ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de Sistemas Computacionais

Sceek: Um Ambiente Para Exploração de Código-Fonte

Trabalho de Conclusão de Curso

Engenharia da Computação

Bruno José de Moraes Melo
Orientador: Prof. Adriano Lorena Inácio de Oliveira

Recife, dezembro de 2007

IN PRIMIS SAPIENTIA
UNIVERSIDADE
DE PERNAMBUCO

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de Sistemas Computacionais

Sceek: Um Ambiente Para Exploração de Código-Fonte

Trabalho de Conclusão de Curso

Engenharia da Computação

Este Projeto é apresentado como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Bruno José de Moraes Melo
Orientador: Prof. Adriano Lorena Inácio de Oliveira

Recife, dezembro de 2007

IN PRIMIS SAPIENTIA
UNIVERSIDADE
DE PERNAMBUCO

Bruno José de Moraes Melo

Sceek: Um Ambiente Para Exploração de Código-Fonte

De omnibus dubitandum.
(Duvide de tudo.)

Resumo

Programadores estão freqüentemente trabalhando em sistemas ou com bibliotecas e *frameworks* que nunca viram antes. Este é um cenário típico durante a fase de manutenção, onde um desenvolvedor encara as tarefas de corrigir um *bug*, adicionar uma nova funcionalidade ou reescrever algum componente. Essas tarefas podem ter o desempenho melhorado através da efetiva compreensão dos códigos-fonte e do reuso de software, contudo como reusar algo o desenvolvedor não sabe que existe e como descobrir software reutilizável dentro da própria organização. A ferramenta Sceek foi desenvolvida com o propósito de ajudar o desenvolvedor durante essas atividades fornecendo uma interface intuitiva e de fácil navegação pelos códigos-fonte de qualquer projeto de software escrito em Java. A navegação é feita através de categorias, ou facetas, obtidas das estruturas e semânticas do código. Foram realizados experimentos com métodos de aprendizado de máquina para a criação de algumas categorias semânticas. Os resultados demonstram a adequação desses métodos, em especial *Support Vector Machine* (SVM), na categorização de código-fonte.

Abstract

Software developers are often working on a system or with libraries and frameworks they have not seen before. This is a typical scenario in the maintenance phase, where a developer faces the task to fix a bug, add a feature, or rewrite some components. These tasks can be improved by effective code comprehension and by reuse, however, how to reuse something if the developer doesn't know that exists and how they discover reusable software made "in house". The Sceek tool was developed to support the programmers at these tasks using an intuitive interface which the source code can be easily browsed. The navigation through large amount of source code is category, or faceted, based. The information about these facets is extracted from the code structure and semantics. Experiments with machine learning techniques were used to create some semantic facets. The results show the feasibility of these methods, in special Support Vector Machine (SVM), to classify source code.

Sumário

Índice de Figuras vi

Índice de Tabelas vii

Tabela de Símbolos e Siglas viii

1 Introdução 10

1.1 Motivação 10
1.2 Objetivos 11
1.3 Estrutura do Documento 11

2 Tópicos em Engenharia de Software 13

2.1 Reuso de Software 13
2.1.1 Desenvolvimento Baseado em Componentes 15
2.1.2 Repositório de Reuso 15
2.1.3 Desenvolvimento Baseado em *Frameworks* 16
2.2 Compreensão de Código 17

3 Tópicos em Aprendizado de Máquina 18

3.1 Aprendizado Supervisionado 18
3.1.1 Support Vector Machine 19
3.1.2 Naive Bayes 20
3.2 Aprendizado Não-Supervisionado 20
3.2.1 K-means 21

4 Sceek: Um Ambiente para Exploração de Código-Fonte 23

4.1 Trabalhos Relacionados 23
4.2 Componentes 26
4.2.1 Extrator de Informação 27
4.2.2 Classificador 28
4.2.3 Interface 29

5 Experimentos 32

5.1 Preparação dos Experimentos 32
5.2 Experimentos com Aprendizagem Supervisionada 33
5.3 Experimentos com Aprendizagem Não-Supervisionada 35
5.4 Análise dos Resultados 37

6 Conclusão 38

6.1 Contribuições 38
6.2 Dificuldades Encontradas 39
6.3 Trabalhos Futuros 39

Índice de Figuras

Figura 1. Exemplo ilustrativo de um problema de classificação 18
Figura 2. Exemplo ilustrativo de um problema de *clustering* 20
Figura 3. Exemplo de visualização do Relo 23
Figura 4. Exemplo de busca com o Sourcerer 24
Figura 5. Organização dos componentes no Sceek 25
Figura 6. Exemplo de XML gerado pelo Java2XML 26
Figura 7. Representação de código como um vetor de palavras 27
Figura 8. Página inicial do Sceek 29
Figura 9. Exibição de um resultado com a possibilidade de encontrar resultados semelhantes 30

Índice de Tabelas

Tabela 1. Categorias baseadas nas estruturas do código Java 27
Tabela 2. Categorias conceituais usadas no Sceek 28
Tabela 3. Distribuição das categorias em cada classe 31
Tabela 4. Taxa de acertos dos classificadores 32
Tabela 5. Medidas de avaliação do classificador *Naive Bayes* na categoria *XML* 32
Tabela 6. Medidas de avaliação do classificador *Naive Bayes* na categoria *UI* 33
Tabela 7. Medidas de avaliação do classificador *Naive Bayes* na categoria *Text Processing* 33
Tabela 8. Medidas de avaliação do classificador *Naive Bayes* na categoria *System* 33
Tabela 9. Medidas de avaliação do classificador SVM na categoria *XML* 33
Tabela 10. Medidas de avaliação do classificador SVM na categoria *UI* 33
Tabela 11. Medidas de avaliação do classificador SVM na categoria *Text Processing* 33
Tabela 12. Medidas de avaliação do classificador SVM na categoria *System* 33
Tabela 13. Medidas de avaliação do classificador Naive Bayes na categoria Networking 34
Tabela 14. Medidas de avaliação do classificador Naive Bayes na categoria IO 34
Tabela 15. Medidas de avaliação do classificador Naive Bayes na categoria Application 34
Tabela 16. Medidas de avaliação do classificador SVM na categoria *Networking* 34
Tabela 17. Medidas de avaliação do classificador SVM na categoria *IO* 34
Tabela 18. Medidas de avaliação do classificador SVM na categoria *Application* 34
Tabela 19. Resultados da execução do *K-means* com K = 8 35
Tabela 20. Resultados da execução do *K-means* com K = 7 35
Tabela 21. Resultados da execução do *K-means* com K = 6 35

Tabela de Símbolos e Siglas

OSS – *Open Source Software* (Software Livre)
NIH - *Not Invented Here* (Não Inventado Aqui)
HTTP – *HyperText Transfer Protocol* (Protocolo de Transferência de Hipertexto)
XML – *eXtensible Markup Language* (Linguagem de Marcação Extensível)
OSGi – *Open Services Gateway initiative*
UML – *Unified Modelling Language*

Agradecimentos

Muitas pessoas contribuíram durante esses cinco anos de muito trabalho. É complicado lembrar-se de todas nesta simples página, mas não posso deixar de citar as que estiveram mais próximas em todos os momentos tristes e felizes.

Primeiramente agradeço à providência divina com todas as suas intervenções, ajudando-me durante as escolhas mais difíceis e trazendo paz nos dias tristes.

Meus pais por terem, de forma particular, me ensinado sobre caráter e dignidade nos primeiros anos da minha vida. À minha mãe, em especial, por ser um exemplo de luta e vitória.

À minha esposa, Amanda, por me acompanhar em todos os momentos da minha vida, conseguindo agüentar meu mau humor, minhas piadas sem graça e minha paixão pela computação.

Aos professores, todos de alguma forma contribuíram para minha formação e influenciaram diretamente nas minhas escolhas. Em especial agradeço ao Prof. Adriano Lorena pela orientação durante quase todo o curso.

Aos grandes amigos que fiz na faculdade, todos foram essenciais para poder superar os desafios que vieram.

Por último, mas não menos importante, aos amigos do Ministério Público de Pernambuco, pelas as boas conversas e conselhos sobre quase todos os assuntos.

Capítulo 1

Introdução

"Software is hard."
-- Donald Knuth

A atividade de desenvolvimento de software é, em certas condições, extremamente complexa devido à grande quantidade de fatores que influenciam todo o processo. Esse trabalho aborda alguns desses fatores com o objetivo de reduzir a complexidade em algumas atividades. Nesse capítulo são explicitados a motivação e os objetivos desse trabalho e apresentado, de forma resumida, a organização dos capítulos seguintes.

1.1 Motivação

Mudança é um fator inerente a software. Seja desenvolvendo um novo programa ou reconstruindo um antigo, o fluxo de alterações é constante. Manutenção de software é responsável por uma considerável fatia nos custos de desenvolvimento de software – mais de 50% em média – constituindo-se uma das atividades mais caras desse processo [1, 2]. Portanto, qualquer solução que melhore a produtividade de manutenção terá um grande impacto nos custos e lucros das empresas.

A compreensão de código é um dos fatores mais relevante para uma efetiva manutenção e evolução de software. Por décadas pesquisadores tentam entender como programadores compreendem programas durante essas atividades [3, 4]. Esse processo se torna mais difícil quando os sistemas são de grande porte com vários desenvolvedores, módulos e camadas diferentes.

No contexto de software livre, ou *open source software* (OSS), esses processos são dificultados pela natureza distribuída dos projetos e facilitados pela motivação dos desenvolvedores no aprendizado de novas tecnologias [5]. Isso implica a necessidade de mais uma habilidade do desenvolvedor: Leitura de código [6]. Normalmente, esses desenvolvedores sentem a necessidade de contribuir com novas funcionalidades ou correção de erros por utilizarem com freqüência um determinado projeto. Porém, a barreira para entender todo projeto ou uma parte especifica onde deseja contribuir desanimam os desenvolvedores menos experientes. No mundo corporativo, a má execução dessas atividades provoca prejuízos financeiros e danos à imagem das instituições.

