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

8 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

8 de Fevereiro de 2016

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 Likehood 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

CONTEÚDO

Lista de Figuras

LISTA DE FIGURAS

Lista de Tabelas

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 Likehood 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 entre 60 mil milhões de dólares por ano em custos associados ao desenvolvimento e distribuição de correções para defeitos de *software* e na sua reinstalação [ZC09]. 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. [AZG09] 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*.

2.1.1 *Program Slicing*

Introduzida por Mark Weiser em 1981 [Wei81, Wei82], esta técnica começa a análise a partir da falha e através do fluxo de dados e de controlo do programa tenta chegar à origem do problema.

Removendo todas as operações que não afetam os dados ou o fluxo do programa, este método define uma porção de código (*slice*) que diz respeito às operações que poderão ser a causa do problema e que terão de ser inspecionadas [PAW04]. Ao reduzir o número de operações que necessitam de ser analisadas, o tempo necessário para a correção do erro diminui.

Normalmente uma análise estática do programa resulta numa porção de código grande, havendo outras técnicas, dinâmicas, que reduzem consideravelmente o seu tamanho, como *program dice* ou *dynamic program slicing* [PAW04].

2.1.2 *Spectrum-based diagnosis*

Spectrum-based Fault Localization (SFL) é uma técnica estatística de deteção 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.

	obs			
	c_1	c_2	c_3	e
t_1	1	1	0	1
t_2	0	1	1	1
t_3	1	0	0	1
t_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* [AZV07], que têm os coeficientes respetivamente calculados do seguinte modo

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

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

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

2.1.3 *Model-based diagnosis*

O princípio base do diagnóstico baseado no modelo (*Model-based diagnosís*) 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].

2.1.4 *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.

	obs			
	c_1	c_2	c_3	e
t_1	1	1	0	1
t_2	0	1	1	1
t_3	1	0	0	1
t_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.

2.1.4.1 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:

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

2.1.4.2 Ordenação de candidatos

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

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

$Pr(obs_i)$ 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

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

Sendo g_j (*component goodness*) a probabilidade do componente c_j executar de forma correta, temos que

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

Tendo em conta o nosso exemplo

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

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

Quando existem valores g_j desconhecidos, é maximizado o valor de $Pr(obs, e|d)$ usando o algoritmo *Maximum Likelyhood 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 $g_3 = 0.50$)

2.1.4.3 *Crowbar*

Crowbar[1], anteriormente conhecido como GZoltar[2], é a ferramenta que materializa o algoritmo *Barinel* e que permite uma análise de projetos *Java*.

Através de injeção de código e da execução dos testes *JUnit*, o *Crowbar* é capaz de identificar os componentes que foram executados e associá-los aos testes falhados.

A representação dos resultados pode ser feita de várias formas, sendo a principal a visualização *Sunburst* que podemos ver na Figura 2.1 e que apresenta em cada anel um grau de granularidade diferente, desde o projeto à linha de código.

[1] http://crowbar.io
[2] http://www.gzoltar.com/

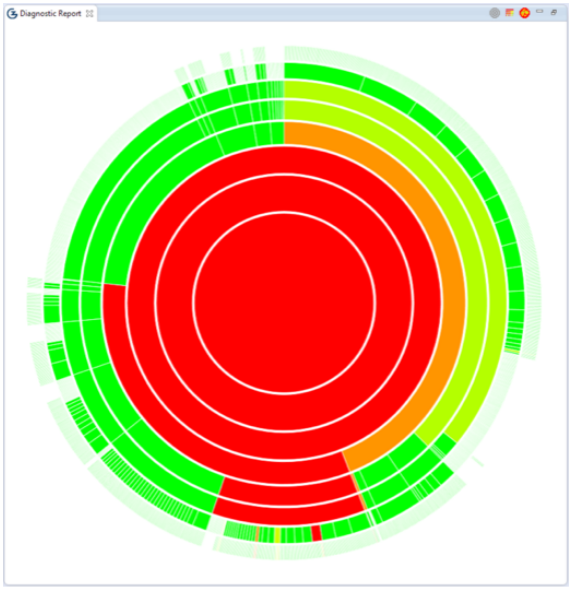

Figura 2.1: Visualização Sunburst do Crowbar

Para além desta informação, a ferramenta apresenta-nos também com a lista de componentes ordenada de acordo com a probabilidade de este ter falhas.

2.2 *Software Repository Mining*

2.2.1 Identificação de correções e defeitos

Na extração de informações de repositórios de código, como o *Git*, torna-se importante identificar o tipo de alterações feitas. A descrição textual associada à alteração é bastante importante nesta identificação e permite fazê-lo com um elevado grau de precisão, dependendo do projeto em causa [MV00].

É estimado que 34 a 46% de todas as alterações correspondam a correções e que estas representem 18 a 27% das linhas adicionadas e removidas [MV00]. Foram também encontrados padrões que relacionam as correções com o seu tamanho, em termos de código, e com o dia da semana em que foram executadas [ŚZZ05].

Após esta identificação é possível identificar as alterações que introduziram os erros usando o algoritmo *SZZ* [ŚZZ05]. Este algoritmo recorre ao histórico e à análise do código em questão para corretamente localizar o *commit* (alteração) que introduziu o defeito, ignorando, por exemplo, alterações que apenas introduzam comentários.

2.2.2 Ferramentas

2.2.2.1 *libgit2*

Libgit2[3] é uma biblioteca multi-plataforma, sem dependências, que permite a interação direta com o repositório *Git* com uma performance nativa.

