Dissertação de Mestrado

Um Método para Descoberta Automática de Regras para a Detecção de Bad Smells

Lucas Benevides Viana de Amorim
lucas@ic.ufal.br

Orientadores:

Evandro de Barros Costa
Baldoino Fonseca dos Santos Neto

Maceió, Abril de 2014

Lucas Benevides Viana de Amorim

Um Método para Descoberta Automática de Regras para a Detecção de Bad Smells

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Curso de Mestrado em Modelagem Computacional de Conhecimento do Instituto de Computação da Universidade Federal de Alagoas.

Orientadores:

Evandro de Barros Costa

Baldoino Fonseca dos Santos Neto

Maceió, Abril de 2014

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária: Maria Auxiliadora G. da Cunha

A524m Amorim, Lucas Benevides Viana de.
Um método para descoberta automática de regras para a detecção de Bad Smells / Lucas Benevides Viana de Amorim – 2014.
65 f. : il., tabs.

Orientador: Evandro de Barros Costa.
Coorientador: Baldoino Fonseca dos Santos Neto.
Dissertação (Mestrado em Modelagem Computacional do Conhecimento) – Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2014.

Bibliografia: f. 58-62.
Apêndices: f. 63-65.

1.Inteligência artificial. 2. Engenharia de software. 3. Anomalias de software. 4. Bad smells. 5. Algoritmo genético. 6. Classificação. I. Título.

CDU: 004.8

UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL
Programa de Pós-Graduação em Modelagem Computacional do Conhecimento
Avenida Lourival Melo Mota, Km 14, Bloco 09, Cidade Universitária
CEP 57.072-900 – Maceió – AL – Brasil
Telefone: (082) 3214-1364

Membros da Comissão Julgadora da Dissertação de Mestrado de Lucas Benevides Viana de Amorim, intitulada: "Um método para descoberta automática de regras para a detecção de bad smells", apresentada ao Programa de Pós-Graduação em Modelagem Computacional de Conhecimento da Universidade Federal de Alagoas em 05 de maio de 2014, às 14h00min, na sala de aula do Mestrado em Modelagem Computacional de Conhecimento da UFAL.

COMISSÃO JULGADORA

Prof. Dr. Evandro de Barros Costa
UFAL – Instituto de Computação
Orientador

Prof. Dr. Baldoino Fonseca dos Santos Neto
UFAL – Instituto de Computação
Coorientador

Prof. Dr. Márcio de Medeiros Ribeiro
UFAL – Instituto de Computação
Examinador

Prof. Dr. Patrick Henrique da Silva Brito
UFAL – Instituto de Computação
Examinador

Prof. Dr. Alessandro Fabricio Garcia
PUC-Rio – Departamento de Informática
Examinador

Maceió, maio de 2014.

Resumo

Uma das técnicas para a manutenção da qualidade de um software é o refatoramento do código, mas para que esta prática traga benefícios, é necessário saber em que partes do código ela deve ser aplicada. Um catálogo com os problemas estruturais mais comuns (Bad Smells) foi proposto na literatura como uma maneira de saber quando um fragmento de código deve ser refatorado, e que tipo de refatoramento deve ser aplicado. Este catálogo vem sendo estendido por outros pesquisadores. No entanto, a detecção desses Bad Smells, está longe de ser trivial, principalmente devido a falta de uma definição precisa e consensual de cada Bad Smell.

Neste trabalho de pesquisa, propomos uma solução para o problema da detecção automática de Bad Smells por meio da descoberta automática de regras baseadas em métricas de software. Para avaliar a efetividade da técnica, utilizamos um conjunto de dados com informações sobre métricas de software calculadas para 4 sistemas de software de código aberto programados em Java (ArgoUML, Eclipse, Mylyn e Rhino) e, por meio de um algoritmo classificador, indutor de Árvores de Decisão, C5.0, fomos capazes de gerar regras para a detecção dos 12 Bad Smells analisados em nossos estudos. Nossos experimentos demonstram que regras geradas obtiveram um resultado bastante satisfatório quando testadas em um conjunto de dados à parte (conjunto de testes).

Além disso, visando otimizar o desempenho da solução proposta, implementamos um Algoritmo Genético para pré-selecionar as métricas de software mais informativas para cada Bad Smell e mostramos que é possível diminuir o erro de classificação além de, muitas vezes, reduzir o tamanho das regras geradas. Em comparação com ferramentas existentes para detecção de Bad Smells, mostramos indícios de que a técnica proposta apresenta vantagens.

Abstract

One of the techniques to maintain software quality is code refactoring. But to take advantage of code refactoring, one must know where in code it must be applied. A catalog of bad smells in code has been proposed in the literature as a way to know when a certain piece of code should be refactored and what kind of refactoring should be applied. This catalog has been extended by other researchers. However, detecting such bad smells is far from trivial, mainly because of the lack of a precise and consensual definition of each Bad Smell.

In this research work, we propose a solution to the problem of automatic detection of Bad Smells by means of the automatic discovery of metrics based rules. In order to evaluate the effectiveness of the technique, we used a dataset containing information on software metrics calculated for 4 open source software systems written in Java (ArgoUML, Eclipse, Mylyn and Rhino) and, by means of a Decision Tree induction algorithm, C5.0, we were capable of generating rules for the detection of the 12 Bad Smells that were analyzed in our study. Our experiments show that the generate rules performed very satisfactorily when tested against a separated test dataset.

Furthermore, aiming to optimize the proposed approach, a Genetic Algorithm was implemented to preselect the most informative software metrics for each Bad Smell and we show that it is possible to reduce classification error in addition to, many times, reduce the size of the generated rules. When compared to existing Bad Smells detection tools, we show evidence that the proposed technique has advantages.

Agradecimentos

"It always seems impossible until its done."
Nelson Mandela (1918-2013)

Agradeço primeiramente à minha família, por ter me dado todo apoio que precisei durante os anos de minha formação acadêmica, desde minha graduação até a finalização deste trabalho de mestrado. Agradeço especialmente à minha esposa, Ana Paula, que nestes últimos meses precisou de uma sobredose de paciência enquanto, eu passava nossos preciosos fins de semana juntos, debruçado sobre artigos, códigos e tabelas. Agradeço-lhe também por ter feito a minha jornada muito mais fácil, com sua agradável e doce companhia.

Aos meus orientadores, Professores Evandro e Baldoino, não só pela eficiente e amistosa orientação, como também, por terem muitas vezes acelerado o desenvolvimento do meu trabalho não só com cobranças, mas também colaborando com várias tardes de discussão e de "mão na massa", nas quais também tiveram importantes participações os professores Mário Hozano e Márcio Ribeiro, e todos os outros membros do grupo de pesquisa.

Não posso deixar de agradecer as professores externos que deram valiosas contribuições para o desenvolvimento deste trabalho: Prof. Aluízio F. R. Araújo, do Centro de Informática da Universidade Federal de Pernambuco e ao Prof. Yann-Gaël Guéhéneuc, da École Polytechnique de Montréal, Québec, Canadá.

Agradeço a todos os professores do PPGMCC, pelo empenho e preocupação com uma formação de qualidade e aos meus colegas de mestrado e amigos, pela companhia e pelo incentivo, tanto na fase da dissertação, como enquanto pagávamos as disciplinas e passávamos as madrugadas resolvendo exercícios ou preparando seminários. Devo também um agradecimento especial aos colegas de trabalho, tanto do IC - UFAL, quanto do DTI - IFAL, pelo incentivo e compreensão em muitos momentos.

