

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Software Repository Mining for Estimating Software Component Reliability

André Duarte - ei11044@fe.up.pt

PREPARAÇÃO DA DISSERTAÇÃO



Mestrado Integrado em Engenharia Informática e Computação

Orientador: Rui Maranhão - rma@fe.up.pt

7 de Fevereiro de 2016

Software Repository Mining for Estimating Software Component Reliability

André Duarte - ei11044@fe.up.pt

Mestrado Integrado em Engenharia Informática e Computação

Resumo

Com a crescente necessidade de identificar a localização dos erros no código fonte de *software*, de forma a facilitar o trabalho dos programadores e a acelerar o processo de desenvolvimento, muitos avanços têm sido feitos na sua automação.

Existem três abordagens principais: *Program-spectra based* (PSB), *Model-based diagnosis* (MDB) e *Program slicing*.

Barinel, solução que integra tanto o PSB como o MDB, é, até hoje, com base na investigação feita, a que apresenta melhores resultados. Contudo, a ordenação de conjuntos de candidatos (componentes faltosos) não tem em conta a verdadeira qualidade do componente em causa, mas sim o conjunto de valores que maximizam a probabilidade do conjunto (*Maximum Likelihood Estimation* - MLE), devido à dificuldade da sua determinação.

Com esta tese pretende-se colmatar esta falha e contribuir para uma melhor ordenação dos conjuntos, classificando, com recurso a técnicas de Machine Learning como *Naive Bayes*, *Support Vector Machines* (SVM) ou *Random Forests*, a qualidade e fiabilidade de cada componente, através das informações disponíveis no sistema de controlo de versões (*Software Repository Mining*), neste caso *Git*, como por exemplo: número de vezes que foi modificado, número de contribuidores, data de última alteração, nome de últimos contribuidores e tamanho das alterações.

A investigação já feita, revelou a existência de algumas soluções de análise preditiva de *software*, como *BugCache*, *FixCache* e *Change Classification*, capazes de identificar componentes com grande probabilidade de falhar e de classificar as revisões (*commits*) como faltosas ou não, mas nenhuma soluciona o problema.

Este trabalho visa também a integração com o *Crowbar* e a contribuição para a sua possível comercialização.

Palavras-chave: *Software-fault Localization, Software Repository Mining, Machine Learning, Classification*

Classificação: *Software and its engineering - Software creation and management - Software verification and validation; Computing methodologies - Machine Learning - Machine Learning Approaches*

“Software is eating the world.”

Marc Andreessen

Conteúdo

1	Introdução	1
1.1	Contexto/Enquadramento	1
1.2	Motivação e Objetivos	1
1.3	Estrutura da Dissertação	2
2	Revisão Bibliográfica	3
2.1	<i>Fault Localization Software</i>	3
2.1.1	<i>Spectrum-based diagnosis</i>	3
2.1.2	<i>Model-based diagnosis</i>	4
2.1.3	<i>Barinel</i>	4
2.2	<i>Software Repository Mining</i>	6
2.3	Abordagens à predição de defeitos	6
2.3.1	<i>BugCache</i>	6
2.3.2	<i>FixCache</i>	6
2.3.3	<i>Buggy Change Classification</i>	6
2.4	<i>Machine Learning</i>	6
2.4.1	<i>Naïve Bayes</i>	6
2.4.2	<i>Support Vector Machines (SVM)</i>	6
2.4.3	<i>Random Forests</i>	6
3	Perspectiva de Solução	7
4	Validação	9
5	Conclusões	11
	Referências	13

CONTEÚDO

Lista de Figuras

LISTA DE FIGURAS

Lista de Tabelas

2.1	<i>Hit-spectra matrix</i>	4
2.2	<i>Hit-spectra matrix</i>	5

LISTA DE TABELAS

Abreviaturas e Símbolos

MSR	Mining Software Repositories
SFL	Spectrum-based Fault Localization
PSB	Program-spectra based
MDB	Model-based diagnosis
MLE	Maximum Likelihood Estimation
SVM	Support Vector Machines

Capítulo 1

Introdução

1.1 Contexto/Enquadramento

Com o elevado crescimento da indústria de desenvolvimento de software, torna-se cada vez mais importante a existência de ferramentas que auxiliem os programadores a desenvolvê-lo mais eficientemente.