Esta biblioteca pode ser usada em qualquer linguagem que permita ligação a *C*, pelo que existem dezenas de implementações em linguagem como *Python*[4], *Javascript*[5] e *Java*[6].

2.2.2.2 *GHTorrent*

GHTorrent[7] é um projeto que tem por objetivo criar um armazém de dados escalável, facilmente pesquisável, mesmo *offline*, que replique a informação obtida através da API REST do GitHub[8].

Esta ferramenta funciona de uma forma distribuída e monitoriza a API do GitHub através da página `https://api.github.com/events`. Cada evento, despoleta a extração de informação e conteúdo para uma base de dados *MongoDB* e outra *MySQL* [GS12].

2.3 Abordagens à predição de defeitos

Foram já exploradas algumas abordagens para a predição da qualidade do código. Salientam-se neste capítulo algumas técnicas como *BugCache*, *FixCache* e *Buggy Change Classification*, que são técnicas que se incluem no tipo de abordagem que pretendemos seguir (*change log*) e que permitem, em conjunto com outras técnicas, otimizar a localização dos defeitos.

2.3.1 *BugCache/FixCache*

Com base na premissa de que as falhas nunca ocorrem isoladamente e que portanto onde existe um defeito existem outros, o *BugCache* cria uma lista de componentes onde a probabilidade de conterem erros é elevada, com base na análise em todo o histórico de alterações do projeto.

Este algoritmo destaca-se pela sua precisão, conseguindo uma exatidão de 73 a 95% quando usado com uma granularidade ao nível do ficheiro, sendo portanto o melhor até à data [Kim06].

Os defeitos são identificados por ordem temporal e adicionados à lista. Quando esta lista atinge o tamanho máximo, os componentes vão sendo removidos de acordo com o método de substituição escolhido. Existem diversos métodos que utilizam diferentes métricas como última utilização (*Least Recent Used - LRU*), número de defeitos recentes e número de alterações recentes [Kim06].

Conclui-se com este algoritmo que [Kim06]:

[3] `https://libgit2.github.com/`
[4] pygit2 - `http://www.pygit2.org/`
[5] nodegit - `https://github.com/nodegit/nodegit`
[6] Jagged - `https://github.com/ethomson/jagged`
[7] `http://ghtorrent.org/`
[8] `http://developer.github.com/`

- Caso tenha sido introduzido um defeito, há uma tendência para serem introduzidos mais defeitos em breve. (*temporal locality*)
- Caso um componente tenha sido acrescentado ou modificado recentemente este tem uma maior probabilidade de ser defeituoso (*changed-entity locality*, *new-entity locality*)
- Caso um componente tenha introduzido um erro, os componentes ligados mais diretamente a este introduzirão também erros no futuro (*spatial locality*)

A diferença entre o *BugCache* e o *FixCache* é o momento em que cada um atualiza a lista. Sendo que o primeiro atualiza a lista quando um defeito é introduzido, o segundo atualiza apenas quando o erro é corrigido. Devido a esta diferença a implementação do *FixCache* torna-se mais fácil.

2.3.2 *Buggy Change Classification*

Change Classification tem uma abordagem fundamentalmente diferente e tem um objetivo também distinto. O *Change Classification*, com recursos a técnicas de *Machine Learning* e às informações disponíveis quanto a erros anteriores, é capaz de prever se uma alteração introduziu ou não um novo defeito. Esta previsão é feita com um rigor muito elevado de 78% [Whi08].

Numa primeira fase, todas as alterações feita até ao momento são classificados como *buggy* ou *clean*. Com esta classificação feita e com a extração de dados para cada também concluída, procede-se à criação de um modelo através do treino de um algoritmo de classificação.

O tipo de dados usados dividem-se em sete grupos: métricas de complexidade, código adicionado, código removido, nome do ficheiro e diretório, novo código e meta-dados [Whi08].

Tanto *Support Vector Machine* (SVM) como um classificador *Naïve Bayes* foram usados no estudo, tendo sido o classificador feito com recurso a SVM aquele que apresentou melhores resultados [Whi08].

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

Capítulo 4

Validação

Capítulo 5

Conclusões

Referências

[AZG09] Rui Abreu, P Zoeteweij 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 Zoeteweij 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.

[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.

[GS12] Georgios Gousios e D. Spinellis. GHTorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21. IEEE, jun 2012. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6224294, doi:10.1109/MSR.2012.6224294.

[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.

[Kim06] Sunghun Kim. Adaptive Bug Prediction by Analyzing Project History. page 145, 2006.

[MS08] Wolfgang Mayer e Markus Stumptner. Evaluating Models for Model-Based Debugging.pdf. pages 128–137, 2008.

[MV00] a. Mockus e L.G. Votta. Identifying reasons for software changes using historic databases. *ICSM conf.*, pages 120–130, 2000. doi:10.1109/ICSM.2000.883028.

[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.

[ŚZZ05] Jacek Śliwerski, Thomas Zimmermann e Andreas Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 30(4):1,

jul 2005. URL: http://dl.acm.org/citation.cfm?id=1082983.1083147, doi:10.1145/1082983.1083147.

[Wei81] Mark Weiser. Program slicing. pages 439–449, mar 1981. URL: http://dl.acm.org/citation.cfm?id=800078.802557.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982. doi:10.1145/358557.358577.

[Whi08] E.J. Whitehead. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, mar 2008. URL: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4408585, doi:10.1109/TSE.2007.70773.

[ZC09] Michael Zhivich e Robert K. Cunningham. The real cost of software errors. *IEEE Security and Privacy*, 7(2):87–90, 2009. doi:10.1109/MSP.2009.56.