Evitando Relatos de CRs duplicadas em Projetos Open Source de Software

Yguaratã C. Cavalcanti[1,2], Alexandre C. Martins[1,2], Eduardo S. Almeida[2], Silvio L. Meira[1,2]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)

[2]Centro de Estudos e Sistemas Avançados do Recife (C.E.S.A.R.)
Recife – PE – Brasil.

{ycc, acm2, srlm}@cin.ufpe.br, esa@rise.com.br

Abstract. *The management of CR (Change Request) repositories can become harder when the amount of users and developers increases. This situation can generates replications of the same defect report in different CRs and rises the costs associated with the time needed to development and maintenance. Thus, this paper presents an approach based on Text Mining to detect the duplicate CRs. With the proposed approach, was possible to detect duplicate CRs with precision of 17,61% and recall of 23,66% in the set of CRs used for tests.*

Resumo. *O gerenciamento de repositórios de CRs (Change Requests) pode se tornar custoso quando a quantidade de usuários e desenvolvedores aumenta. Essa situação pode gerar replicações dos mesmos defeitos relatados em CRs diferentes e, conseqüentemente, o aumento dos custos associados ao tempo de desenvolvimento e manutenção do software, por exemplo. Dessa forma, esse artigo apresenta uma abordagem baseada em Text Mining para detecção de relatos de CRs duplicadas. Com a técnica desenvolvida, foi possível detectar CRs duplicadas com precisão de 17,61% e cobertura de 23,66% no conjunto de CRs utilizado nos testes.*

1. Introdução

De acordo com Summerville [10], os projetos de software precisam passar por mudanças durante seu ciclo de vida para permanecerem utilizáveis. Essas mudanças podem ser correções de erros, adição ou modificação de funcionalidades, ou adaptação do sistema para outros ambientes de execução. Adicionalmente, essas mudanças no software devem ser executadas de acordo com as atividades de gerenciamento de mudanças, definidas no processo de desenvolvimento de software. Como mostra Erlikh [3], as mesmas podem corresponder até 90% do custo de desenvolvimento do software.

Com o objetivo de aprimorar o processo de mudanças, algumas organizações têm utilizado sistemas automatizados para gerenciar essa tarefa. Esses sistemas, geralmente denominados de *CR Tracking Systems* ou *Bug Trackers*, são responsáveis por gerenciar e armazenar as solicitações de mudanças submetidas para um artefato. Por solicitações de mudanças ou *change requests* (CRs), entende-se que é um artefato de software que

descreve algum defeito, melhoria, mudanças, ou alguma solicitação em geral, que é submetida para os sistemas gerenciadores de CRs.

Contudo, os sistema gerenciadores de CRs não trazem apenas benefícios, eles também introduzem novos problemas que precisam ser sanados. Um dos problemas mais críticos é a duplicação de relatos de CRs [1]. Esse problema é caracterizado pela submissão de dois ou mais relatos que descrevem a mesma solicitação de mudança no software. A principal consequência desse problema é a sobrecarga de retrabalho que essas CRs duplicadas provocam no momento em que o desenvolvedor está utilizando o sistema gerenciador de CRs.

Trabalhos recentes [1, 9] têm mostrado que, em média, de 10% a 30% das CRs de um repositório são compostos por CRs duplicadas. Sendo assim, com base no exemplo de Anvik et al. [1], suponha que um projeto de software receba em média 120 CRs por dia, e que um desenvolvedor gaste em média 15 minutos para analisar uma CR. Então seriam necessárias 30 horas, ou 30 pessoas-hora, para gerenciar essas CRs. Em média, de 3 a 9 horas seriam gastas na análise CRs duplicadas. O que ocasionaria perde de centenas de horas sem trazer nenhuma melhoria para o software.

O objetivo desse trabalho é aplicar uma solução baseada em busca por palavras-chaves e conceitos de *Text Mining* para evitar que CRs duplicadas sejam submetidas aos sistemas gerenciadores de CRs. Para tanto, é utilizado um conjunto de CRs do projeto Firefox, e desenvolvido um sistema de testes automatizado em Python [8] para validar a solução.

O restante desse artigo está organizado da seguinte maneira: na Seção 2 é apresentado o conjunto de CRs utilizado nos testes; a Seção 3 apresenta a solução baseada em *Text Mining*; na Seção 4 é apresentada a arquitetura da solução; a Seção 5 mostra os resultados obtidos; e por fim, os trabalhos relacionados e as conclusões são apresentados nas seções 6 e 7 respectivamente.