Durante a fase de construção e evolução do software alguns artefatos podem ser reusados trazendo benefícios como redução no tempo de desenvolvimento, qualidade final do produto e complexidade reduzida [7]. Porém, para uma efetiva solução desses problemas são necessárias ferramentas que, no caso de reuso, faça a busca por artefatos para serem reusados e, na fase de manutenção, auxilie na compreensão do projeto.

1.2 Objetivos

Os principais objetivos dessa monografia são avaliar a utilização de técnicas de aprendizado de máquina para fazer busca e exploração de código-fonte e criar um protótipo onde essas atividades sejam feitas com facilidade. Para alcançar esses objetivos foram estabelecidas as seguintes metas:

- Investigar a aplicação de técnicas de aprendizado de máquina para classificar e agrupar código-fonte em determinadas categorias.
- Criar as bases de dados para os experimentos com aprendizado de máquina utilizando um projeto *open source* de grande utilização pela comunidade de Java.
- Integrar uma interface gráfica com suporte a navegação por categorias. Algumas categorias serão atribuídas de forma automatizada pelas técnicas de aprendizado de máquina.

Como resultado dessas atividades o sistema desenvolvido nesse trabalho, chamado de Sceek – *Source Code Exploration Environment*, possibilitará a navegação de um projeto de software desenvolvido em Java através de categorias baseadas nas estruturas dos códigos e classificadas utilizando a técnica de aprendizado de máquina que obteve o melhor desempenho.

A navegação desse espaço de informação será feita utilizando uma interface baseada no projeto *Flamenco* [8]. O *Flamenco* é uma interface *web* para a navegação de grandes coleções de itens como documentos ou imagens utilizando categorias de forma que possibilite o refino e a expansão das consultas.

1.3 Estrutura do Documento

Este trabalho está organizado em 6 (seis) capítulos. Nesse capítulo a motivação e os objetivos do trabalho foram apresentados. Os próximos capítulos estão organizados da seguinte forma:

- No **Capítulo 2** são introduzidos os conceitos básicos de engenharia de software, mais precisamente relativos a reuso de software e compreensão de código, necessários para um completo entendimento das motivações desse trabalho.
- O **Capítulo 3** descreve as técnicas de aprendizado de máquina utilizadas nesse trabalho. *Support Vector Machines* e *Naive-Bayes* são as duas técnicas de aprendizado supervisionado, e *K-means* a técnica de aprendizado não-supervisionado detalhadas na seção.
- O **Capítulo 4** descreve detalhadamente o desenvolvimento do Sceek, apresentando sua organização, componentes e funcionamento.

- No **Capítulo 5** são descritos a criação das bases de dados utilizadas e os experimentos, e são apresentados os resultados comparativos entre as duas técnicas de aprendizado supervisionado aplicadas em um software *open source*.

- Por fim, o **Capítulo 6** conclui o trabalho e discute as principais contribuições, melhorias e trabalhos futuros.

Capítulo 2

Tópicos em Engenharia de Software

"Programs must be written for people to read,
and only incidentally for machines to execute."
-- Abelson & Sussman,
Structure and Interpretation of Computer Programs

Observando o dia-a-dia da nossa civilização nota-se a dependência dos sistemas baseados em software. Cada vez mais os produtos incorporam de algum modo, computadores e software durante seu desenvolvimento. Nesses sistemas, o software representa um grande percentual dos custos do sistema [1].

A engenharia de software é uma disciplina de engenharia, cuja principal meta é o desenvolvimento de sistemas de software com boa relação custo-benefício. Sua responsabilidade cobre todos os aspectos da produção de software, desde os estágios de especificação do sistema até a manutenção.

Adiante serão brevemente revisados os dois assuntos de engenharia de software de maior relevância para essa monografia: Reuso de Software e Compreensão de Código.

2.1 Reuso de Software

A conferência de engenharia de software da OTAN de 1968 é geralmente considerada o berço da engenharia de software. O foco da conferência foi a *crise de software* – o problema de desenvolver software de grande porte e confiável em um ambiente controlado de forma economicamente efetiva.

No começo, reuso foi pensado como uma forma de solucionar a crise de software. O primeiro artigo sobre reuso foi apresentado por McIlroy nessa conferência com o título: *Mass Produced Software Components* [7]. Esse trabalho propôs a criação de uma biblioteca de componentes reusáveis e uma série de técnicas para automatizar a personalização dos componentes em diferentes graus de precisão e robustez. A idéia era fazer em engenharia de software o que já existia em outras engenharias, onde a especificação de um projeto se baseia em componentes que já foram experimentados e testados em outros sistemas.

Reuso de software pode ser definido como o processo de desenvolver software a partir de artefatos já existentes. Os tipos de artefatos não são limitados ao código-fonte, mas podem incluir especificações e documentos que são utilizados durante as várias fases do processo de desenvolvimento. Uma vantagem evidente do reuso é a redução dos custos, pois menos componentes precisam ser desenvolvidos, basta apenas serem re-utilizados. Portanto, a utilização de reuso de forma sistêmica implica em diversas vantagens que, resumidamente, são apresentadas abaixo:

Qualidade – As correções de erros são acumuladas a cada reuso. Isso implica em componentes de alta qualidade, diferente dos componentes que são desenvolvidos e usados apenas uma vez.

Produtividade – Um ganho na produtividade é alcançado devido à menor quantidade de código que deve ser escrito resultando em menos esforços para testes, analise e projeto. Quando uma política de reuso está sendo implantada há uma provável perda de produtividade causada pelo crescente esforço de aprendizagem e a necessidade de desenvolver componentes reusáveis. Essa perda é temporária e facilmente compesanda pelo aumento em longo prazo.

Confiança – A utilização de componentes testados aumenta a confiança do sistema. Além disso, o uso de um componente em vários sistemas aumenta a chance de encontrar erros durante a fase de desenvolvimento, isto é, o componente é refinado com sua implantação em diversos locais.

Time to market – O sucesso ou fracasso de um produto de software é freqüentemente determinado pelo seu *time to market*. Utilizando componentes reusáveis observa-se uma considerável redução desse tempo. Algo em torno de 42% em alguns projetos [9], isto é, um projeto com cronograma de dez meses poderia ter seu prazo reduzido para algo em torno de seis meses através da utilização de reuso.

Contudo, existem alguns obstáculos que inibem a utilização de reuso de forma generalizada. Vários fatores influenciam direta ou indiretamente no sucesso ou fracasso do reuso. Esses fatores podem ser de natureza conceitual, técnica, gerencial, organizacional, psicológica, econômica ou legal. Alguns obstáculos são apresentados abaixo:

Fatores gerenciais – O grande custo associado à atividade de desenvolvimento para reuso sempre envolve decisões dos diretores gerais da organização. Além disso, o gerenciamento de projetos com reuso é mais complexo e impactam em todo o ciclo de vida do software.

***Not invented here* (NIH)** – Muitos desenvolvedores sentem-se impedidos de expressar sua criatividade utilizando software desenvolvido por outros. Eles preferem desenvolver o software "do zero" a manter software desenvolvido por outrem. Esse tipo de atitude é chamado de síndrome do *not-invented-here* [10].

Dificuldade de encontrar software reutilizável – Software não pode ser reusado ao menos que possa ser encontrado. Reuso é improvável quando há um repositório com informações insuficientes ou quando os componentes são mal classificados. Por exemplo,

se os componentes de uma organização são armazenados de forma *ad hoc*, como um desenvolvedor pode encontrar um componente que envie um arquivo utilizando o protolo HTTP ou um componente que gere um XML representando um objeto Java.

A implantação de uma política de reuso pode ser feita de várias maneiras, variando de acordo com as características de cada organização como o tamanho do time de desenvolvimento e o domínio das aplicações. Nosso trabalho foca em duas abordagens de desenvolvimento para e com reuso: desenvolvimento baseado em componentes e desenvolvimento baseado em *framework*. Além dessas abordagens, são discutidas questões relacionadas ao armazenamento e busca de componentes. Esses tópicos são detalhados nas próximas subseções.

2.1.1 Desenvolvimento Baseado em Componentes

O desenvolvimento baseado em componentes teve seu primeiro passo, assim como a área de reuso, com o trabalho de McIlroy [7] onde ele sugeria a criação de uma indústria de componentes de vários tipos e características. Por componente entende-se qualquer parte do software que é identificável e reutilizável (por exemplo, funções e classes) ou entidades com um nível maior de abstração, por exemplo, um *bundle* na tecnologia OSGi [11]. O OSGi é um sistema de módulos em Java, cuja principal funcionalidade consiste em um *framework* baseado em componentes com registro de serviços e um ciclo de vida próprio, chamados de *bundles*. O OSGi é usado como base do ambiente de desenvolvimento Eclipse [12].

Componentes reusáveis podem ser definidos, mais precisamente, como artefatos claramente identificáveis que descrevem e/ou executam funções especificas e possuem interfaces claras, com documentação e status definido de reuso.

No desenvolvimento orientado ao reuso, os requisitos de sistema são modificados de acordo com os componentes reutilizáveis disponíveis. O projeto também se baseia nos componentes existentes. Naturalmente, isso significa que é possível que haja uma conciliação de requisitos. O projeto pode ser menos eficiente do que um projeto de propósito especial. Contudo, os menores custos de desenvolvimento, a entrega mais rápida do sistema e o aumento da confiabilidade do sistema compensam esse aspecto.