Lucas Amorim

Sumário

1 Introdução **1**
1.1 Contexto e Problema da Pesquisa 1
1.2 Objetivos 3
1.3 Questões de Pesquisa 3
1.4 Organização da Dissertação 4

2 Fundamentação **5**
2.1 Bad Smells 5
2.2 Métricas de Software 6
2.3 Classificação 7
2.3.1 Indução de Árvore de Decisão 8
2.4 Seleção de Atributos com Algoritmo Genético 12

3 Problema **15**

4 Proposta **19**

5 Experimentos e Resultados **22**
5.1 Planejamento 22
5.2 Seleção da Amostra 23
5.3 Experimentos 25
5.3.1 Experimento 1 25
5.3.2 Experimento 2 26
5.3.3 Experimento 3 28
5.3.4 Experimento 4 29

6 Discussão **39**
6.1 Respostas às Questões de Pesquisa 39
6.1.1 Questão 1 39
6.1.2 Questão 2 40
6.1.3 Questão 3 42
6.1.4 Questão 4 43
6.2 Benefícios da Técnica Proposta 45
6.3 Ameaças à Validade 46

7 Trabalhos Relacionados **49**
7.1 Técnicas tradicionais 49
7.2 Técnicas formais 51
7.3 Técnicas inteligentes 53

8 Conclusão **55**

Referências **56**

Apêndices **61**

A Avaliação do Modelo Classificador **62**

Lista de Figuras

2.1 Representação de um modelo classificador. 8
2.2 Árvore de Decisão que determina se uma dada instância de dados pertence à classe "A"ou à classe "B"de acordo com os valores de atributos desta instância. 9
2.3 Fluxograma que representa o funcionamento de um Algoritmo Genético. . . . 13

3.1 Exemplo de uma classe onde o Bad Smell Blob foi detectado. 16
3.2 Exemplo de uma classe onde o Bad Smell Blob **não** foi detectado. 17

4.1 Representação de uma Árvore de Decisão para detectar o Bad Smell "Swiss Army Knife." 20
4.2 Representação gráfica da proposta. O Fluxograma mostra como o algoritmo classificador é integrado na função de aptidão do Algoritmo Genético 21

5.1 Esquema do conjunto de dados utilizado. 24
5.2 Comparação dos valores de Precision para cada Bad Smell com todas as métricas e somente com as métricas pré-selecionadas. 27
5.3 Comparação dos valores de Recall para cada Bad Smell com todas as métricas e somente com as métricas pré-selecionadas. 28
5.4 Comparação dos valores de Kappa para cada Bad Smell com todas as métricas e somente com as métricas pré-selecionadas. 29
5.5 Comparação dos tamanhos médios das Árvores de Decisão geradas para cada Bad Smell com todas as métricas e somente com as métricas pré-selecionadas. 30

7.1 *Rule card* do algoritmo de detecção do Bad Smell "Spaghetti Code"(SC) expresso na DSL definida em [29]. 51
7.2 Algoritmo para detecção do Bad Smell Parâmetro Obsoleto expresso em LMP, segundo Tourwé and Mens [45]. 52
7.3 Algoritmo para detecção do Bad Smell "Refused Bequest"em OCL, como definido em Kim et al. [20]. 53

Lista de Tabelas

2.1 Exemplo de conjunto de treinamento (a) e conjunto de teste (b), com atributos contínuos (Atrib1 e Atrib 2) e discreto (Atrib 3). 8
2.2 Conjunto de teste após rotulado através da classificação utilizando a árvore de decisão da Figura 2.2. 9

3.1 Valores calculados para algumas métricas analisando as duas classes do exemplo. 17

5.1 Resultados do Experimento 1. 25
5.2 Resultados do Experimento 2. 27
5.3 Configuração dos pares dos conjuntos de treinamento e teste para a realização do Experimento 3. Cada linha da tabela representa uma configuração possível. 31
5.4 Resultados do Experimento 3 utilizando as classes do projeto **ArgoUML** como conjunto de treinamento. 33
5.5 Resultados do Experimento 3 utilizando as classes do projeto **Eclipse** como conjunto de treinamento. 34
5.6 Resultados do Experimento 3 utilizando as classes do projeto **Mylyn** como conjunto de treinamento. 35
5.7 Resultados do Experimento 3 utilizando as classes do projeto **Rhino** como conjunto de treinamento. 36
5.8 Equivalência entre os nomes originais dos Bad Smells detectados pelas ferramentas e os nomes dos Bad Smells presentes no conjunto de dados. 37
5.9 Projetos cujos códigos foram analisados por cada ferramenta. 37
5.10 Resultados da detecção de Bad Smells utilizando a ferramenta **inCode**. 37
5.11 Resultados da detecção de Bad Smells utilizando a ferramenta **inFusion**. 37
5.12 Resultados da detecção de Bad Smells utilizando a ferramenta **iPlasma**. 38
5.13 Resultados da detecção de Bad Smells utilizando a ferramenta **PMD**. 38

6.1 Distribuição por categorias dos valores de Kappa do Experimento 1. 39
6.2 Média das diferenças e p-valor obtido a partir do teste T de Student para cada parâmetro. 41
6.3 Média dos parâmetros resultantes do Experimento 3, para cada Conjunto de Treinamento. 42
6.4 Média dos valores de Kappa resultantes do Experimento 4, para cada ferramenta, por projeto. 45
6.5 Média dos valores de Kappa resultantes do Experimento 4, para cada Bad Smell, por projeto. 48

A.1 Matriz de confusão para um problema com duas classes. Adaptado de [42]. . . 62
A.2 Interpretação dos valores de Kappa de acordo com Landis and Koch [22]. 64

Capítulo 1

Introdução

1.1 Contexto e Problema da Pesquisa

O processo de desenvolvimento de software tem evoluído bastante desde seu surgimento, na época dos primeiros computadores eletrônicos com arquitetura de Von Neumann, no final dos anos 40. Nos anos que se seguiram, a medida que computadores eram projetados cada vez mais com o foco na ideia de uma máquina de múltiplos propósitos, o processo de desenvolvimento de software mostrou-se de fundamental importância para permitir a aplicabilidade da computação nos diversos ramos da tecnologia que se desenvolvia no século 20.

No entanto, com uma presença cada vez maior do software no cotidiano das pessoas e especialmente em tarefas críticas (com exemplos desde controle de aeronaves a equipamentos médicos), a preocupação com a qualidade do software tem, desde então, despertado interesse da indústria e da academia. Esta é uma das principais preocupações da área de Engenharia de Software, que é a disciplina da engenharia que se preocupa com todos os aspectos envolvidos na produção de software [40].

Embora a Engenharia de Software fomente a ideia de que o software deve ser primeiro projetado e só então desenvolvido, e desde o início, baseado numa abordagem sistemática e disciplinada, muitas vezes é necessário alterar o projeto do software mesmo em estágio avançado de seu desenvolvimento. Isto ocorre principalmente porque ao longo do tempo, o código tende a ser modificado, com adição de novas funcionalidades, previstas ou não no projeto inicial, ou com correção de problemas e outras tarefas de manutenção. A medida que essas mudanças ocorrem, o código tende a distanciar-se do projeto inicial e sua integridade, qualidade e facilidade de efetuar novas manutenções podem vir a ficar comprometidas.

Uma das técnicas para a manutenção da qualidade de software em um cenário como este, é o refatoramento do código, que é definido como uma técnica disciplinada para restruturar um corpo de código existente, alterando sua estrutura interna sem mudar seu comportamento externo [8]. Mas para que o refatoramento de código traga benefícios, é necessário

saber em que partes do código ele deve ser aplicado.

Para isso, muitos desenvolvedores inspecionam visualmente o código em busca de partes que possam trazer problemas no futuro. No entanto, a percepção destas partes propensas a erro é, talvez, muito subjetiva e limitada à experiência, conhecimento e intuição do desenvolvedor. Porém, se for possível utilizar algum tipo de catálogo com os problemas estruturais mais comuns em códigos, aqui chamados de Bad Smells,[1], tal procedimento pode ser uma boa maneira de decidir quando um refatoramento deve ser aplicado. Tal catálogo de Bad Smells em código foi proposto por Fowler et al. [8] como uma maneira de saber quando um fragmento de código deve ser refatorado e, mais além, que tipo de refatoramento deve ser empregado.

Além disso, estudos mostram que a presença de Bad Smells no código está relacionada a diversos aspectos como manutenibilidade [33], integridade arquitetural [24], além de susceptibilidade a falhas [19]. É portanto notória a relevância da detecção dos Bad Smells durante o desenvolvimento de um projeto de software, pois pode ajudar a manter a qualidade ao longo do desenvolvimento do software e evitar manutenções desnecessárias no futuro, diminuindo a propensão a falhas e reduzindo os custos de desenvolvimento.