2. Conjunto de CRs Utilizado nos Testes

O conjunto de CRs utilizado nos testes é composto por CRs criadas no projeto Firefox [6] durante seu processo de desenvolvimento, evolução e manutenção. Mais especificamente, elas foram geradas entre os períodos de jan/2006 e jun/2006 totalizando 8,185 CRs. Também foi feita uma análise estatística do conjunto de CRs, utilizando *scripts* Python, onde pôde-se constatar que 32% do mesmo é composto por CRs duplicadas, e que 21% do total de CRs possuem uma ou mais CRs duplicadas para as mesmas.

3. Solução Baseada em *Text Mining*

Conforme foi descrito por Feldman e Sanger [4], sistemas de *Text Mining* lidam com documentos escritos em linguagem natural não estruturados. Portanto, dado que o conteúdo das CRs são descrições em linguagem natural, e sem nenhuma estrutura explícita, pode-se aplicar técnicas de *Text Mining* para evitar a duplicação de CRs. Para isso, basta considerar que os arquivos em XML para cada CR são documentos, e que quando um usuário do sistema gerenciador de CRs submete uma nova CR o sistema faz uma consulta utilizando as informações dessa CR, retornando as CRs mais similares para alertar ao mesmo se ele deve ou não submeter a nova CR.

Dessa forma, a solução baseada em *Text Mining* consiste em preparar os documentos da seguinte maneira: as CRs são recuperadas em formato XML; depois são realizadas operações de texto para eliminar caracteres e textos desnecessários, e aplicação de *stemming*[1]; em seguida os documentos são indexados em um arquivo de índice que mapeia as palavras encontradas no conjunto de CRs para as CRs que contêm as mesmas. Após esses passos, tem-se uma base de documentos pronta para receber consultas, as quais são formuladas a partir da nova CR que está sendo submetida.

4. Arquitetura do Sistema de Testes

O sistema desenvolvido tem o objetivo de testar, automaticamente, a técnica de busca por palavras-chaves juntamente com técnicas de *Text Mining* na procura de CRs duplicadas em repositórios de CRs. A arquitetura de tal sistema está dividida em 2 módulos principais e 5 módulos auxiliares, conforme será descrito a seguir.

4.1. Módulo Indexador

Na Figura 1, é representado o primeiro módulo principal, o qual é responsável pela indexação das CRs. Sendo assim, o funcionamento desse módulo segue a seguinte seqüência: primeiramente lê-se as CRs em formato XML de um determinado repositório, usando o módulo auxiliar *parsers*; depois é feito o pré-processamento dos dados extraídos dessas CRs através do módulo auxiliar *analysis*; e por fim, os documentos são indexados utilizando o módulo auxiliar PyLucene [11]. O papel de cada módulo auxiliar é descrito a seguir.

parsers. Na fase de leitura das CRs, utiliza-se o módulo auxiliar *parsers*, o qual possui *parsers* para diversas ferramentas de gerenciamento de CRs. No caso desse trabalho, é utilizado o *parser* para extrair as informações das CRs em arquivos XML exportados pelo Bugzilla.

analysis. Na fase de pré-processamento das CRs, o módulo auxiliar *analysis* é utilizado para eliminar caracteres e textos desnecessários com o auxílio de expressões regulares, assim como as *stop-words*[2] definidas para o idioma utilizado nas CRs e *stop-words* específicas de domínio. Para *stop-words* de idioma foi utilizada uma lista com 373 termos em Inglês, e para *stop-words* de domínio foi utilizada uma lista com 54 termos. Nesse mesmo módulo também é aplicada a técnica de *stemming* utilizando o algoritmo de Porter [7].

PyLucene. O módulo auxiliar *PyLucene* é uma extensão em Python para acessar o arcabouço Lucene [2] para buscas textuais . No módulo principal *indexer*, o PyLucene é utilizado para indexar os documentos utilizando a estrutura de índice invertido [4]. Além da indexação, o PyLucene também é utilizado para fazer buscas no índice. Nesse caso, o mesmo traz como resultado uma lista de CRs ordenadas por similaridade entre as mesmas e a *query* de busca usando a medida do cosseno [4] para tanto.

4.2. Módulo de Testes

Na Figura 2 é representado o segundo módulo principal do sistema. Esse módulo é responsável pelos testes automáticos que são feitos para avaliar a precisão e a cobertura

[1] *Stemming* é uma técnica para reduzir uma palavra ao seu radical.
[2] *Stop-words* são palavras freqüentemente utilizadas que não influenciam nas buscas.