2.1.2 Repositório de Reuso

Em qualquer processo de desenvolvimento de software baseado em reuso é possível distinguir pelo menos duas atividades: desenvolvimento *com* reuso e desenvolvimento *para* reuso. Portanto, nesse contexto, são necessários recursos para a produção e publicação dos componentes (desenvolvimento *para* reuso) e desenvolvedores concentrados na busca e uso dos componentes (desenvolvimento *com* reuso). Dessa forma, é essencial a construção de um repositório não apenas para o armazenamento dos componentes, mas também como uma ferramenta para a publicação e consumo.

Um artefato reutilizável pode ser definido como qualquer trabalho com possibilidade de ser utilizado em mais de um projeto. Os artefatos abaixo são alguns exemplos de artefatos reusáveis:

- <u>Código Compilado</u>: objetos executáveis;
- <u>Código-Fonte</u>: classes e métodos;

- Casos de Teste;
- Modelos e Projetos: *framework*, padrões;

Portanto, os elementos suportados por um repositório não precisam ser limitados a apenas um tipo de artefato.

Na construção de um repositório de reuso é preciso levar em consideração vários aspectos, muitos dependentes das reais necessidades de cada organização. Nesta seção são levados em consideração apenas dois requisitos que representam a base desse trabalho: Busca e Navegação em repositórios de reuso.

Busca

O repositório deve fornecer mecanismos de busca que permitam aos usuários encontrar artefatos que satisfaçam suas necessidades. A busca pode ser feita utilizando texto livre, onde o usuário fornece um conjunto de palavras e o mecanismo faz o casamento com os fragmentos dos artefatos, ou através de facetas [13, 14], onde cada artefato é associado a categorias (por exemplo, linguagem de programação, domínio, etc.) possibilitando assim a busca por artefatos de categorias especificas.

A atividade de recuperação de um componente funciona da seguinte maneira, como descrito em [15]: Quando o desenvolvedor encara um problema, ele o entende de sua própria maneira e elabora uma consulta. Na prática, nesse primeiro momento há perda de informação, pois o desenvolvedor nem sempre é capaz de entender exatamente qual é o problema ou codificá-lo na linguagem de consulta. Para ser recuperada, a informação sobre o componente deve ser codificada. Esse processo, chamado de indexação, resulta em perda de informação. Portanto, o processo de busca consiste simplesmente em comparar a consulta com os índices e retornar os que casam.

Navegação

A navegação possibilita uma visão completa dos artefatos armazenados em um repositório. Os artefatos podem ser agrupados em diferentes categorias possibilitando aos usuários navegar pelos artefatos e descobrir relações através das categorias.

2.1.3 Desenvolvimento Baseado em *Frameworks*

Um dos mais importantes tipos de reuso é o de projetos de software. Uma coleção de classes pode ser usada para expressar um projeto abstrato. O projeto de um software é normalmente descrito em termos dos componentes do software e como eles interagem. Um compilador, por exemplo, pode ser descrito como consistindo de um analisador léxico, um *parser*, uma tabela de símbolos e um gerador de código.

Um projeto abstrato e orientado a objeto, conhecido como *framework*, consiste em classes abstratas para os principais componentes. Os *frameworks* são um projeto de subsistema constituído de um conjunto de classes abstratas e concretas e da interface entre elas. Detalhes específicos do subsistema de aplicações são implementados com o acréscimo de componentes e o fornecimento da implementação concreta das classes abstratas. Os *frameworks* raramente são aplicações propriamente ditas. As aplicações normalmente são construídas pela integração de diversos *frameworks*.

Frameworks possibilitam a reutilização de código e conhecimento de uma forma particular. Componentes independentes podem ser reusados rapidamente. Porém, componentes complexos, agregados a lógica de determinada aplicação podem ser usados como exemplos para compreender o *framework* por completo [16].

A ferramenta Sceek é um exemplo da utilização de *framework*. A interface do sistema foi desenvolvida utilizando o Flamenco que é um *framework* de interface de busca baseado em categorias. Adicionalmente, o Flamenco utiliza um *framework* para criação de aplicações *web*, o Webware [17].

2.2 Compreensão de Código

As pesquisas em compreensão de código [3, 18, 19, 20] são motivadas pela noção de que se for mais fácil para desenvolvedores responsáveis pela manutenção de grandes sistemas entenderem o código, será mais fácil fazer modificações nesse código. Um modelo dos processos cognitivos e dos hábitos de trabalho dos desenvolvedores pode ser usado para projetar ferramentas que auxiliem a compreensão do código. Uma ferramenta efetiva de compreensão de código traz uma série de benefícios para os desenvolvedores, principalmente os que trabalham em grandes sistemas. Menos erros no processo de compreensão resultam em rápidas modificações e código de qualidade. A combinação de esforço reduzido e a qualidade adquirida implicam em menos custo com manutenção.

Código-fonte tem uma estrutura que dificulta sua leitura de forma linear, e esse problema é agravado em grandes sistemas quando é impossível ler todo o código. Conseqüentemente, o desenvolvedor deve ler seletivamente, tornando a tarefa mais difícil e suscetível a erros. Atualmente, muitos desenvolvedores fazem uso de ferramentas de busca de propósito geral, como editores de texto e utilitários do sistema operacional (ex. *grep*). Um dos propósitos desse trabalho é integrar busca como um componente nas tarefas de manutenção de software, isto é, possibilitar que durante a criação de uma nova funcionalidade ou a correção de um *bug* o desenvolvedor facilmente encontre um componente para reusar ou um exemplo de funcionalidade para entender algum código.

Capítulo 3

Tópicos em Aprendizado de Máquina

"Intelligence is what you use when you don't know what to do."
-- Jean Piaget

Aprendizado de Máquina é um campo da ciência que procura responder a seguinte pergunta: "Como podemos desenvolver sistemas computacionais capazes de automaticamente aprender e evoluir com a experiência?". Essa questão envolve várias tarefas de aprendizagem, como a mineração de dados médicos para aprender quais futuros pacientes irão responder melhor a quais tratamentos ou como um engenho de busca pode categorizar os códigos de um software para ajudar na navegação do desenvolvedor. Para ser mais preciso, é dito que uma máquina aprende com respeito a uma tarefa T, uma métrica de desempenho D e um tipo de experiência E, se um sistema melhora seu desempenho D na tarefa T seguindo a experiência E [21]. Dependendo de como T, D e E são definidos, a tarefa de aprendizagem pode ser chamada de mineração de dados, visão computacional, etc.

Esse capítulo aborda as técnicas usadas no sistema desenvolvido nesse trabalho. As técnicas foram organizadas pelos tipos de aprendizado que elas suportam: aprendizado supervisionado, com as técnicas de classificação *Support Vector Machine* (SVM) [22] e *Naive Bayes* [21], e não-supervisionado, com a técnica de agrupamento *K-means* [21].

3.1 Aprendizado Supervisionado

Os algoritmos que utilizam a aprendizagem supervisionada funcionam da seguinte maneira: um conjunto de exemplos de treinamento é apresentado ao algoritmo, normalmente esse algoritmo é responsável por resolver um problema de classificação, isto é, dados exemplos de várias classes o algoritmo deve encontrar uma função que separe essas classes. O propósito dessa técnica é possibilitar à função encontrada classificar exemplos que não foram visto durante o treinamento, ou seja, a função ser capaz de generalizar.

Um exemplo de propósito didático é ilustrado na Figura 1. O conjunto de dados nesse exemplo contém apenas duas classes: *triângulos* e *quadrados*. Resumidamente, o objetivo de um classificador é encontrar uma função nesse caso, uma função linear, por ser este um problema linearmente separável, que delimite a região de cada classe. Esse classificador é chamado de *binário* por tratar de duas classes apenas.

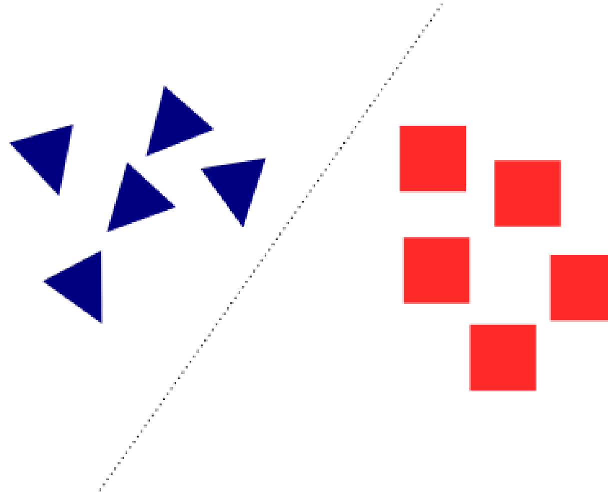

Figura 1. Exemplo ilustrativo de um problema de classificação.

3.1.1 Support Vector Machine

Máquina de Vetor de Suporte (*Support Vector Machine* – SVM) é uma técnica recente de classificação e regressão desenvolvida por Vapnik [22] baseado no principio da minimização do risco estrutural (*Strucutural Risk Minimization* – SRM) [23], cujo estabelece que para obter um bom desempenho de generalização, um algoritmo de aprendizado de máquina deve tentar minimizar o risco estrutural ao invés do risco empírico. O risco empírico é o erro no conjunto de treinamento, enquanto o risco estrutural considera tanto o erro no conjunto de treinamento quanto à complexidade das classes de funções ajustadas aos dados.

A principal tarefa no SVM é a construção de um hiperplano ótimo, isto é, hiperplanos que maximizam a margem de separação das classes, possibilitando a separação dos padrões de treinamento de diferentes classes. Um classificador SVM minimiza a Equação 3.1 baseada na condição especificada na Equação 3.2.

$$\min_{w,b,\xi} \frac{1}{2} w^T w + C \sum_{i=1}^{l} \xi_i \quad (3.1)$$

$$y_i(w^T \phi(x_i) + b) \geq 1 - \xi_i,\ \xi_i \geq 0 \quad (3.2)$$

Os padrões de treinamento são mapeados em um espaço de alta dimensão pela função . O classificador SVM encontra um hiperplano que consegue separar o espaço linearmente com a máxima margem nesse espaço de alta dimensão. Esse mapeamento é feito utilizando uma função *kernel*, $K(\vec{x},\vec{y}) = \phi^T($. Nesse trabalho foi utilizada a função de base radial, ou *radial basis function* (RBF), que dada por $K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$.