Contudo, a identificação destes Bad Smells é, por si só, uma tarefa complexa (especialmente em códigos muito grandes), já que estes são definidos, em geral, de maneira conceitual sem a possibilidade de associação direta dessas definições com valores quantitativos, como por exemplo, métricas de software (tal como linhas de códigos, números de métodos e etc). O próprio trabalho de Fowler et al. [8] afirma que não há um conjunto de métricas que permita detectar Bad Smells em código tão bem quanto a intuição humana. É evidente, entretanto, que em projetos com grandes quantidades de linhas de código, o esforço humano faria desta tarefa de detecção algo impraticável.

Mesmo com os problemas de imprecisão na definição dos Bad Smells, uma das principais vertentes na pesquisa relacionada à sua detecção, tem sido justamente a criação de regras baseadas em métricas [28, 30, 23, 31] que permitam a automatização e a escalabilidade desta tarefa. No entanto, a maioria das abordagens, apesar de proverem meios que permitem a detecção automática, ainda necessitam de alguns passos manuais para definir parâmetros das regras de detecção.

O problema que pretendemos tratar neste trabalho de pesquisa é a automatização não somente da detecção em si, mas também da definição de regras eficientes de detecção de Bad Smells específicas para o projeto de software onde serão aplicadas.

[1] O termo Bad Smell, originalmente criado por Kent Beck [8] refere-se a estruturas no código que sugerem a possibilidade ou mesmo a necessidade de refatoramento.

1.2 Objetivos

O foco deste trabalho é a detecção de Bad Smells. Mais especificamente, a descoberta automática de regras baseadas em métricas de software, para realizar esta detecção em um determinado projeto de software que se pretenda analisar.

- **Objetivo geral**
  - Propor um método para a descoberta automática de regras para a detecção de Bad Smells em software.
- **Objetivos específicos**
  - Avaliar a efetividade de um algoritmo de classificação baseado em Indução de Árvores de Decisão em detectar Bad Smells.
  - Investigar a possibilidade de melhorar a eficiência da classificação por meio da pré-seleção dos atributos utilizados.
  - Verificar se as regras geradas em para projetos específicos podem ser generalizadas ou não.
  - Comparar o desempenho das regras geradas por nossa abordagem com o resultado da detecção realizada por ferramentas existentes.

1.3 Questões de Pesquisa

Os objetivos específicos deste trabalho serão tratados como questões de pesquisa às quais iremos responder ao longo do mesmo. É necessário, portanto, formulá-las da seguinte maneira:

Q1: O algoritmo de classificação baseado em Indução de Árvores de Decisão (C5.0) é efetivo em detectar Bad Smells?

Q2: É possível melhorar a eficiência da classificação pré-selecionando as métricas com um Algoritmo Genético?

Q3: Regras de detecção de Bad Smells aprendidas em um projeto de software preservam sua qualidade quando aplicadas a outros projetos?

Q4: Como o desempenho das regras geradas pelo C5.0 se compara ao desempenho de ferramentas existentes para a detecção de Bad Smells?

1.4 Organização da Dissertação

Além deste capítulo introdutório, esta dissertação contém mais sete capítulos organizados da seguinte maneira: No Capítulo 2, Fundamentação, exploramos alguns assuntos necessários para a compreensão do contexto deste trabalho. O capítulo de fundamentação é composto por quatro seções onde discutiremos Bad Smells, com foco nos Bad Smells analisados neste trabalho; Métricas de Software, também com foco nas métricas utilizadas neste trabalho; Classificação, onde fundamentaremos principalmente a técnica de Indução de Árvore de Decisão, que é a utilizada em nossa proposta; E por fim uma seção sobre Seleção de Atributos, com foco na utilização de um Algoritmo Genético para esta tarefa.

No Capítulo 3, discutimos o nosso problema de pesquisa por meio de um exemplo real de Bad Smells em classes do conjunto de dados utilizado, enfatizando a dificuldade de detecção pela simples inspeção visual do código ou das métricas de software. No Capítulo 4, apresentamos a nossa proposta de solução ao problema descrito. Em seguida, nos Capítulos 5 e 6 realizamos a experimentação e a discussão, respectivamente, necessárias para a avaliação da proposta desta dissertação. Na sequência, no Capítulo 7, fazemos uma revisão da literatura relacionada a este trabalho e, por fim, apresentamos uma discussão de nossas conclusões no Capítulo 8.

Capítulo 2

Fundamentação

Neste capítulo trataremos de alguns temas que servem de base para a compreensão da abordagem proposta. Na Seção 2.1 definiremos o conceito de Bad Smell, e descreveremos brevemente aqueles que serão abordados neste trabalho.

Na Seção 2.2, serão apresentadas as listas das métricas de software que utilizamos em nosso conjunto de dados. Em seguida, na Seção 2.3 discutimos a tarefa de classificação, do ponto de vista de seu funcionamento como uma técnica de aprendizagem de máquina para mineração de dados, dando maior foco na técnica de Indução de Árvore de Decisão.

Finalmente, na Seção 2.4, apresentamos a técnica de seleção de atributos com Algoritmo Genética, cuja compreensão é essencial para entender como foi realizada a otimização do modelo de detecção de Bad Smells proposto.

2.1 Bad Smells

O termo Bad Smell, originalmente criado por Kent Beck [8], refere-se a estruturas no código que sugerem a possibilidade ou mesmo a necessidade de refatoramento. Cada Bad Smell pode ser visto como um indicador que aponta a violação de princípios de projeto orientado a objetos, tais como abstração de dados, encapsulamento, modularidade e hierarquia[39].

Nesta seção iremos descrever brevemente os 12 Bad Smells analisados neste trabalho. Como exposto na Seção 5.2, o conjunto de dados utilizado neste trabalho é derivado de um outro trabalho [18] que utiliza este subconjunto de 12 Bad Smells por eles terem sido relativamente bem descritos por Brown et al. [3]. Estas descrições dos Bad Smells, são baseadas naquela encontrada em [18], dado que as anotações quanto a existência dos Bad Smells, presentes no conjunto de dados utilizado, são provenientes deste trabalho e, portanto, foram realizadas levando em consideração estas descrições:

Anti Singleton: Uma classe que fornece variáveis de classe mutáveis, que consequentemente poderiam ser usadas como variáveis globais.

Blob: Uma classe muito grande e insuficientemente coesa, que monopoliza maior parte de processamento, toma a maior parte das decisões e está associada com classes que apresentam muitos campos e poucos métodos (Data Class).

Class Data Should Be Private (CDSBP): Uma classe que expõe seus campos, violando o princípio do encapsulamento.

Complex Class: Uma classe que tem pelo menos um método grande e complexo, em termos de complexidade ciclomática e linhas de código.

Large Class: Uma classe que tem pelo menos um método grande, em termos de linhas de código.

Lazy Class: Uma classe que tem poucos campos e métodos (com baixa complexidade).

Long Method: Uma classe que tem, pelo menos, um método demasiadamente grande, em termos de linhas de código.

Long Parameter List (LPL): Uma classe que tem pelo menos um método com uma lista de parâmetros relativamente muito longa, se comparada à média do número de parâmetros por método no sistema.

Message Chain: Uma classe que usa uma longa cadeia de invocação de métodos para realizar pelo menos uma de suas funcionalidades.

Refused Parent Bequest (RPB): Uma classe que redefine um método herdado por um método vazio, quebrando o polimorfismo.

Speculative Generality (SG): Uma classe que é definida como abstrata mas tem muito poucas classes filhas, as quais não fazem uso de seus métodos.

Swiss Army Knife: Uma classe cujos métodos podem ser divididos em um conjunto disjunto de muitos métodos, portanto provendo muitas funcionalidades não relacionadas.