Estima-se que a economia dos Estados Unidos perca cerca de 60 mil milhões de dólares por ano em custos associados ao desenvolvimento e distribuição de correções para defeitos de *software* e na sua reinstalação. Pelo que, podemos afirmar que as ferramentas de localização das falhas de *software* (*Software Fault Localization*), ajudando a reduzir o tempo investido nesta tarefa, poderão ter um impacto significativo na economia. Nesta área os avanços são consideráveis. *Ochiai*, *Tarantula*, *Bayes-A* e *Barinel* são apenas algumas das soluções existentes, sendo o algoritmo *Barinel* aquele que apresenta melhores resultados. Apesar dos bons resultados apresentados pelo *Barinel*, este poderá apresentar resultados ainda mais rigorosos se tivermos informações relativas ao projeto, como a probabilidade média de erro ou a probabilidade de dado componente, que o constitui, ter defeitos.

Ferramentas de controlo de versões, como o *Git*, uma vez que mantêm todo o histórico do projeto e informações relacionadas com as diversas alterações (p.e. conteúdo, data e autores), em conjunto com técnicas de *Machine Learning*, poderão ser a chave para a melhoria deste algoritmo.

1.2 Motivação e Objetivos

Tendo em conta as possibilidades que a extração de dados de repositórios de controlo de versões e o *Machine Learning* nos dão, pretende-se com esta dissertação:

- Optimizar a ordenação de resultados candidatos do algoritmo *Barinel*.
- Ter a capacidade de prever a probabilidade de erro de cada um dos componentes de um dado projecto de *software* que use o *Git* para controlo total de versões, com uma precisão útil.

1.3 Estrutura da Dissertação

Para além da introdução, esta dissertação contém mais 3 capítulos. No capítulo 2, é descrito o estado da arte e são apresentados trabalhos relacionados.

Completar, quando terminar

Capítulo 2

Revisão Bibliográfica

Neste capítulo é feita uma revisão bibliográfica e descrito o estado da arte do *software* de localização de falhas, como o Barinel, das diferentes abordagens de predição de defeitos e ainda de *Software Repository Mining*.

2.1 *Fault Localization Software*

O *Fault Localization Software* auxilia na localização automática do código que origina falhas na sua execução, diminuindo o custo desta identificação que teria de ser feita manualmente pelo programador. Existem duas categorias principais: *Spectrum-based diagnosis* e *Model-based diagnosis*.

Vale a pena incluir Program Slicing?

2.1.1 *Spectrum-based diagnosis*

Spectrum-based Fault Localization (SFL) é uma técnica estatística de detecção de falhas que calcula a probabilidade de cada componente de *software* conter falhas, através da análise de informação relativa às execuções, bem sucedidas ou falhadas [AZV07]. Esta técnica apresenta bons resultados quando o projeto têm um número elevado de casos de teste e é capaz de executar num tempo reduzido, escalando bem para projetos grandes [MS08].

Esta técnica gera uma matriz, com base nos dados guardados durante a execução (*program spectrum* [RBDL97]), que relaciona as execuções de casos de teste, com os componentes que executou e com o sucesso ou insucesso do mesmo.

	<i>obs</i>			<i>e</i>
	<i>c</i> ₁	<i>c</i> ₂	<i>c</i> ₃	
<i>t</i> ₁	1	1	0	1
<i>t</i> ₂	0	1	1	1
<i>t</i> ₃	1	0	0	1
<i>t</i> ₄	1	0	1	0

 Tabela 2.1: *Hit-spectra matrix*

Com esta matriz, também denominada *hit-spectra matrix*, é calculado o coeficiente de similaridade (*similarity coefficient*) para cada um dos componentes [AZG09], que corresponde à probabilidade desse componente ter uma falha. A forma como este coeficiente é calculado difere de algoritmo para algoritmo. Dando como exemplos o Pinpoint [CKF⁺02], o Tarantula [JH05] e o Ochiai [AZvG07], que têm os coeficientes respectivamente calculados do seguinte modo

$$s_J(j) = \frac{a_{11}(j)}{a_{11}(j) + a_{01}(j) + a_{10}(j)} \quad (2.1)$$

$$s_T(j) = \frac{\frac{a_{11}(j)}{a_{11}(j) + a_{01}(j)}}{\frac{a_{11}(j)}{a_{11}(j) + a_{01}(j)} + \frac{a_{10}(j)}{a_{10}(j) + a_{00}(j)}} \quad (2.2)$$

$$s_O(j) = \frac{a_{11}(j)}{\sqrt{(a_{11}(j) + a_{01}(j)) * (a_{11}(j) + a_{10}(j))}} \quad (2.3)$$