Máquina de Vetor de Suporte tem conseguido uma notável precisão em diversos problemas [24, 25, 26], aplicações como identificação facial [27], categorização de documentos [28] e classificação de projetos de software.

3.1.2 Naive Bayes

O classificador *Naive Bayes* (NB) é baseado no modelo probabilístico do teorema de Bayes. O teorema de Bayes relaciona as probabilidades condicionais e marginais dos eventos C e A:

$$Pr(C|A) = \frac{Pr(C)\, Pr(A|C)}{Pr(A)} \tag{3.3}$$

Onde $Pr(C)$ e $Pr(A)$ são as probabilidades dos eventos C e A, respectivamente. No contexto do classificador NB, o evento C representa a categoria de um determinado padrão, enquanto A representa o conjunto de atributos dos padrões. Na prática, pode-se desconsiderar o denominador dessa equação já que não depende de C e os valores do conjunto de A são conhecidos, ou seja, o denominador é uma constante.

Após varias aplicações recursivas do teorema de Bayes no numerador da equação anterior, e considerando os valores de A independentes uns dos outros, a Equação 3.3 pode ser reescrita como:

$$Pr(C|A_{1,\dots,}A_n) = Pr(C)\prod_{i=1}^{n} Pr\,(A_i|C) \tag{3.4}$$

Resumidamente, o classificador *Naive Bayes* consiste em encontrar as probabilidades dos atributos com relação a cada classe objetivando a seguinte equação:

$$classificador(a_{1,\dots,}a_n) = \text{argmax}_c\; Pr(C = c)\prod_{i=1}^{n} Pr\,(A_i = a_i|C = c) \tag{3.5}$$

O *Naive Bayes* é bastante utilizando em problemas de classificação de *e-mail*, categorizando entre *spam* e não-*spam* [29], classificação de textos em tópicos e vários problemas que envolvem uma grande quantidade de documentos com categorias independentes.

3.2 Aprendizado Não-Supervisionado

Na aprendizagem não-supervisionada, também chamada de *clustering* ou agrupamento, os dados não contem informações sobre qual classe cada exemplo pertence. A tarefa de *clustering* consiste em agrupar os padrões baseado em alguma medida de semelhança. A medida mais freqüentemente utilizada é a distância Euclidiana. A Figura 2 exibe um exemplo de agrupamento. Nesse caso, as figuras geométricas mais semelhantes são agrupadas próximas uma das outras no espaço de características, nesse exemplo o espaço é bidimensional.

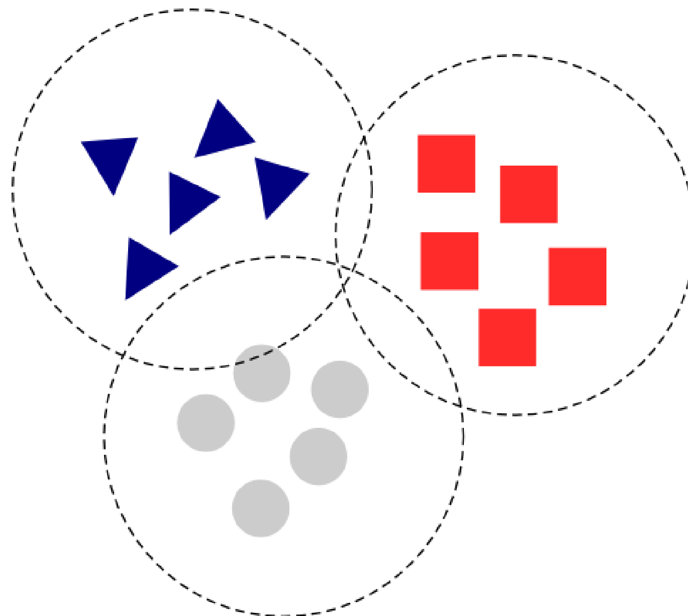

Figura 2. Exemplo ilustrativo de um problema de *clustering*.

3.2.1 K-means

O *K-means* é um dos algoritmos mais simples e efetivos para agrupamento de dados. O objetivo do *K-means* é dividir os padrões de treinamento em *K clusters* de forma que alguma relativa ao centro dos clusters seja minimizada. Conhecido o valor de *K*, o algoritmo pode ser descrito da seguinte maneira:

1. Escolha os centros iniciais dos *K clusters* , $m_2(0), \dots,$ onde representa o centro do *cluster* número *i* na iteração *l*. Para a primeira iteração, os valores inicias para os centros dos *clusters* são escolhidos aleatoriamente.

2. Atribua padrões desconhecidos para os *clusters*. Para cada padrão calcule a distância Euclidiana para o *cluster* e atribua o padrão que tem a menor distância. Portanto, um padrão é atribuído para a classe se:

$$\left\|m_j(k) - x_i\right\| < \|m_i(k) - x_i\| \; para \;\; i = 1,2,\dots,k, i \neq j \tag{3.6}$$

onde:

$$\|x\| = \left[\sum_{i=1}^{n} x_i^2\right]^{\frac{1}{2}} \tag{3.7}$$

3. Calcule os centros dos novos *clusters*. Usando os novos conjuntos de classes (ou agrupamentos) do *cluster* recalcule o valor de cada cento como:

$$m_j(k+1) = \frac{1}{n} \sum_{x \in \omega_j} x_i \tag{3.8}$$

onde *n* é o número de padrões no *cluster j*.

4. Verificação da convergência. A condição de convergência é que nenhum *cluster* tenha mudado o valor do centro no passo anterior. Essa condição pode ser expressa como:

$$m_j(k+1) = m_j(k) \; para \; j = 1,2,\dots,k \tag{3.9}$$

Se a condição da equação anterior é satisfeita, então o algoritmo convergiu, caso contrário o algoritmo deve continuar com o passo 2. O número de *clusters*, a escolha dos centros iniciais e a ordem na qual os padrões são inseridos influenciam no comportamento do algoritmo.

Capítulo 4

Sceek: Um Ambiente para Exploração de Código-Fonte

A ferramenta desenvolvida nesse trabalho, chamada de Sceek – *Source-Code Exploration Environment* – tem como propósito o apoio aos desenvolvedores durante a fase de desenvolvimento, na tarefa de busca por artefatos para serem reusados bem como na fase de manutenção, na tarefa de exploração com o objetivo de compreender o código da aplicação.

Embora essa ferramenta possa ser implantada em qualquer organização que desenvolve e faz manutenção de software, o ambiente de software livre mostra-se um candidato ideal. Para sua utilização neste ambiente, as fases de desenvolvimento são bem delimitadas e seu desenvolvimento ocorre normalmente de forma distribuída, dificultando um pouco a troca de conhecimento entre os desenvolvedores. Em projetos de software livre sempre existe o interesse de desenvolvedores em colaborar com a adição de novas funcionalidades ou correções de erros, porém essa atividade não é simples. O desenvolvedor precisa compreender o software e decidir como serão feitas as modificações seguindo as regras arquiteturais e o estilo de programação usado no projeto. Se existirem componentes reutilizáveis encontrados por alguma ferramenta seu trabalho se torna eficaz. Logo, reuso na fase da evolução do software é possível se forem utilizadas as ferramentas corretas.

Nesse capitulo, o Sceek é apresentado em detalhes. Primeiramente são analisados alguns trabalhos acadêmicos e aplicações comerciais relacionados a esse (Seção 4.1). Na Seção 4.2, a arquitetura do sistema é apresentada e na Seção 4.3 são discutidos os componentes que fazem parte do sistema.

4.1 Trabalhos Relacionados

Vários trabalhos de propósito acadêmico e comercial são, em alguns aspectos, relacionados com o Sceek. Embora exista uma grande quantidade de projetos com o objetivo de busca em código-fonte [30, 31, 32, 33], nessa seção serão analisados, com mais detalhes, apenas dois projetos:

Relo [34] e Sourcerer [35]. Estes projetos foram escolhidos por possuir, separadamente, os objetivos semelhantes aos do Sceek.

Relo

O Relo foi desenvolvido para ajudar desenvolvedores explorar e entender pequenas partes de um grande projeto de software. Relo cria e automaticamente gerencia o contexto desses trechos na visualização, desse modo ajudando o desenvolvedor criar uma representação mental do código. A Figura 3 mostra um exemplo de visualização do Relo, onde nota-se a semelhança com UML, pois um dos objetivos do Relo é a usabilidade e o fácil entendimento pelos desenvolvedores. Nesse exemplo as relações que a classe `PolyLineFigure` possui são exibidas.

O Sceek tem o mesmo objetivo de fornecer ao desenvolvedor suporte nas tarefas de exploração e compreensão, porém diferentemente do Relo, que foi desenvolvido como um *plug-in* do Eclipse, ele foi desenvolvido como um sistema web e a maneira como o software é visualizado se baseia em facetas (características) [7, 14].

