563308009.

E. Rios. *Análise de Riscos em Projetos de Teste de Software 131p.* Castelo Rio de Janeiro: Alta Books, 2008.

R. Soares. Uso de meta-aprendizado para a seleção e ordenação de algoritmos de agrupamento aplicados a dados de expressão gênica. Master's thesis, Centro de Informática-Universidade Federal de Pernambuco, Recife, 2 2008.

E. Souza. Rbtprocess: Modelo de processo de teste de software baseado em riscos. 2008 144 f. Master's thesis, Centro de Informática-Universidade Federal de Pernambuco, Recife-PE, 12 2008.

SWEBOK. *SWEBOK Guide to the Software Engineering Body of Knowledge*. 2004.

R. Wahono and N. Herman. Genetic feature selection for software defect prediction. *Advanced Science Letters V. 20*, pages 239–244, 2014.