2.1.2 *Model-based diagnosis*

O princípio base do diagnóstico baseado no modelo (*Model-based diagnosis*) é o de comparar o modelo, isto é a descrição de funcionamento do sistema, ao comportamento efetivamente observado [MS08]. Sendo depois a diferença entre os dois usada para identificar os componentes que possam explicar os erros. Isto na prática requer uma descrição formal do sistema, o que torna a tarefa bastante difícil [PAW04].

De forma a facilitar o uso deste método recorre-se por vezes à inferência do modelo, através do próprio *software*, mais especificamente através dos testes definidos neste [PAW04].

Apesar da elevada fiabilidade dos resultados que resultam desta técnica, o esforço computacional necessário na criação do modelo de uma programa de grande dimensão impede, maior parte das vezes, o seu uso em projetos reais [MS08].

2.1.3 *Barinel*

O *Barinel* é um algoritmo que se inspira nos dois métodos descritos anteriormente, *program-spectra based* e *model-based diagnosis*, e que com isto consegue melhores resultados que as outras soluções com um custo pouco superior [AZG09].

O algoritmo começa por analisar uma *hit-spectra matrix*, que representa os testes executados em relação aos componentes que foram executados e ao seu resultado final.

	<i>obs</i>			<i>e</i>
	<i>c</i> ₁	<i>c</i> ₂	<i>c</i> ₃	
<i>t</i> ₁	1	1	0	1
<i>t</i> ₂	0	1	1	1
<i>t</i> ₃	1	0	0	1
<i>t</i> ₄	1	0	1	0

 Tabela 2.2: *Hit-spectra matrix*

Na tabela 2.2, temos identificados 3 componentes distintos (*c*₁, *c*₂ e *c*₃), 4 testes executados (*t*₁, *t*₂, *t*₃ e *t*₄) e o respectivo resultado da execução (*e*). O valor 1 em qualquer uma das colunas das observações (*obs*) indica que o dado componente foi executado nesse teste e o valor 0 indica o contrário, que o componente não foi executado. Na coluna *e*, o algarismo 1 declara que o teste correspondente falhou. Pelo que, por exemplo, o teste *t*₄ executou os componentes *c*₁ e *c*₃ e foi concluído com sucesso.

2.1.3.1 Geração de candidatos

Com base nesta matriz, uma lista de conjuntos de candidatos (*d*) é gerada, sendo esta reduzida ao número mínimo de candidatos possível.

Neste caso, seriam gerados apenas dois candidatos:

- $d_1 = \{c_1, c_2\}$
- $d_2 = \{c_1, c_3\}$

2.1.3.2 Ordenação de candidatos

Para cada candidato *d*, é calculada a probabilidade de acordo com a regra de *Naïve Bayes*:

$$\Pr(d \mid obs, e) = \Pr(d) \cdot \prod_i \frac{\Pr(obs_i, e_i \mid d)}{\Pr(obs_i)} \quad (2.4)$$

$\Pr(obs_i)$ é apenas um termo normalizador idêntico para todos os candidatos, pelo que não é usado para proceder à ordenação.

Sendo p_j a probabilidade à *priori* do componente *c*_{*j*} originar uma falha, podemos definir $\Pr(d)$, probabilidade do candidato ser responsável pelo erro, não tendo em conta evidências adicionais, como

$$\Pr(d) = \prod_{j \in d} p_j \cdot \prod_{j \notin d} (1 - p_j) \quad (2.5)$$

Sendo g_j (*component goodness*) a probabilidade do componente *c*_{*j*} executar de forma correta, temos que

$$\Pr(obs_i, e_i \mid d) = \begin{cases} \prod_{j \in (d \cap obs_i)} g_j & \text{if } e_i = 0 \\ 1 - \prod_{j \in (d \cap obs_i)} g_j & \text{otherwise} \end{cases} \quad (2.6)$$