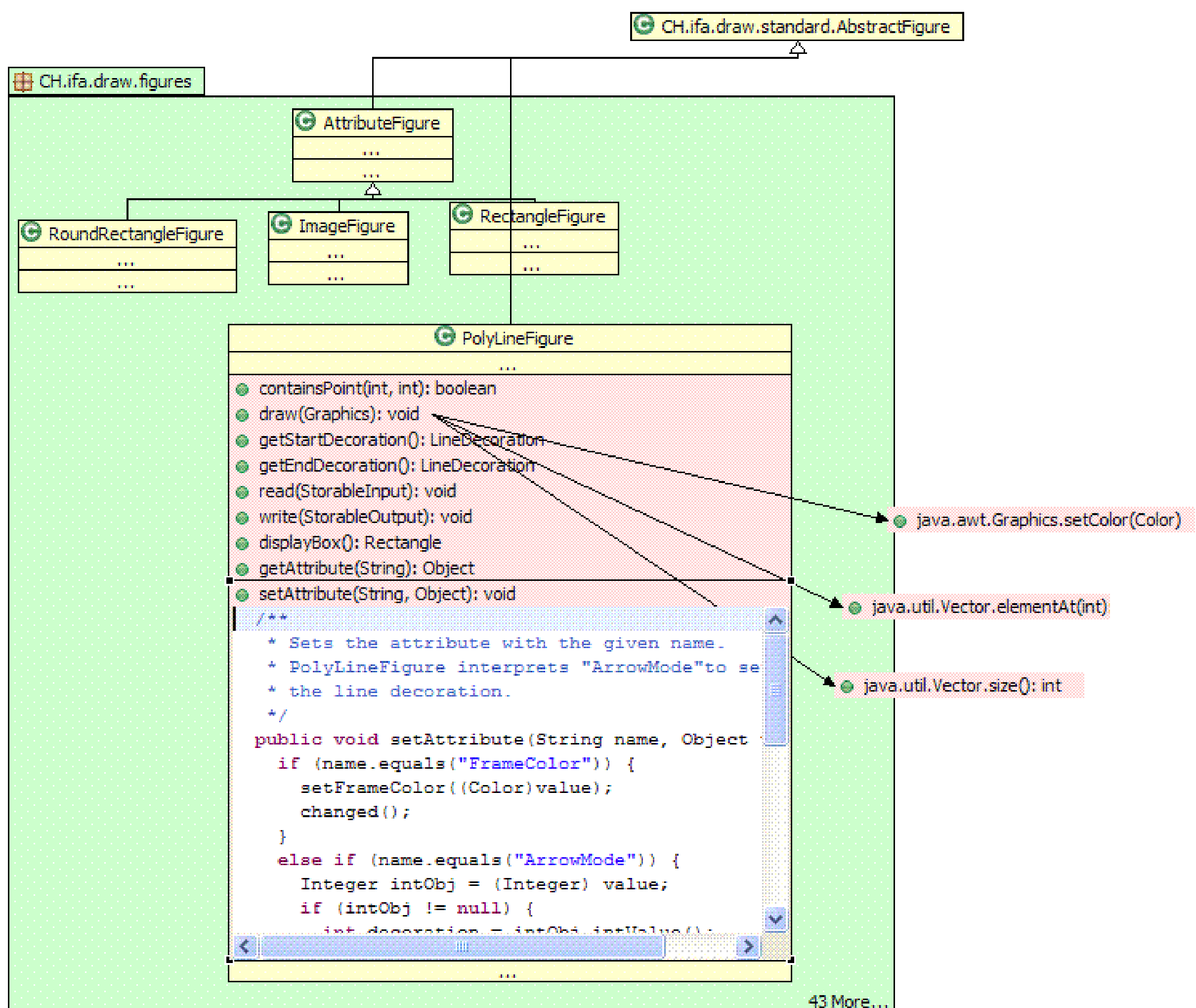

Figura 3. Exemplo de visualização do Relo.

Sourcerer

O Sourcerer foca na busca e recuperação de código-fonte utilizando não apenas palavras-chave, mas também propriedades estruturais, como a assinatura dos métodos e os elementos sintáticos no código, além dos relacionamentos entre elementos dos códigos [35]. O Sceek também utiliza propriedades estruturais na construção de algumas facetas. Além da busca por certas propriedades, isso possibilita uma navegação pelos códigos do software de forma intuitiva. A Figura 4 mostra um exemplo de busca feita com o Sourcerer.

SOURCERER

All | Components | Functions | Fingerprints

test | Go

Search in comments ?

1 2 3 4 5 6 7 8 9 10 | >>

PasswordTest

CLASS (rank: 0.15)

Relations >> Find uses

Fingerprints >> Show Details

[X] CLOSE

Code Structure

Synchronized	0	Waits	0	Notifys	0	Starts	0
Joins	0	Loops	0	IF	0	SWITCH	0
Lines Of Code	0	Instantiations	0	Path	0	Average Loop Length	0
MAX Loop Nesting	0						

Java Attributes

Modifiers (+,-,#)=(1,2,4)	1	Field Self Type?	0	Classes	0	Interfaces	0
Declared Methods	1	Method	13	Declared Constructors	0	Constructors	1
Declared Fields	0	Fields	0	Static Initializers	0	Parameters	0
Declared Overloads	0	Overloads	0	Overrides	0	Implements	0
Parents	1						

Figura 4. Exemplo de busca com o Sourcerer.

4.2 Componentes

O Sceek foi desenvolvido utilizando vários componentes e *frameworks*, alguns componentes foram desenvolvidos como parte desse trabalho, como o extrator de informações do código e a adaptação interface gráfica, e outros foram reutilizados (classificadores). Uma boa parte dos componentes foi desenvolvida na linguagem Python [36] por sua fácil adequação a problemas de processamento de texto [37].

A abordagem utilizada é orientada a projetos, isto é, a busca e navegação de código-fonte são feitas por projeto de software, apenas nos códigos deste. A razão dessa abordagem é o uso do Sceek na fase de manutenção. O desenvolvedor só precisa se preocupar com os artefatos do projeto em questão, muita informação pode prejudicar a compreensão de todo o projeto de software.

O fluxo de dados entre os componentes do sistema funciona da seguinte maneira: o projeto selecionado é processado pelo extrator de informação que analisa todo código-fonte Java do projeto extraindo informações sobre a estrutura das classes, relacionamentos entre tipos e os comentários. Classificadores binários são treinados com uma fração dessas informações e depois utilizados no restante.

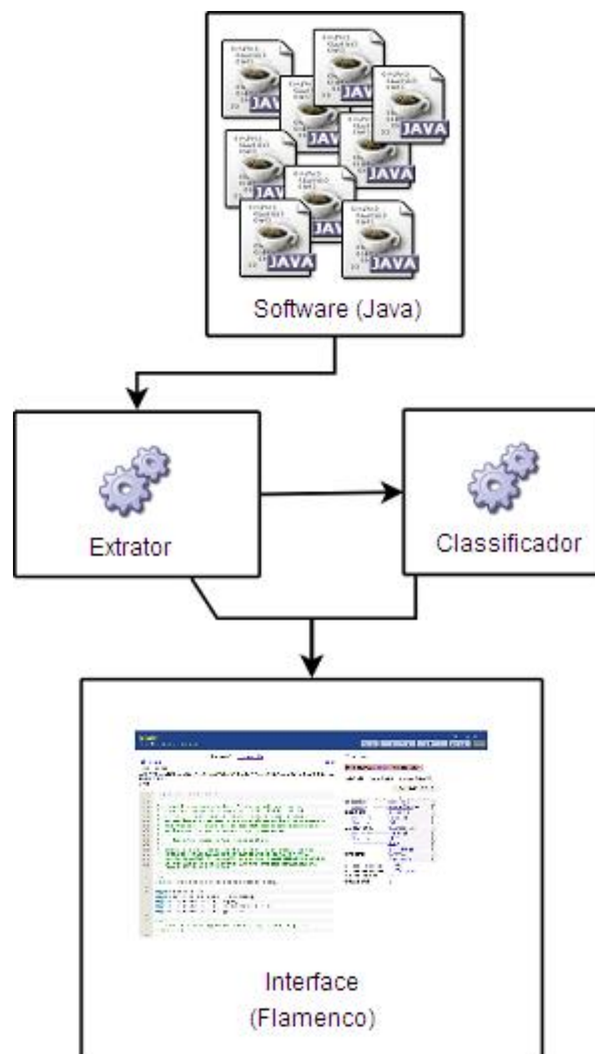

Figura 5. Organização dos componentes no Sceek.

Toda essa informação extraída e classificada é exibida na interface gráfica como categorias, onde é possível navegar e fazer busca textual. A Figura 5 ilustra a organização e os relacionamentos dos componentes do Sceek. Nas próximas seções os componentes do sistema são apresentados, com alguns exemplos de seu uso.

4.2.1 Extrator de Informação

O extrator de informação é responsável por analisar as estruturas dos códigos-fonte e obter os dados para a criação das facetas que serão exibidas na interface gráfica e as bases usadas para treinar os classificadores. As informações obtidas dos códigos foram os tipos dos parâmetros, das exceções e do retorno dos métodos das classes, assim como informação sobre herança. Essas informações complementadas com os comentários são usadas para treinar os classificadores usados na construção das categorias conceituais, isto é, categorias que são definidas por desenvolvedores sobre o domínio daquele código, por exemplo, um código Java pode ser classificado como "*networking*" por envolver tarefas relacionadas com redes e/ou transmissão de dados.

Java

```
import java.util.*;

public class HelloWorld
extends Thread
{
  private int N;

  private long sleep;

  public HelloWorld(int N, long
sleep)
  {
    this.N = N;
    this.sleep = sleep;
  }

}
```

XML

```
<java-class-file name="helloworld.java">
<import module="java.util.*"/>
<classname="HelloWorld" visibility="public">
 <superclass name="Thread"/>
  <field name="N" visibility="private">
   <type primitive="true" name="int"/>
  </field>
  <field name="sleep" visibility="private">
   <type primitive="true" name="long"/>
  </field>
 <constructor visibility="public"
  name="HelloWorld">
  <formal-arguments>
   <formal-argument name="N">
    <type primitive="true" name="int"/>
   </formal-argument>
   <formal-argument name="sleep">
    <type primitive="true" name="long"/>
   </formal-argument>
  </formal-arguments>
  <assignment-expr op="=">
   <lvalue>
   <field-access field="N">
    <this/>
   </field-access>
   </lvalue>
   <var-ref name="N"/>
  </assignment-expr>
  <assignment-expr op="=">
   <lvalue>
    <field-access field="sleep">
     <this/>
    </field-access>
   </lvalue>
   <var-ref name="sleep"/>
  </assignment-expr>
 </constructor>
</class>
</java-class-file>
```

Figura 6. Exemplo de XML gerado pelo Java2XML.