2.2 Métricas de Software

Métricas de software são medidas de propriedades específicas de fragmentos de código fonte de um software, tal como número de linhas de código, número de métodos de uma classe ou número de parâmetros de um método. As métricas de software utilizadas neste trabalho, tal como descrito na Seção 5.2, são oriundas das ferramentas CKJM [14] e POM [9], que são ferramentas capazes de calcular os valores da métrica tendo como entrada o código fonte do projeto de software.

Das métricas que utilizamos, 18 foram calculadas pela ferramenta CKJM e 44 métricas calculadas pela ferramenta POM. Abaixo apresentamos uma lista dessas métricas. [1]

Métricas calculadas com a ferramenta CKJM: WMC, DIT, NOC, CBO, RFC, LCOM, Ca, Ce, NPM, LCOM3, LOC, DAM, MOA, MFA, CAM, IC, CBM, AMC.

Métricas calculadas com a ferramenta POM: ACAIC, ACMIC, ANA, AID, CLD, DCAEC, DCC, DCMEC, DIT, DSC, ICHClass, IR, LCOM1, LCOM2, LCOM5, LOC, MLOCsum, McCabe, NAD, NADExtended, NCM, NMA, NMD, NMDExtended, NMI, NMO, NOA, NOC, NOD, NOF, NOH, NOM, NOP, NOPM, NOParam, NOTC, NOTI, RFC_New, SIX, VGsum, WMC1, WMC_New, cohesionAttributes, connectivity.

2.3 Classificação

Classificação é uma tarefa de mineração de dados e aprendizagem de máquina cujo objetivo é atribuir objetos (i.e. instâncias de dados) a uma de várias categorias predefinidas. Por exemplo, a classificação de galáxias entre elípticas e espirais, a classificação de pacientes quanto ao diagnóstico de uma doença entre positivo e negativo de acordo com resultados de exames, e a classificação de partes do código de um software quanto a presença ou não de um determinado Bad Smell.

[42] define a tarefa de classificação, mais formalmente, da seguinte maneira: Classificação é a tarefa de aprender uma **função alvo** f que mapeia cada conjunto de atributos x em uma das categorias predefinidas y.

A função alvo f é também conhecida como um modelo classificador. Que pode ter um propósito descritivo (quando o modelo serve como uma ferramenta de distinção entre objetos de diferentes classes) ou preditivo (quando o modelo serve para predizer a classe de instâncias cuja classe ainda é desconhecida).

O fato das categorias serem predefinidas, fazem da tarefa de classificação uma tarefa de aprendizagem supervisionada, diferenciando-a da tarefa de agrupamento, que é uma tarefa de aprendizagem não-supervisionada, onde as categorias não são predefinidas e o algoritmo precisa encontrar a forma como os dados podem ser agrupados mais naturalmente.

Há também um outro aspecto importante da tarefa de classificação: O atributo alvo y, ou seja, a classe, precisa ser um atributo discreto. Esta é a característica chave que a diferencia da tarefa de regressão, uma outra tarefa de predição onde o atributo alvo é contínuo.

Dentre os algoritmos de classificação conhecidos estão os algoritmos de Indução de Árvore de Decisão, Redes Bayesianas, k-Vizinhos Mais Próximos (ou *KNN: k-Nearest Neighbor*),

[1] A definições das métricas podem ser encontradas nos respectivos sites das ferramentas:
http://wiki.ptidej.net/doku.php?id=pom#computing_metrics
http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/metric.html

Redes Neurais, e Máquinas de Vetores de Suporte. Cada uma dessas técnicas aplicam um algoritmo de aprendizagem para identificar o modelo que melhor satisfaz a relação entre o conjunto de atributos de entrada e o atributo alvo da classificação nos dados de entrada, em outras palavras, o modelo com a menor taxa de erro de classificação.

A Figura 2.1 representa o funcionamento de um modelo classificador, que tem como entrada um conjunto de treinamento, que consiste de um conjunto de amostras (ou instâncias) de dados onde a classe já é conhecida (ver Tabela 2.1a). A partir desse conjunto de dados, o processo de aprendizagem induz um modelo classificador que em seguida é testado junto a um conjunto de testes, que consiste de um conjunto de amostras cujas classes são ocultadas (ver Tabela 2.1b) e precisam ser preditas a partir do modelo.

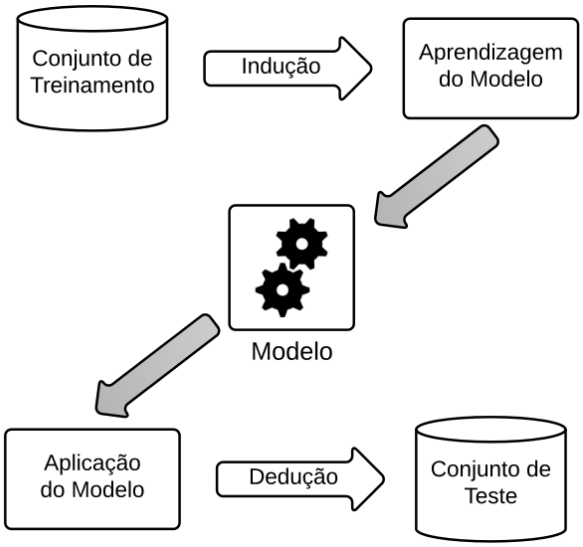

Figura 2.1: Representação de um modelo classificador.

Atrib1	Atrib2	Atrib3	Classe
28.7	-5.0	Verdadeiro	A
35.5	12.0	Falso	B
33.2	-0.5	Falso	A
33.8	8.0	Verdadeiro	B

(a)

Atrib1	Atrib2	Atrib3	Classe
11.0	-4.0	Falso	?
11.8	45.3	Verdadeiro	?
35.0	0.0	Falso	?
11.2	15.7	Verdadeiro	?

(b)

Tabela 2.1: Exemplo de conjunto de treinamento (a) e conjunto de teste (b), com atributos contínuos (Atrib1 e Atrib 2) e discreto (Atrib 3).

2.3.1 Indução de Árvore de Decisão

Indução de Árvore de Decisão é uma técnica de geração de modelos estatísticos que utilizam treinamento supervisionado para classificação e predição dos dados. Ou seja, no conjunto de treinamento as classes de cada instância são conhecidas e a informação da classe de cada

instância deste conjunto será levada em conta para a aprendizagem do modelo de classificação, que neste caso é representado por uma árvore de decisão.

Uma árvore de decisão possui uma estrutura de árvore, onde cada nó interno (não-folha), pode ser entendido como um atributo de teste, e cada nó-folha (nó-terminal) possui um rótulo de classe [10]. O nó de mais alto nível numa árvore de decisão é chamado de nó-raiz. Um exemplo de árvore de decisão, que poderia ser gerado a partir da aprendizagem com o conjunto de treinamento da Tabela 2.1(a) pode ser visto na Figura 2.2.

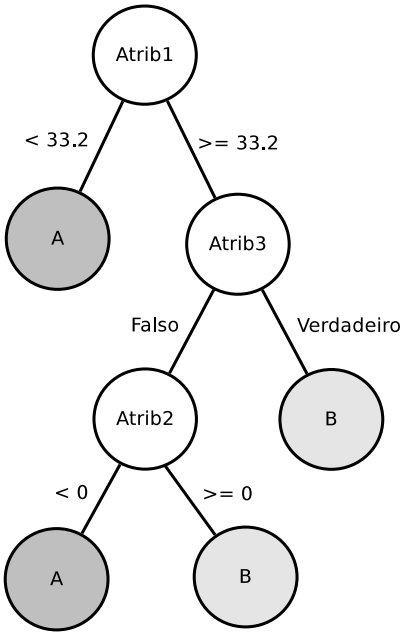

Figura 2.2: Árvore de Decisão que determina se uma dada instância de dados pertence à classe "A"ou à classe "B"de acordo com os valores de atributos desta instância.

Após aprendido os parâmetros do modelo, a árvore de decisão irá classificar uma instância cuja a classe ainda é desconhecida, de acordo com o caminho que satisfizer as condições desde o nó-raiz até o nó-folha, ao final do processo a instância será rotulada de acordo com o nó-folha. Por exemplo, utilizando a árvore da Figura 2.2 podemos agora classificar o conjunto da Tabela 2.1(b) obtendo o resultado mostrado na Tabela 2.2.