Tendo em conta o nosso exemplo

$$\Pr(d_1 \mid obs, e) = \overbrace{\left(\frac{1}{1000} \cdot \frac{1}{1000} \cdot \left(1 - \frac{1}{1000} \right) \right)}^{\Pr(d)} \times \overbrace{\underbrace{(1 - g_1 \cdot g_2)}_{t_1} \times \underbrace{(1 - g_2)}_{t_2} \times \underbrace{(1 - g_1)}_{t_3} \times \underbrace{g_1}_{t_4}}^{\Pr(obs, e \mid d)} \quad (2.7)$$

$$\Pr(d_2 \mid obs, e) = \overbrace{\left(\frac{1}{1000} \cdot \frac{1}{1000} \cdot \left(1 - \frac{1}{1000} \right) \right)}^{\Pr(d)} \times \overbrace{\underbrace{(1 - g_1)}_{t_1} \times \underbrace{(1 - g_3)}_{t_2} \times \underbrace{(1 - g_1)}_{t_3} \times \underbrace{g_1 \cdot g_3}_{t_4}}^{\Pr(obs, e \mid d)} \quad (2.8)$$

Quando existem valores g_j desconhecidos, é maximizado o valor de $\Pr(obs, e \mid d)$ usando o algoritmo *Maximum Likelihood Estimation* (MLE).

Neste caso, todos os valores de g_j são desconhecidos. Executando o algoritmo MLE para ambas as funções e calculando o resultado final temos que:

- $\Pr(d_1, obs, e) = 1.9 \times 10^{-9}$ ($g_1 = 0.47$ e $g_2 = 0.19$)
- $\Pr(d_2, obs, e) = 4.0 \times 10^{-10}$ ($g_1 = 0.41$ e $g_3 = 0.50$)

Vale a pena falar da Assertion Confidence?

Devo incluir alguma coisa sobre o Crowbar/GZoltar?

2.2 Software Repository Mining

Identificação de bugs - Erros libgit2 node-git

2.3 Abordagens à predição de defeitos

2.3.1 BugCache

2.3.2 FixCache

2.3.3 Buggy Change Classification

2.4 Machine Learning

2.4.1 Naïve Bayes

2.4.2 Support Vector Machines (SVM)

2.4.3 Random Forests

Capítulo 3

Perspectiva de Solução

Perspectiva de Solução

Capítulo 4

Validação

Validação

Capítulo 5

Conclusões

Conclusões

Referências

- [AZG09] Rui Abreu, P Zoetewij e a J C Van Gemund. Spectrum-Based Multiple Fault Localization. *Automated Software Engineering 2009 ASE 09 24th IEEEACM International Conference on*, pages 88–99, 2009. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5431781>, doi:10.1109/ASE.2009.25.
- [AZV07] Rui Abreu, Peter Zoetewij e Arjan J C Van Gemund. On the accuracy of spectrum-based fault localization. *Proceedings - Testing: Academic and Industrial Conference Practice and Research Techniques, TAIC PART-Mutation 2007*, pages 89–98, 2007. doi:10.1109/TAICPART.2007.4344104.
- [AZvG07] Rui Abreu, Peter Zoetewij e Arjan J.C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, sep 2007. URL: <http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4344104>, doi:10.1109/TAIC.PART.2007.13.
- [CKF⁺02] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox e Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, 2002. doi:10.1109/DSN.2002.1029005.
- [JH05] J.a. James a Jones e Mary Jean M.J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. *Automated Software Engineering*, pages 282–292, 2005. URL: <http://portal.acm.org/citation.cfm?id=1101949>, doi:http://doi.acm.org/10.1145/1101908.1101949.
- [MS08] Wolfgang Mayer e Markus Stumptner. Evaluating Models for Model-Based Debugging.pdf. pages 128–137, 2008.
- [PAW04] Alexandre Perez, Rui Abreu e Eric Wong. A Survey on Fault Injection Techniques. 1:171–186, 2004. doi:10.1.1.167.966.
- [RBDL97] Thomas Reps, Thomas Ball, Manuvir Das e James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *ACM SIGSOFT Software Engineering Notes*, 22(6):432–449, 1997. doi:10.1145/267896.267925.