A primeira atividade desse componente é geração de um XML representando os códigos com suas estruturas. Nessa fase foi utilizado o Java2XML [38], uma biblioteca que gera uma representação em XML do código Java. A Figura 6 mostra um exemplo de transformação feita com o Java2XML.

Um *parser* XML do tipo SAX é executado nesse XML obtendo apenas os atributos de interesse: classe, métodos e exceções. Essas informações são usadas na construção das categorias estruturais. As categorias estruturais utilizadas nesse trabalho são apresentadas na tabela abaixo.

Tabela 1. Categorias baseadas nas estruturas do código Java.

Categoria	Exemplos de elementos
Herança (*implements*, *extends*)	*extends* >> Task
Tipo do Retorno (*object*, *primitive*)	*object* >> Map, *primitive* >> int
Tipo do Parâmetro (*object*, *primitive*)	*object* >> List, *primitive* >> double
Exceção	FileNotFoundException

As informações sobre os tipos usados em um código unidos aos comentários, que são extraídos com um script [39] desenvolvido em AWK, são usadas para criar as bases utilizadas no treinamento dos classificadores.

4.2.2 Classificador

As categorias conceituais são atribuídas aos códigos utilizando o classificador *Support Vector Machine* (SVM), já exposto no capítulo três. Primeiramente, as bases são criadas a partir dos comentários e dos tipos mais usados nos códigos. A abordagem utilizada para o treinamento da SVM é a mesma utilizada nos problemas de categorização de textos [28].

Figura 7. Representação de código como um vetor de palavras.

As bases usadas para o treinamento consistem em vetores com todos os valores numéricos. O tamanho desses vetores varia de acordo com cada documento, mas o número máximo de elementos é limitado pelo tamanho do dicionário. O dicionário é criado analisando as palavras mais freqüente em todo o conjunto de dados.

Tabela 2. Categorias conceituais usadas no Sceek.

Categorias	Explicação
Input/Output	Utilizado no tratamento de arquivos, diretórios, *streams*, etc.
Networking	Tarefas relacionadas a redes, comunicação, protocolos, *sockets*, etc.
User Interface	Código utilizando funcionalidades relacionadas à exibição de dados, *Swing*, SWT, etc.
Text Manipulation	Uso de expressões regulares, tratamento de *Strings*, etc.
Application	Código com a lógica da aplicação ou relacionado a isso.
XML	Manipulação de tecnologias relacionadas à XML.
System	Tarefas do sistema.

Essas atividades iniciais de geração são executadas utilizando uma biblioteca de mineração de texto chamada de TSMK - *Text Miner Software Kit* [40]. O TMSK é composto de algumas rotinas para processamento e predição de categorias de textos baseados no formato XML. Uma representação didática da organização dos vetores gerados é apresentada na Figura 7. Nesse exemplo as palavras existentes nos comentários do código Java são mapeadas, com o número da freqüência de aparições, em um vetor com as palavras do dicionário criado. Uma vez que a base de treinamento é construída, os classificadores são treinados usando a biblioteca LIBSVM [41]. O modelo criado pelo LIBSVM após o treinamento representa o classificador e já é capaz de classificar códigos não usados durante o treinamento.

As categorias utilizadas inicialmente no Sceek são apresentadas na tabela acima, acompanhadas de uma explicação de como essas categorias podem ser identificadas.

4.2.3 Interface

As informações obtidas nas fases anteriores precisam ser exibidas de forma intuitiva e possibilitar uma fácil navegação por todas as categorias. Para alcançar esse objetivo foi usado o *Flamenco* [8] para a construção da interface de busca. O *Flamenco* é um *framework* para interface de busca cujo objetivo é permitir que usuários explorem grandes espaços de informação de maneira flexível sem sentir-se perdido. Uma propriedade importante do *Flamenco* é a exposição dos metadados das categorias, possibilitando escolher varias categorias e organizar os resultados da busca por palavra-chave. A interface usa metadados de categorias hierárquicas de forma que permite os usuários refinar e expandir a consulta atual, enquanto mantêm uma representação consistente da estrutura da coleção.

As figuras 8 e 9 exibem as interfaces do Sceek utilizando o *Flamenco*. A interface é integrada com busca textual, permitindo aos usuários seguirem *links*¸ adicionar outros termos de pesquisa, seguir mais *links*, sem interromper o fluxo de interação.

Figura 8. Página inicial do Sceek.

A interface do Sceek permite uma série de funcionalidades:

- **<u>Busca textual</u>** – É possível fazer busca utilizando palavras-chave para pesquisar códigos. Essa funcionalidade é implementada utilizando o *Lucene* e mesmo durante a busca é possível refinar a consulta com bases na facetas.
- **<u>Busca baseada em facetas</u>** – É possível explorar os códigos utilizando as facetas (simples e hierárquicas). Um exemplo de faceta com hierarquia é a *Herança*, podendo ter dois tipos, utilizando o *implements* ou o *extends*.
- **<u>Histórico</u>** – Os históricos de todas as buscas ficam armazenados por cada seção.
- **<u>Recomendações</u>** – É possível encontrar resultados semelhantes durante a visualização de algum resultado (Figura 9).

```
356.         if (null == u) {
357.             return "";
358.         }
359.         return u.getName();
360.     }
361.
362. }
```

Código

Current search:

INHERITANCE: extends > Task ×

Select any link to see items in a related category.

Semelhança baseada em facetas

Find Similar Items

more general categories	information about this item
INHERITANCE	INHERITANCE
extends (528)	Task (105)
RETURN TYPE	RETURN TYPE
object (545)	TypeNames (1)
	String (376)
	Server (2)
	View (6)
ARGUMENT TYPE	ARGUMENT TYPE
primitive (373)	int (178)
object (714)	View (7)
	Exception (101)
	String (516)
EXCEPTION	EXCEPTION
	BuildException (363)

Faceta

Figura 9. Exibição de um resultado com a possibilidade de encontrar resultados semelhantes.

Capítulo 5

Experimentos

Este capítulo tem como objetivo descrever os experimentos feitos utilizando bases de dados criadas a partir dos códigos Java. Esses códigos foram obtidos do projeto *open source*: Apache *Ant* [42]. Além disso, é apresentada uma análise dos resultados obtidos com a aplicação das técnicas de aprendizagem de máquina SVM, *Naive Bayes* e *K-means*.

5.1 Preparação dos Experimentos

Os experimentos foram realizados utilizando o LIBSVM [41] e o Weka [43]. O Weka é software desenvolvido pela Universidade de Wakato, Nova Zelândia, utilizando Java e possui diversos algoritmos e ferramentas para processamento de dados, classificação, regressão, *clustering*, regras de associação e visualização.

As bases de dados extraída do projeto *Ant* contem apenas 250 padrões de treinamento gerados a partir de 250 classes desse projeto. As classes são escolhidas aleatoriamente e depois de uma leitura é escolhida a(s) categoria(s) a qual ela pertence. Algumas classes podem pertencer a mais de uma categoria.

Para cada categoria há um classificador binário, com duas classes (+1 e -1) que representam que determinado padrão é (+1) ou não (-1) daquela categoria. Na Tabela 3 são apresentadas as quantidades de padrões por cada classe.

Tabela 3. Distribuição das categorias em cada classe.

	+1	-1
XML	11	239
UI	13	237
TEXT PROCESSING	17	233
SYSTEM	39	211
NETWORKING	17	233
IO	119	131
APPLICATION	90	160

5.2 Experimentos com Aprendizagem Supervisionada

Os experimentos com aprendizagem supervisionada foram feitos utilizando os valores padrões nos parâmetros dos classificadores. O treinamento foi feito utilizando validação cruzada [26], que consiste em dividir o conjunto total de padrões em N conjuntos com tamanhos aproximadamente iguais, realizando o treinamento N vezes, sendo a cada treinamento um dos conjuntos deixado para teste e os outros N-1 para treinamento, dessa forma aumentando o número de situações diferentes apresentadas ao classificador. Os erros de classificação obtidos nesses N treinamentos são usados no cálculo da média dos erros na classificação. Nesse trabalho, foi considerado o N = 10.

Na avaliação dos classificadores foram utilizadas as seguintes medidas:

- ***True Positive*** **(TP)** – Quantidade percentual de padrões positivos classificados como positivo.
- ***False Positive*** **(FP)** – Quantidade percentual de padrões negativos classificados como positivo.
- **Precisão** – Quantidade percentual de padrões que foram corretamente classificados como pertencente à determinada categoria.
- **Cobertura** – Quantidade percentual de padrões que conseguiram ser recuperados.
- ***F-measure*** – Medida harmônica da precisão e cobertura é utilizada para medir a exatidão do classificador quando um único número é preferido.

Os resultados dos experimentos com *Naive Bayes* e SVM para cada categoria são apresentados a partir da Tabela 4 até o final dessa seção.

Tabela 4. Taxa de acertos dos classificadores.

	Taxa de Acerto	
	SVM	Naive Bayes
XML	**96.8%**	72.8%
UI	**95.2%**	79.6%
TEXT PROCESSING	**94.8%**	79.6%
SYSTEM	**91.6%**	63.2%
NETWORKING	**95.2%**	64%
IO	**74.4%**	59.6%
APPLICATION	**64.8%**	55.6%

XML

Tabela 5. Medidas de avaliação do classificador *Naive Bayes* na categoria *XML*.