Atrib1	Atrib2	Atrib3	Classe
11.0	-4.0	Falso	A
11.8	45.3	Verdadeiro	A
35.0	0.0	Falso	B
11.2	15.7	Verdadeiro	A

(b)

Tabela 2.2: Conjunto de teste após rotulado através da classificação utilizando a árvore de decisão da Figura 2.2.

Nesta árvore de decisão, podemos ver que o atributo Atrib1 foi escolhido como o nó raiz, e os nós folhas, são as classes "A"e "B". Se tomarmos como exemplo a terceira amostra

do conjunto da Tabela 2.2, onde Atrib1 = 35.0, Atrib2 = 0.0 e Atrib3 = Falso, veremos que no primeiro ramo, o atributo Atrib1 é testado e a amostra satisfaz o ramo da direita, em seguida, o atributo Atrib3 é testado e a amostra em questão satisfaz o ramo da esquerda, finalmente, o atributo Atrib2 é testado e a amostra é classificada pelo ramo da direita na classe "B".

Como vimos, entender o significado e utilizar uma árvore de decisão para classificar uma amostra é uma tarefa trivial, no entanto, construir uma árvore a partir do conjunto de dados de treinamento, pode ser uma tarefa muito complexa. Inclusive, construir uma árvore de decisão ótima é um problema classificado como NP completo, portanto, heurísticas que visem construir árvores próximas da otimalidade tem sido um campo de estudo de teóricos [35].

É consenso que o atributo que melhor divide o conjunto de dados deve ser o nó raiz, pois este pode ser entendido como o atributo mais informativo. Existem várias medidas usadas para determinar tal atributo [42], tais como o ganho de informação (ou Entropia) [11], o índice de Gini [1], e o erro de classificação.

Estas medidas procuram capturar o grau de impureza dos nós filhos, de forma que quanto menor o grau de impureza, mais assimétrica é a divisão. Um nó com uma distribuição de classes uniforme (50% de cada classe) tem grau de impureza máximo, já um nó com uma distribuição assimétrica (Ex.: Todos da mesma classe) tem grau de impureza zero[42].

Seja $p(i|t)$ a fração de amostras que pertencem a classe i em um dado nó t, o grau de impureza de um nó pode ser avaliado das seguintes formas, de acordo com cada medida:

$$\text{Entropia}(t) = -\sum_{i=0}^{c-1} p(i|t)\log_2 p(i|t)$$

$$\text{Gini}(t) = 1 - \sum_{i=0}^{c-1} [p(i|t)]^2$$

$$\text{Erro de Classificação}(t) = 1 - \max_i [p(i|t)]$$

De acordo com Tan et al. [42], para determinar o melhor atributo para uma divisão, compara-se o grau de impureza do nó pai com o grau de impureza dos filhos. Faz-se isso para todos os atributos possíveis até encontrar aquele que resultará na maior diferença entre os graus de impureza do pai e dos filhos. Esta diferença é conhecida por ganho. Quando a medida de impureza utilizada é a entropia, o ganho é chamado de Δ_{info}, conhecido por **ganho de informação**.

Do exposto podemos compreender a razão de se procurar por atributos que dividam o conjunto de dados de maneira mais assimétrica: Estes atributos são mais informativos, ou seja, ocasionam maior ganho de informação, uma vez que a assimetria da divisão atesta que há uma probabilidade maior das amostras pertencerem ao ramo onde a fração $p(i|t)$ (que por sinal, também pode ser compreendida como uma probabilidade) é maior. Em contra partida, uma divisão perfeitamente simétrica é indecisiva, ou seja não adiciona informação ao modelo, pois cada amostra terá 50% de probabilidade de pertencer a qualquer um dos

dois ramos.

Um algoritmo básico para indução de árvore de decisão, que trabalha com o conceito de ganho de informação e é a base de muitos algoritmos de indução de árvore de decisão existentes, tais como ID3, C4.5, C5.0 e CART, é o **Algoritmo de Hunt**. [35] apresenta o Algoritmo de Hunt da seguinte maneira:

Algoritmo 1 Algoritmo de Hunt

1: Cria um nó N;
2: **SE** as amostras forem todas da mesma classe C, **ENTÃO**
3: Retorna N como um nó terminal rotulado com a classe C.
4: **SE** a lista-de-atributos estiver vazia, **ENTÃO**
5: Retorna N como um nó terminal rotulado com a classe mais comum entre as amostras.

6: Seleciona o atributo-de-teste: Aquele com o maior Δ_{info} na lista de atributos;
7: Rotula o nó N com o atributo-de-teste;
8: **PARA CADA** valor conhecido a_i do atributo-de-teste:
9: Cria um ramo partindo do nó N para a condição atributo_de_teste = a_i;
10: Seja s_i o conjunto de amostras para as quais atributo-de-teste = a_i;
11: **SE** s_i estiver vazio, **ENTÃO**
12: Conecta um nó terminal rotulado com a classe mais comum entre as amostras.
13: **SENÃO** Conecta o nó retornado por este algoritmo executado recursivamente no nó filho.

O Algoritmo de Hunt, nesta forma original, possui algumas limitações e vai funcionar bem somente se todas as combinações de valores de atributos estiverem presentes nos dados de treinamento, de forma que o modelo gerado seja completo, e além disso, cada combinação deve possuir um rótulo de classe único. Na prática, estas condições são difíceis de atingir, e alguns procedimentos extras são tomados para aproximar o modelo [42].

Um problema que ocorre com algoritmos que envolvem aprendizagem de máquina para gerar modelos classificadores, não exclusivamente com algoritmos de indução de árvore de decisão, é o *overfitting* do modelo nos dados de treinamento. Ou seja, o modelo passa a ter um erro baixíssimo nos dados de treinamento e um erro alto no conjunto de dados inteiro. Mais formalmente, [35] define que o *overfitting* ocorre quando existe uma outra hipótese h_2 que tem maior erro do que h (hipótese/modelo gerada pelo algoritmo) quando testada nos dados de treinamento, mas um erro menor que h quando testada no conjunto de dados inteiro.

Existem várias técnicas para simplificar árvores de decisão e evitar o problema do *overfitting* [2]. Duas abordagens comuns são: i) Parar o algoritmo de treinamento antes que alcance o ponto onde o erro nos dados de treinamento torna-se zero. ii) Podar a árvore de decisão e verificar se possui a mesma precisão de predição do que a árvore original, se sim,

a árvore menor é escolhida.

Os algoritmos de árvore de decisão mais populares são o C4.5 [37], C5.0 [38] e o CART [1]. O algoritmo C4.5, gera um modelo de classificação utilizando o conceito de entropia da informação, para decidir o próximo nó da árvore, conforme descrito acima [37]. O algoritmo C5.0 é uma melhoria do algoritmo C4.5 que promete regras mais precisas, árvores de decisão menores e outras melhorias relacionadas à eficiência e ao custo computacional do algoritmo em si.

2.4 Seleção de Atributos com Algoritmo Genético

A tarefa de seleção de atributos é muito importante quando se trabalha com conjunto de dados com alta dimensionalidade, ou seja, com grande número de atributos, o que aumenta o custo computacional da tarefa de classificação, onde comumente trabalha-se apenas com um subconjunto do conjunto de atributos original. Os atributos contidos no subconjunto são escolhidos de acordo com a informação que deseja extrair.

A escolha pode ser manual, por um especialista no domínio dos dados, ou automática, por algum algoritmo de seleção automática de atributos. Com o exposto anteriormente, na Seção 2.3, podemos ver que a escolha correta dos atributos que farão parte deste subconjunto pode influenciar bastante na qualidade do modelo de classificação gerado.

A técnica de Algoritmo Genético é uma das utilizadas para seleção de atributos, iremos focar esta seção apenas nesta técnica pois é a que implementamos em nossa proposta. Por ser uma técnica de busca e otimização de solução desenvolvida para atuar em grandes espaços de busca, a técnica é bastante adequada para a otimização do subconjunto de atributos mais importantes para tarefa de Classificação que virá em seguida.