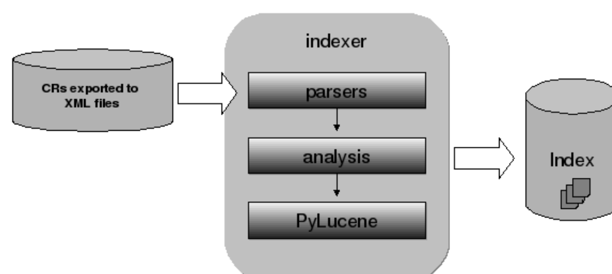

Figura 1. Arquitetura do sistema. Módulo indexador.

das buscas. A execução desse módulo toma como entrada o índice gerado pelo módulo indexador e as CRs em formato XML. A seqüência de execução é da seguinte forma: é montada uma representação em memória das CRs duplicadas e suas respectivas CRs originais; logo em seguida, uma CR aleatória de cada conjunto é selecionada para formar as diversas *queries* de busca do sistema; cada *query* é analisada pelo módulo auxiliar *analysis*, como no módulo indexador; e então usa-se o PyLucene para fazer as buscas no índice de palavras; por último, os resultados das buscas são analisados em termos de *precision*, *recall*, ASL (*Average Search Length*) e medida harmônica *F-measure*. O papel de cada módulo auxiliar é descrito a seguir, com exceção dos módulos já descritos anteriormente.

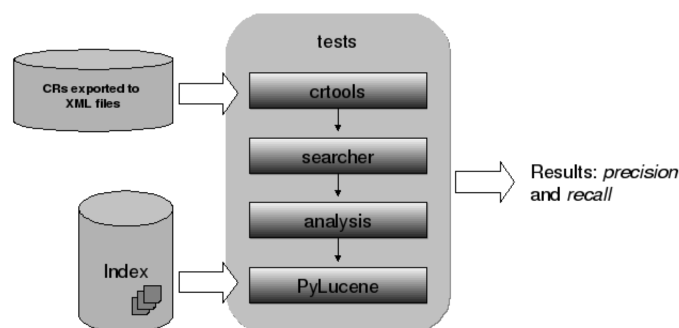

Figura 2. Arquitetura do sistema. Módulo de testes.

crtools. O módulo auxiliar *crtools* é responsável por montar a representação em memória das CRs e suas duplicatas. Essa representação é feita de tal forma que ao final se tenha um mapeamento das CRs originais para suas duplicatas, como mostrado na Figura 3. Com essa representação, o módulo percorre todo o mapeamento selecionando, aleatoriamente, uma CR de cada conjunto de duplicatas para montar a *query* de busca. A resultado da busca de cada *query* é então comparado com o conjunto original de CRs duplicadas, ao qual a CR da busca pertence, para calcular as métricas.

searcher. O módulo auxiliar *searcher* é responsável por montar a *query* de busca de acordo com os dados de cada CR selecionada, utilizando para isso o módulo *analysis* para fazer o pré-processamento do texto. Com a *query* pronta, o módulo faz a busca no índice de palavras utilizando o módulo PyLucene. Esse módulo retorna uma lista de CRs ordenadas por similaridade entre as mesmas e a CR de busca.

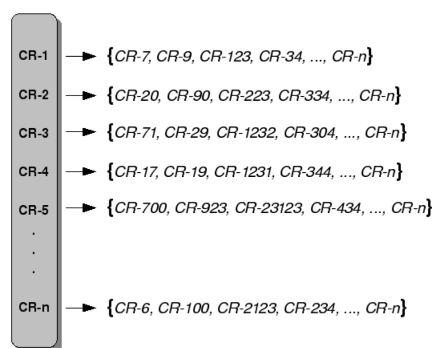

Figura 3. Representação em memória das CRs e suas duplicatas.

5. Resultados

A análise dos resultados foi feita utilizando as métricas de *precision*, *recall*, ASL e medida harmônica *F*. *Precision* mede a porcentagem de CRs corretamente recuperadas dentre todas as que foram recuperadas, já a medida *recall* mede a porcentagem de CRs que foram recuperadas dentre todas as CRs que deveriam ser recuperadas. A medida ASL calcula o posicionamento das CRs retornadas; valores próximos de zero mostram que as CRs duplicadas foram posicionadas no início da lista de resultados. Já a medida harmônica *F-measure* faz uma relação entre as medidas *precision* e *recall*.