TP	FP	Precisão	Cobertura	F-Measure	Classe
0.737	0.571	0.978	0.737	0.84	“-1”
0.429	0.263	0.045	0.429	0.081	“+1”

Tabela 6. Medidas de avaliação do classificador SVM na categoria *XML*.

TP	FP	Precisão	Cobertura	F-Measure	Classe
0.996	1	0.972	0.996	0.984	"-1"
0	0.004	0	0	0	"+1"

UI

Tabela 7. Medidas de avaliação do classificador *Naive Bayes* na categoria *UI*.

TP	FP	Precisão	Cobertura	F-Measure	Classe
0.812	0.545	0.97	0.812	0.884	"-1"
0.455	0.188	0.1	0.455	0.164	"+1"

Tabela 8. Medidas de avaliação do classificador SVM na categoria *UI*.

TP	FP	Precisão	Cobertura	F-Measure	Classe
0.996	1	0.956	0.996	0.975	"-1"
0	0.004	0	0	0	"+1"

Text Processing

Tabela 9. Medidas de avaliação do classificador *Naive Bayes* na categoria *Text Processing*.

TP	FP	Precisão	Cobertura	F-Measure	Classe
0.797	0.231	0.984	0.797	0.881	"-1"
0.769	0.203	0.172	0.769	0.282	"+1"

Tabela 10. Medidas de avaliação do classificador SVM na categoria *Text Processing*.

TP	FP	Precisão	Cobertura	F-Measure	Classe
1	1	0.948	1	0.973	"-1"
0	0	0	0	0	"+1"

System

Tabela 11. Medidas de avaliação do classificador *Naive Bayes* na categoria *System*.

TP	FP	Precisão	Cobertura	F-Measure	Classe
0.648	0.55	0.931	0.648	0.764	"-1"
0.45	0.352	0.1	0.45	0.164	"+1"

Tabela 12. Medidas de avaliação do classificador SVM na categoria *System*.

TP	FP	Precisão	Cobertura	F-Measure	Classe
0.996	1	0.92	0.996	0.956	"-1"
0	0.004	0	0	0	"+1"

Networking

Tabela 13. Medidas de avaliação do classificador *Naive Bayes* na categoria *Networking*.

TP	FP	Precisão	Cobertura	F-Measure	Classe
0.646	0.462	0.962	0.646	0.773	“-1”
0.538	0.354	0.077	0.538	0.135	“+1”

Tabela 14. Medidas de avaliação do classificador SVM na categoria *Networking*.

TP	FP	Precisão	Cobertura	F-Measure	Classe
0.979	0.538	0.971	0.979	0.975	“-1”
0.462	0.021	0.545	0.462	0.5	“+1”

IO

Tabela 15. Medidas de avaliação do classificador *Naive Bayes* na categoria *IO*.

TP	FP	Precisão	Cobertura	F-Measure	Classe
0.703	0.441	0.354	0.703	0.471	“-1”
0.559	0.297	0.846	0.559	0.673	“+1”

Tabela 16. Medidas de avaliação do classificador SVM na categoria *IO*.

TP	FP	Precisão	Cobertura	F-Measure	Classe
0.156	0.054	0.5	0.156	0.238	“-1”
0.946	0.844	0.765	0.946	0.846	“+1”

Application

Tabela 11. Medidas de avaliação do classificador *Naive Bayes* na categoria *Application*.

TP	FP	Precisão	Cobertura	F-Measure	Classe
0.481	0.311	0.733	0.481	0.581	“-1”
0.689	0.519	0.428	0.689	0.528	“+1”

Tabela 18. Medidas de avaliação do classificador SVM na categoria *Application*.

TP	FP	Precisão	Cobertura	F-Measure	Classe
0.906	0.811	0.665	0.906	0.767	“-1”
0.189	0.094	0.531	0.189	0.279	“+1”

5.3 Experimentos com Aprendizagem Não-Supervisionada

Os experimentos realizados com aprendizagem não-supervisionada usaram a algoritmo *K-means* da biblioteca TSMK [41]. A execução do experimento consistiu em variar o valor de *K* para obter a distribuição dos códigos e quais são as palavras descritivas de cada agrupamento de dados, ou *cluster*.

A base usada representa todos os 712 códigos funcionais Java existente no Apache Ant sem nenhuma informação sobre as categorias que pertencem. Os resultados dos experimentos são apresentados nas seguintes tabelas:

Tabela 19. Resultados da execução do *K-means* com K = 8.

Cluster	Tamanho	Palavras Descritivas
1	47	resource resources iterator add resourcecollection buildexception object nested check
2	114	buildexception error task string
3	72	ioexception stream string constructor read file error inputstream
4	43	constructor null ioexception object stream reader tools input filter read specified character creates based string line
5	67	checkstyle bc visibilitymodifier buildexception file task error use
6	85	task buildexception command file commandline execute line build path specified run
7	119	file object use string buildexception files java check directory
8	165	file buildexception task use add string

Tabela 20. Resultados da execução do *K-means* com K = 7.

Cluster	Tamanho	Palavras Descritivas
1	99	buildexception error resource resources iterator object string check
2	62	ioexception stream string read constructor inputstream error outputstream
3	74	use constructor file string object error java number
4	67	null ioexception stream creates constructor object specified tools task line use project error read
5	130	buildexception task file command execute build specified commandline line
6	119	object buildexception string use instance file add project element check reference
7	161	file use files buildexception task add error directory ioexception list

Tabela 21. Resultados da execução do *K-means* com K = 6.

Cluster	Tamanho	Palavras Descritivas
1	104	buildexception error resources resource object iterator
2	64	constructor buildexception ioexception version type stream
3	88	ioexception stream error read input string output object constructor null
4	150	task buildexception file command line execute build specified commandline
5	120	file use string instance buildexception java check directory
6	186	file buildexception use task add error string list vector

5.4 Análise dos Resultados

Os resultados dos classificados demonstram sua aplicabilidade no problema de categorização de códigos Java. O classificador SVM foi o que obteve um melhor resultado em relação ao número de acertos na base de treinamento. Contudo, os resultados, em geral, não foram expressivos devido ao desbalanceamento das classes na base. O número de exemplos negativos é maior em todas as classes, isso provoca o aumento de desempenho com relação apenas aos exemplos negativos.

A categoria com o melhor desempenho foi *IO* que obteve um *F-measure* de 0.846 para a classe positiva (Tabela 17). É importante destacar que a categoria *IO* é a que possui o maior numero de exemplos positivos (Tabela 3). Portanto, o desempenho dos classificadores nas tarefas de categorizar código depende diretamente da quantidade e qualidade dos exemplos usados no treinamento.

Nos experimentos com *K-means* o desempenho pode ser medido avaliando a qualidade dos *clusters* construídos de forma subjetiva, ou seja, como a organização dos códigos foi útil para a exploração do software. Nesse caso, para uma completa avaliação do aprendizado não-supervisionado seria necessária a opinião dos usuários em como e qual configuração dos *clusters* foi mais útil.

Capítulo 6

Conclusão

A atividade de desenvolvimento de software é bastante complexa, a ponto de ser considerada arte em alguns aspectos [44]. Os processos de manutenção e evolução são os mais complexos e caros do desenvolvimento de software por envolverem várias dificuldades (novos desenvolvedores, novas funcionalidades que quebram a arquitetura, etc). Essas atividades envolvem a necessidade de compreender grandes conjuntos de informações como código, documentação, diagramas, etc.

A compreensão e a leitura de código-fonte são as atividades mais habituais de todo desenvolvedor, seja na tarefa de manutenção, corrigindo erros e adicionando novas funcionalidades, ou na exploração de um projeto de software buscando componentes para serem reusados.

Esse trabalho teve como objetivo desenvolver um ambiente com suporte a exploração e busca de código-fonte Java baseado em facetas, estruturais e conceituas. Adicionalmente, para a criação da facetas conceituais ou semânticas foi feito experimentos com alguns classificadores e analisados sua aplicabilidade nesse projeto.

Nesse capitulo serão discutidos as contribuições desse projeto, assim como as dificuldades encontradas durante sua construção e como ele pode ser estendido e aperfeiçoado com trabalhos futuros.

6.1 Contribuições

As contribuições desse trabalho podem ser organizadas pelas áreas em que os resultados geraram algum beneficio. Com relação aos experimentos com as técnicas de aprendizado de máquina as contribuições foram:

- Utilização e avaliação de técnicas de aprendizado de máquina na categorização de código-fonte como facetas em um determinado projeto.

- Criação de uma série de componentes para extrair informação de código-fonte e conseqüentemente criar novas bases de dados.

- Incorporação de informações sobre os tipos utilizados nos códigos nas bases de dados, aumentando o grau de informação de cada padrão uma vez que os tipos representam algum conceito mais abstrato.

Os benefícios para os desenvolvedores e as organizações que utilizarem o Sceek são destacados abaixo:

- Criação de uma ferramenta que contêm todas as informações relativas ao código estruturadas pensando na usabilidade.

- Uma interface alternativa e eficaz para a leitura de código-fonte. No Sceek é possível ler código de forma exploratória, refinando os critérios em qualquer momento.

- A tarefa de manutenção ganha em produtividade com o Sceek. É possível encontrar elementos que possam ser reusado no próprio projeto.

6.2 Dificuldades Encontradas

No decorrer desse trabalho foram encontradas algumas dificuldades em certas tarefas. Mesmo não impedindo a geração de um bom resultado gerado pela ferramenta, a solução desses problemas ajudaria em vários aspectos.