A Figura 2.3 apresenta um fluxograma que representa o funcionamento de um Algoritmo Genético, em seguida descrevemos cada etapa instanciando-as para o problema específico de seleção de atributos.

Representação da Solução: Num Algoritmo Genético, as soluções são representadas por cromossomos artificiais, cujas características (genes) serão alteradas ao longo dos ciclos do laço principal do algoritmo (gerações). Para a tarefa de seleção de atributos, essa estrutura geralmente é implementada por meio de um vetor binário, onde cada bit (gene) indica se um dado atributo será considerado pelo algoritmo de classificação ou não.

Geração da População Inicial: Esta etapa pode ser realizada por meio da criação de sequências aleatórias de bits para preencher os n cromossomos da população. Ou seja, seleciona-se aleatoriamente os atributos que farão parte de cada cromossomo. Alternativamente, a população inicial pode ser alguma solução existente, possivelmente definida por um especialista, que se pretenda melhorar.

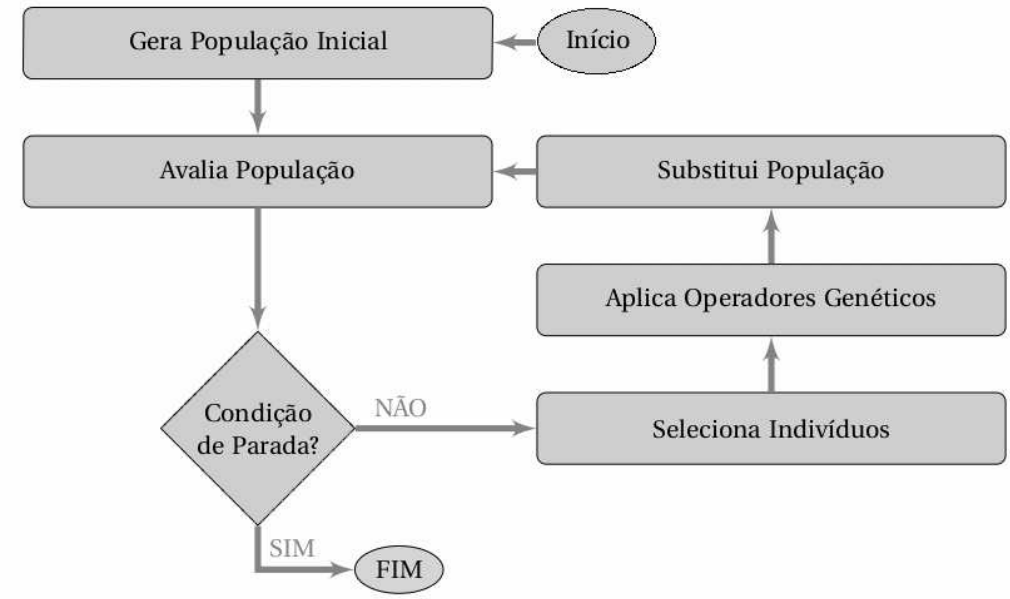

Figura 2.3: Fluxograma que representa o funcionamento de um Algoritmo Genético.

Avaliação da População: Executa o algoritmo de classificação para cada cromossomo (solução), gerando e testando o modelo, obtendo assim a alguma medida relacionada à precisão de classificação deste modelo. A partir deste retorno, é calculada uma nota para cada cromossomo, atribuindo notas melhores para aqueles que resultarem em melhores modelos, de forma que objetivo do Algoritmo Genético é maximizar a eficácia da classificação.

Condição de parada: Geralmente a condição de parada é alcançar uma quantidade grande de gerações (ciclos do laço principal do algoritmo). Pode-se também elaborar uma condição de parada relacionada à qualidade da solução obtida, parando apenas quando atingir um determinado valor de precisão, por exemplo.

Seleção de indivíduos: A seleção pode ocorrer de forma elitista, onde seleciona-se os $n/2$ melhores indivíduos da população gerada é selecionada para formar a próxima população, ou pode utilizar alguma técnica que inclua um fator de aleatoriedade de forma a evitar uma convergência pré-matura do algoritmo. Uma dessas técnicas é a da seleção por roleta, onde um sorteio é realizado e os indivíduos mais aptos (aqueles que possuem as maiores avaliações) têm maiores chances de serem selecionados.

Operadores Genéticos: O operador de cruzamento é responsável por combinar dois cromossomos (duas soluções) gerando cromossomos filhos como resultado. O tipo de cruzamento mais comum é o de um ponto fixo, onde divide-se os cromossomos em uma determinada posição e recombina-se as partes. O operador de mutação é responsável por alterar as características dos genes, que neste caso, significa inverter um bit, removendo ou adicionando uma determinada métrica. Em sua implementação mais

comum, o operador de mutação percorre cada bit do vetor alterando-os de acordo com uma determinada probabilidade de mutação.

Substituição da população: É a etapa responsável pela renovação da população antiga, utilizando os indivíduos mais novos.

Vários parâmetros de um Algoritmo Genético podem ser configurados de acordo com características específicas do problema, incluindo os valores das probabilidades de mutação e cruzamento, tamanho da população e condição de parada. Para o problema de seleção de atributos, pode-se determinar também, por exemplo, o número máximo de atributos que deverá ser selecionado, o que pode ser útil quando o conjunto de dados a serem classificados for muito grande e demandar grande custo computacional. Devido à flexibilidade, a facilidade de representação da solução, e o bom desempenho em encontrar boas soluções em um curto período de tempo, os Algoritmos Genéticos são frequentemente escolhidos para esta tarefa.

Capítulo 3

Problema

Uma das técnicas para a manutenção da qualidade do software é o refatoramento do código, que é definido como uma técnica disciplinada para restruturar um corpo de código existente, alterando sua estrutura interna sem mudar seu comportamento externo [8]. Mas para que o refatoramento de código traga benefícios, é necessário saber quando ele deve ser aplicado. Para isso, muitos desenvolvedores inspecionam visualmente o código em busca de partes que possam trazer problemas no futuro, no entanto, a percepção destas partes propensas a erro é, talvez, muito subjetiva e limitada à experiência, conhecimento e intuição do desenvolvedor. Entretanto, se for possível utilizar algum tipo de catálogo com os problemas estruturais mais comuns em códigos, a.k.a Bad Smells, esta pode ser uma boa maneira de decidir quando um refatoramento deve ser aplicado.

Tal catálogo de Bad Smells em código foi proposto por Fowler et al. [8] como uma maneira de saber quando uma certa peça de código deve ser refatorada e, mais além, que tipo de refatoramento deve ser aplicado. Um exemplo ilustrativo de um Bad Smell, em particular, o Blob ou God Class, é mostrado na Figura 3.1. Blobs ocorrem quando muita funcionalidade é atribuída a uma única classe. Frequentemente, um Blob é também uma classe muito grande e com muitas associações a outras classes.

O diagrama UML da Figura 3.1 representa a classe AbstractFilePersister do projeto ArgoUML, que por sinal está presente no conjunto de dados utilizado neste trabalho de pesquisa, tal como descrito na Seção 5.2. Esta classe, neste conjunto de dados, está anotada como uma classe que apresenta o Bad Smell Blob, e não é muito difícil de acreditar nesta suposição, uma vez que pode-se ver pelo diagrama que a classe é responsável por muitas operações além de estar associada a muitas outras classes.

Mas se olharmos para a Figura 3.2, que representa a classe UMLComboBoxEntry, do mesmo projeto, podemos nos perguntar porque esta segunda classe também não foi detectada como um Blob. É por isso que é necessária uma análise mais profunda para classificar uma classe como Blob ou não. Esta análise poderia ser realizada por meio de uma inspeção visual minuciosa do código de cada classe além de levar em conta o contexto onde a mesma

Figura 3.1: Exemplo de uma classe onde o Bad Smell Blob foi detectado.

se insere, o que faz desta tarefa manual de detectar Bad Smells algo muito trabalhoso.