Na Tabela 1 é mostrado os melhores resultados obtidos durante a seqüência de testes. Esses resultados são decorrentes de 2,619 buscas feitas no sistema, o que corresponde ao total de CRs duplicadas do conjunto de testes. Para obter esses resultados, adicionalmente foram selecionadas apenas as três palavras mais freqüentes da CR selecionada para compor a *query* de busca; *queries* com menos ou mais de três palavras obtiveram valores muito inferiores aos apresentados.

Tabela 1. Melhores resultados dos testes

$Precision$	$Recall$	$F-measure$	ASL
17.61%	23.66%	20.19%	5

Como se pode ver na tabela anterior, os valores para *recall* e *precision* não foram muito elevados. Uma justificativa para essa ocorrência, é o fato de que a técnica utilizada nesse trabalho não leva em consideração a semântica dos textos das CRs. Sendo assim, aplicando técnicas de semântica nas buscas pode-se elevar consideravelmente esses resultados, já que muitas CRs são escritas de formas distintas – por exemplo, utilizando sinônimos e/ou homônimos – para descrever o mesmo problema.

Apesar dos resultados baixos para *precision* e *recall*, os mesmos mostram-se satisfatórios, já que busca por palavras-chaves é uma técnica simples de ser implementada, quando comparada com as técnicas utilizadas nos trabalhos relacionados (Seção 6). Contudo, considerando o fato de que é mais importante retornar as CRs duplicadas no começo do resultado da busca, a medida ASL mostra que a técnica utilizada é eficiente boa para recomendar possíveis CRs duplicadas.

6. Trabalhos Relacionados

O problema de CR duplicadas foi primeiramente abordado por Anvik et al. [1], onde foi usado um modelo estatístico juntamente com técnicas de aprendizado de máquina para sugerir se uma determinada CR é duplicada ou não. Para comparar a similaridade entre duas CRs foi usada a medida do cosseno. A técnica também foi aplicada em CRs do projeto Firefox e conseguiu sugerir corretamente 28% das vezes.

O mesmo problema também foi atacado por Hiew [5], onde, mais uma vez, foi utilizada a técnica do cosseno para medir a similaridade entre duas CRs. O trabalho em questão foi testado em CRs dos projetos Firefox, Eclipse, Apache e Fedora, e obteve um máximo de *precision* e *recall* de 29% e 50% respectivamente.

Já no trabalho de Runeson et al. [9], a questão de CRs duplicadas foi tratada sob a ótica de processamento de linguagem natural, juntamente com técnicas de *Text Mining*. A técnica foi aplicada em CRs de uma organização privada, onde aproximadamente 10% das CRs era duplicadas. A técnica conseguiu evitar a submissão de CRs duplicadas em 40% das vezes.

7. Conclusão

Nesse trabalho foi apresentado o uso de uma técnica simples de buscas por palavras-chaves, juntamente com técnicas de *Text Mining*, para sugerir possíveis CRs duplicadas em sistemas gerenciadores de CRs. Apesar de não obter valores muito altos de *precision* e *recall*, a solução demonstrada é satisfatória pois, além de sua fácil implementação, o posicionamento das possíveis CRs duplicadas, medida através da métrica ASL, mostrou-se bastante eficaz.

References

[1] Anvik, J., Hiew, L., and Murphy, G. C. (2005). Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, New York, NY, USA. ACM Press. 1, 6

[2] Apache Software Foundation (2007). Lucene. Last access on fev/2008. 4.1

[3] Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional*, **2**(3), 17–23. 1

[4] Feldman, R. and Sanger, J. (2007). *The Text Mining Handbook: advanced approaches in analyzing unstructured data*. Cambridge University Press. 3, 4.1

[5] Hiew, L. (2006). *Assisted Detection of Duplicate Bug Reports*. Master's thesis, The University Of British Columbia. 6

[6] Mozilla (2005-2008). Firefox Web Browser. Último acesso em fev/2008. 2

[7] Porter, M. F. (1980). An algorithm for suffix stripping. *Program*, **3**(14), 130–137. 4.1

[8] Python Software Foundation (1990-2007). Python Programming Language. Último acesso em fev/2008. 1

[9] Runeson, P., Alexandersson, M., and Nyholm, O. (2007). Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 499–510, Los Alamitos, CA, USA. IEEE Computer Society. 1, 6

[10] Sommerville, I. (2007). *Software Engineering*. Addison Wesley, 8 edition. 1

[11] Vajda, A. (2005). Pulling java lucene into python: Pylucene. In *PyCon*. 4.1