Novamente, podemos dividir entre as dificuldades relativas às técnicas de aprendizado de máquina e as do sistema Sceek. Com relação aos experimentos com os algoritmos de aprendizado de máquina:

- A construção das bases de treinamento consome muito tempo. Uma solução para esse problema poderia ser a incorporação dessa atividade na própria interface, desse modo mais usuários poderiam contribuir na construção da base.

- As categorias são desbalanceadas. Dependendo do projeto podem existir mais padrões de determinada classe do que das outras degradando o desempenho dos classificadores.

No desenvolvimento do sistema Sceek as seguintes dificuldades foram encontradas:

- A ausência de flexibilidade do *Flamenco* em alterar certas propriedades arquiteturais da interface.

- Dificuldade de definir as facetas conceituais.

6.3 Trabalhos Futuros

Como trabalhos futuros, existem várias funcionalidades que podem ser adicionadas ao Sceek, assim como novos experimentos que poderiam ser feitos avaliando novos algoritmos com bases diferentes.

As seguintes sugestões foram pensadas durante o desenvolvimento do trabalho, demonstrando sua utilidade e capacidade de evolução:

- Suporte às estruturas existentes no Java 6 [45], como anotações e/ou *generics*. Essa funcionalidade poderia ser feita utilizando um *parser* de Java 6 e extraindo as informações desses elementos de maneira semelhante a descrita na Seção 4.2.1.

- Suporte a outra linguagens: C/C++, Python, Ruby. Essa tarefa é bastante desafiante nas linguagens dinâmicas, pois a semântica de boa parte do código só é conhecida em tempo de execução.

- Suportar orientação a aspectos [46], isto é, possibilitar navegação baseada em aspectos. Para a execução dessa tarefa seria necessário o uso de um *parser* para extrair essas informações.

- Criar um componente de *folksonomy* [47], também chamado de classificação social, consiste na pratica de criar e gerenciar colaborativamente palavras de certas categorias. O uso de *folksonomy* facilitaria a construção das bases de treinamento.

- Modificar a infra-estrutura, incorporando o *Hadoop* [48]. *Hadoop* é uma plataforma que possibilita executar aplicações com grande processamento de dados de forma distribuída e paralela.

Bibliografia

[1] BOEHM, B. W. K., *Software Economics: A Roadmap*. The Future of Software Engineering. 22nd International Conference on Software Engineering, pp.319-343, 2000.

[2] JARZABEK, Stanislaw, *Effective Software Maintenance and Evolution: A Reuse-Based Approach*. Auerbach Publications, 2007.

[3] MAYRHAUSER, A., VANS, A. M., *Program Comprehension During Software Maintenance and Evolution*. Computer 28, 1995.

[4] KARAHASANOVIĆ, A., LEVINE, A. K. e THOMAS, R., *Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study*. J. Syst. Software. 80, 2007.

[5] YE, Y. E KISHIDA, K. *Toward an Understanding of the Motivation of Open Source Software Developers*, Proceedings of 2003 International Conference on Software Engineering (ICSE2003), Portland, Oregon, pp 419-429, May 3-10, 2003.

[6] SPINELLIS, Diomidis. *Code Reading: The Open Source Perspective*. Addison Wesley, 2003.

[7] MCILROY, M. D. *Mass Produced Software Components*, NATO Software Engineering Conference Report, Garmisch, Germany, Outubro, 1968, pp. 79-85.

[8] YEE, P., SWEARINGEN, K., LI, K. e HEARST, M. *Faceted Metadata for Image Search and Browsing*, Proceedings of ACM CHI, 2003.

[9] SAMETINGER, J. *Software Engineering with Reusable Components*, Springer-Verlag, 1997, pp.275.

[10] KRUEGER, C.W. *Software Reuse*, ACM Computing Surveys, Vol. 24, No. 02, Junho, 1992, pp. 131-183.

[11] OSGi. Disponível em:<http://www.osgi.org/> Acesso em: 10 de Novembro de 2007.

[12] Eclipse. Disponível em: <http://www.eclipse.org/>Acesso em: 10 de Novembro de 2007.

[13] PRIETO-DIAZ, R.; Freeman, P. *Classifying Software for Reusability*, IEEE Software, Vol. 04, No. 01, Janeiro, 1987, pp. 06-16.

[14] PRIETO-DIAZ, R. *Implementing Faceted Classification Software Reuse*. Communications of the ACM, Vol. 34, No. 5, 1991.

[15] MILI, R.; MILI, A.; MITTERMEIR, R. T. *Storing and Retrieving Software Components: A Refinement Based System*. IEEE Transactions on Software Engineering, Vol. 23, No. 7, 1997.

[16] GANGOPADHYAY, D. e MITRA, S.. *Understanding Frameworks by Exploration of Exemplars*. Proceedings of the International Workshop on Computer-Aided Software Engineering (CASE'95), 1995.

[17] Webware. Disponível em: < http://www.webwareforpython.org/>Acesso em: 10 de Novembro de 2007.

[18] CLEAR, T., *Comprehending Large Code Bases - The Skills Required for Working in a "Brown Fields" Environment*. SIGCSE Bulletin, 37(2), 2005.

[19] SILLITO, J., MURPHY, G. C., e DE VOLDER, K., *Questions programmers ask during software evolution tasks.* Proceedings of the 14th ACM SIGSOFT international Symposium on Foundations of Software Engineering, 2006.

[20] SINHA, V.; MILLER, R.; KARGER, D., *Incremental exploratory visualization of relationships in large codebases for program comprehension.* OOPSLA '05, 2005.

[21] MITCHELL, T. *Machine Learning*. McGraw-Hill. 1997.

[22] VAPNIK, V. *Support-vector networks*. Machine Learning, v.20, p.273-297, November, 1995.

[23] VAPNIK, V. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

[24] CRISTIANINI, N. e SHAWE-TAYLOR, J. *An Introduction to Support Vector Machines.* Cambridge University Press, 2000.

[25] WEBB, A. *Statistical Pattern Recognition*. John Wiley & Sons, second edition, 2002.

[26] DUDA, R. O., HART, P. E. e STORK, D. G. *Pattern Classification*. Wiley- Interscience, second edition, 2000.

[27] OSUNA, E., FREUND, R. e GIROSI, F. *Training support vector machines: an application to face detection*. CVPR'97, 1997.

[28] THORSTEN, J. *Text Categorization with Support Vector Machines: Learning with Many Relevant Features*. Proceedings of the European Conference on Machine Learning, Springer, 1998.

[29] SpamBayes. Disponível em:<http://spambayes.sourceforge.net/> Acesso em: 10 de Novembro de 2007.

[30] YE, Y., FISCHER, G., *Supporting reuse by delivering task-relevant and personalized information.* In Proceedings of the 24th international Conference on Software Engineering, 2002.

[31] Codase - Source Code Search Engine. Disponível em: < http://www.codase.com/ > Acesso em: 10 de Novembro de 2007.

[32] Koders - Source Code Search Engine. Disponível em: < http://www.koders.com/ > Acesso em: 10 de Novembro de 2007.

[33] SEACORD R. C., HISSAM S. A., WALLNAU K. C. *Agora: A Search Engine for Software Components*, IEEE Internet Computing, Vol. 02, No. 06, Novembro, 1998, pp. 62-70.

[34] SINHA, V., KARGER, D. e MILLER, R., *Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases.* OOPSLA'05 Eclipse Technology eXchange (ETX) Workshop, 2005.

[35] BAJRACHARYA, S *et al*. *Sourcerer: a search engine for open source code supporting structure-based search*. 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPSLA '06. ACM, New York, NY, 681-682, 2006.

[36] Python Programming Language. Disponível em: <http://www.python.org/ > Acesso em: 10 de Novembro de 2007.

[37] Mertz, D., *Text Processing in Python.* Addison-Wesley Professional, 2003.

[38] java2xml. Disponível em: < https://java2xml.dev.java.net/> Acesso em: 10 de Novembro de 2007.

[39] XSCC. Disponível em: <http://members.tripod.com/vgoenka/unixscripts/xscc.html/> Acesso em: 10 de Novembro de 2007.

[40] Text Mining Software. Disponível em: <http://www.data-miner.com/>, Acesso em: 10 de Novembro de 2007.

[41] LIBSVM - A Library for Support Vector Machines. Disponível em: <http://www.csie.ntu.edu.tw/~cjlin/libsvm/> Acesso em: 10 de Novembro de 2007.

[42] Apache Ant. Disponível em: <http://ant.apache.org/> Acesso em: 10 de Novembro de 2007.

[43] Weka. Disponível em: <http://www.cs.waikato.ac.nz/ml/weka/> Acesso em: 10 de Novembro de 2007.

[44] Mob Software: The Erotic Life of Code. Disponível em: <http://www.dreamsongs.com/MobSoftware.html> Acesso em: 10 de Novembro de 2007.

[45] Java. Disponível em: <http://java.sun.com/> Acesso em: 10 de Novembro de 2007.

[46] AspectJ. Disponível em: <http://www.eclipse.org/aspectj/> Acesso em: 10 de Novembro de 2007.

[47] MATHES, A. *Folksonomies -- Cooperative Classification and Communication Through Shared Metadata*. Computer Mediated Communication -- LIS590CMC, Dezembro, 2004.

[48] Hadoop. Disponível em: <http://lucene.apache.org/hadoop/> Acesso em: 10 de Novembro de 2007.

[49] HENNINGER, S. *An Evolutionary Approach to Constructing Effective Software Reuse Repositories*. ACM Transactions on Software Engineering and Methodology, Vol. 06, No. 02, 1997, pp. 111-140.

[50] MILI A., MILI R., MITTERMEIR R. *A Survey of Software Reuse Libraries*, Annals Software Engineering, Vol. 05, 1998, pp. 349-414.

[51] THORSTEN, J. *Transductive Inference for Text Classification using Support Vector Machines*. Proceedings of the International Conference on Machine Learning (ICML), 1999.