Projetos pequenos, com cerca de uma dúzia de classes, podem ser manualmente analisado afim de detectar a presença de cada Bad Smell, permitindo que o desenvolvedor realize as técnicas de refatoramento apropriadas para cada situação na maior parte do tempo. Mas à medida em que o código cresce torna-se cada vez mais trabalhoso e difícil realizar esta inspeção manual, se levado em conta o código como um todo. Em um projeto como o Eclipse (uma IDE com código aberto), por exemplo, com mais de 10000 classes e 3,5 milhões de linhas de código, tal tarefa torna-se quase inviável.

Para reduzir o incômodo de ter que analisar minuciosamente, visualmente, o código para decidir se uma certa classe possui um determinado Bad Smell, pode-se analisar as métricas (ver Seção 2.2) calculadas para a classe em questão obtendo assim uma visão global do código ou uma análise mais objetiva. Para fins de exemplificação, a Tabela 3.1 mostra algumas das métricas calculadas [1] para as duas classes mostradas no exemplo desta seção. Ao olhar para tabela, é possível notar algumas diferenças entre as classes, mas ainda não está tão claro porque uma classe é considerada um Blob e a outra não.

Embora os Bad Smells sejam uma melhor indicação de quando o código deve ser refatorado do que simplesmente analisar a estética do código, Fowler et al. [8] presumem que

[1] NOM: Número de Métodos, NOF: Número de Campos (atributos), RFC: Número de métodos desta classe somados com aqueles das outras entidades que são invocadas por esta. NMDextended: Número de Métodos Declarados pela entidade e suas entidades membro, AMC: Complexidade Média dos Métodos, CE: Quantidade de classes usadas por esta classe específica, LOC: Linhas de Código.

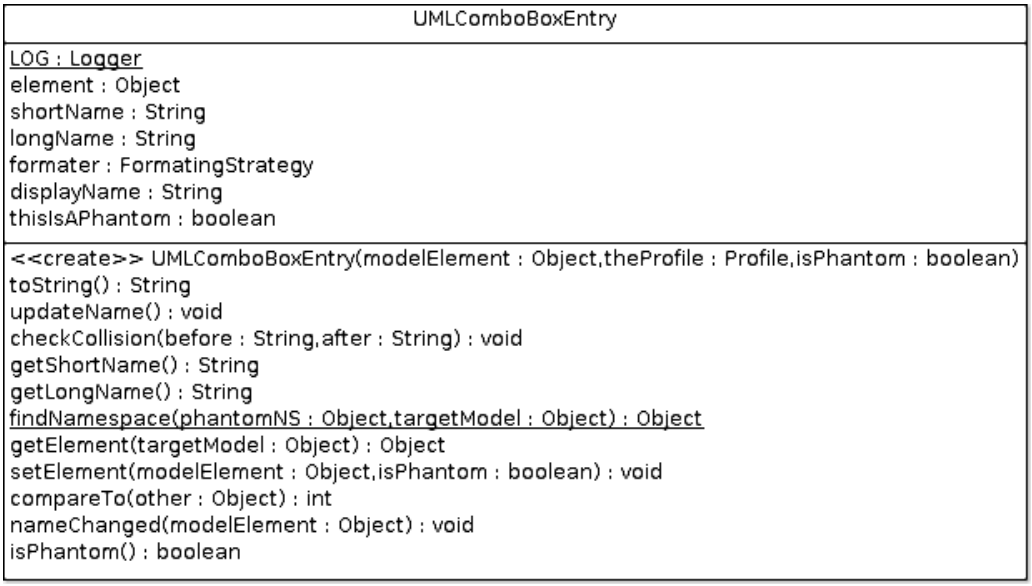

Figura 3.2: Exemplo de uma classe onde o Bad Smell Blob **não** foi detectado.

Métrica	**AbstractFilePersister**	**UMLComboBoxEntry**
NOM	26	13
NOF	4	7
RFC	73	47
NMDextended	34	13
AMC	13	27.7
CE	8	8
LOC	368	381

Tabela 3.1: Valores calculados para algumas métricas analisando as duas classes do exemplo.

não há um conjunto de métricas que possa dizer a um desenvolvedor quando existe um Bad Smell, melhor do que a intuição humana. Assim, não há atualmente uma definição precisa de cada Bad Smell que permita que os mesmos sejam detectados automaticamente com precisão.

Entretanto, muitas técnicas têm sido propostas visando a detecção automática de Bad Smells [27, 28, 30, 39, 29, 4, 45, 36, 20], mas a maioria destas abordagens dependem de regras, baseadas em métricas, para a detecção de cada Bad Smell. Estas regras são, frequentemente, uma expressão lógica comparando métricas previamente selecionadas com limiares predefinidos. A escolha das métricas e a definição destes limiares é geralmente baseada na experiência da equipe de desenvolvimento e frequentemente resultado de um extenso processo de tentativa e erro. Com isso, a eficiência da detecção vai depender bastante do quão boa é a regra proposta.

Além da ineficiência que se presume que teria uma regra gerada manualmente, a abordagem descrita acima também precisa de intenso esforço humano e portanto ainda não é escalável. Para lidar com o problema da definição de regras que tenham um bom desempe-

nho com pouco ou nenhum esforço humano, alguns pesquisadores têm proposto técnicas que geram automaticamente estas regras de detecção de Bad Smells, geralmente utilizando uma algoritmo de Programação Genética [17, 25, 34] resultando em regras parecidas ou do mesmo tipo das geradas pelas abordagens manuais.

Este tipo de abordagem automática depende da aprendizagem das regras por meio de exemplos de classes previamente anotados, (ou seja, rotulados com verdadeiro ou falso para o Bad Smell em questão) em seguida essas regras são melhoradas de maneira incremental por meio da abordagem evolutiva do algoritmo. Embora a Programação Genética tenha se mostrado como um forte candidato para esta tarefa, ainda existe a necessidade de experimentação com outras técnicas de aprendizagem de máquina.

Capítulo 4

Proposta

Neste trabalho propomos a utilização do algoritmo de classificação C5.0 (ver Seção 2.3 para a tarefa de geração automática das regras de detecção de Bad Smells. Por ser um algoritmo gerador de Árvore de Decisão, o C5.0 gera, como regras de detecção, Árvores de Decisão que representam o modelo de classificação aprendido por meio da informação contida no conjunto de treinamento.

Uma das grandes vantagens da abordagem por Indução de Árvore de Decisão é que, ao contrário de outras abordagens de aprendizagem que geram modelos "caixa preta", esta gera um modelo que pode ser inspecionado facilmente pelo desenvolvedor ou engenheiro de software do projeto que está sendo analisado, podendo ser validado, melhorado, por meio de uma pré-seleção manual de métricas, por exemplo, além de discutido e compreendido pelos membros da equipe de desenvolvimento. Escolhemos especificamente o algoritmo C5.0 por ser o mais avançado da família dos algoritmos de Indução de Árvore de Decisão.

Em nossa proposta, utilizamos um conjunto de dados contendo diversas métricas calculadas para cada classe dos projetos estudados além da informação sobre a existência ou não de cada Bad Smell em cada uma dessas classes. Dessa forma, o Algoritmo C5.0 é capaz de gerar Árvores de Decisão como a do exemplo da Figura 4.1 através da qual é possível identificar se uma determinada classe pertence ou não um determinado Bad Smell, bastando para isso obter os valores das métricas de software para a classe em questão.

Nota-se, pela Figura 4.1 que os nós da árvore são métricas de software. O algoritmo C5.0 tende a colocar no topo da árvore as métricas mais significativas para a detecção do Bad Smell analisado, e as métricas menos informativas vão aparecendo a medida em que a árvore vai se aprofundando e se especializando. No entanto, através de alguns experimentos preliminares, detectamos que ao fornecer para o classificador C5.0 o valor de todas as métricas presentes no conjunto de dados, nem sempre as mais importantes são escolhidas para a construção da árvore. Frequentemente, em nossos testes, foi possível reajustar o conjunto de métricas e obter uma árvore mais precisa e/ou menor.

Para abordar, portanto, o problema de encontrar o conjunto mais adequado de métri-

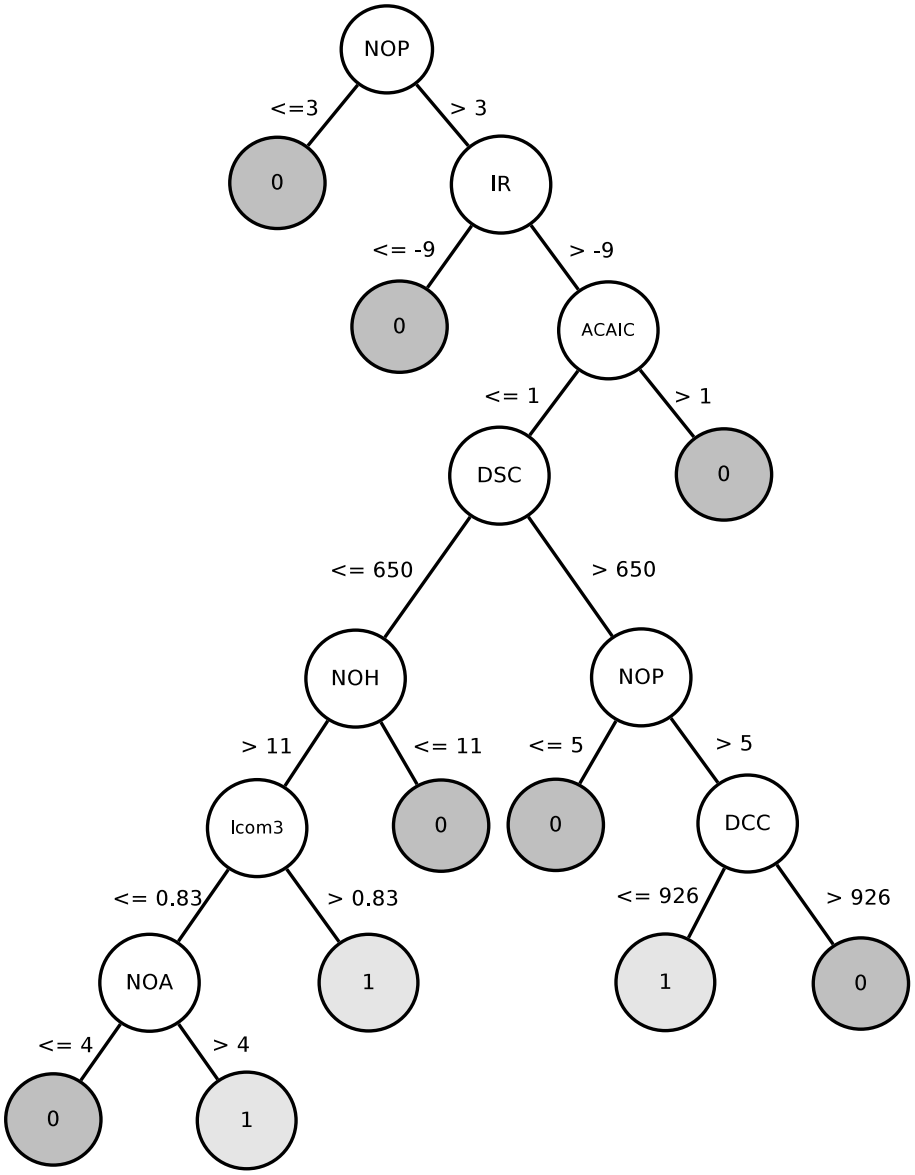

Figura 4.1: Representação de uma Árvore de Decisão para detectar o Bad Smell "Swiss Army Knife."

cas para gerar árvores de decisão melhores, propomos a utilização de um Algoritmo Genético para pré-selecionar estas métricas. Para isso, nesta implementação, a solução (cromossomo), isto é, o conjunto de métricas pré-selecionadas, é representada por um vetor binário, cujo tamanho é o número total de métricas, no qual cada bit representa se uma determinada métrica de software será utilizada ou não pelo classificador. Esta solução é modificada, de maneira evolutiva, a cada novo ciclo do Algoritmo Genético, em busca de melhores subconjuntos de métricas, por tanto, a cada ciclo o C5.0 é executado para gerar o novo modelo cuja a precisão é aferida para determinar o quão bom é cada subconjunto de métricas. A Figura 4.2 mostra como integramos o classificador C5.0 dentro do Algoritmo Genético.

Inicialmente, uma população inicial contendo soluções aleatórias é criada (cada métrica tem 50% de ser selecionada para estas primeiras soluções).

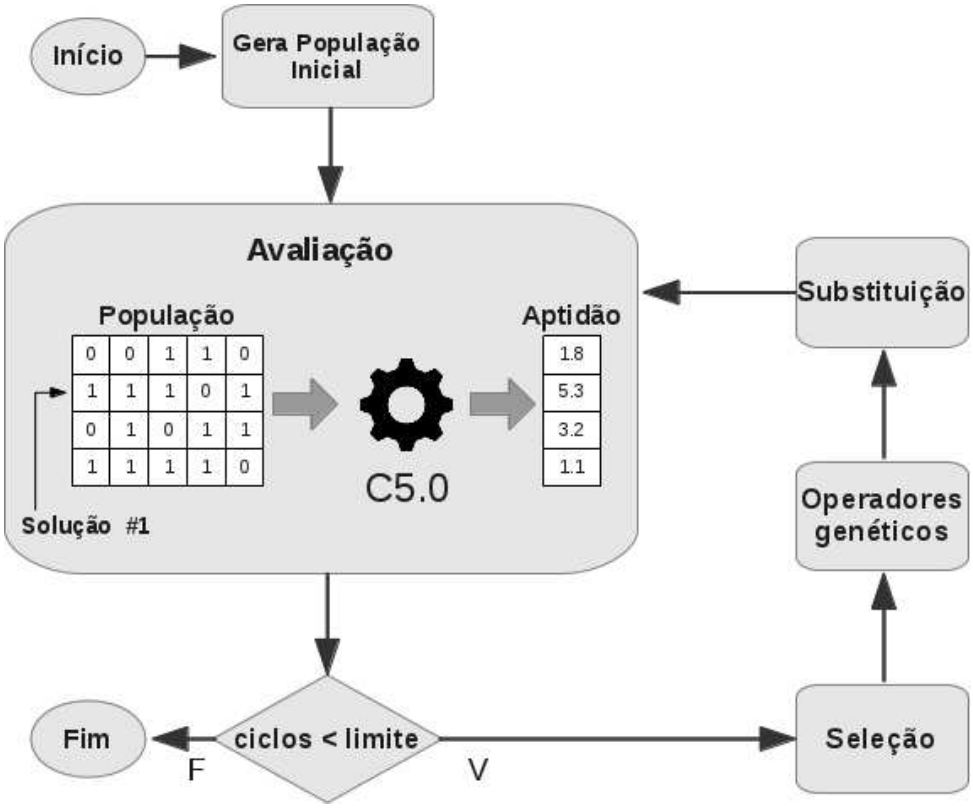

Figura 4.2: Representação gráfica da proposta. O Fluxograma mostra como o algoritmo classificador é integrado na função de aptidão do Algoritmo Genético

Em seguida, cada uma dessas soluções é avaliada através da execução do classificador com o subconjunto de métricas representado pela solução. Esta avaliação serve para atribuir um valor de aptidão a cada uma das soluções. A aptidão de cada solução é inversamente proporcional ao erro de classificação obtido com modelo gerado com o subconjunto que representa.

Após isso, se o número limite de ciclos não for atingido, os passos comuns de um Algoritmo Genético são executados: A seleção das soluções que irão originar a próxima população, os operadores genéticos (cruzamento e mutação) e finalmente a substituição da população atual pela nova, composta dessas novas soluções, que serão avaliadas no próximo passo, fechando o ciclo tal como mostrado na Figura 4.2.

Capítulo 5

Experimentos e Resultados

5.1 Planejamento

O objetivo deste estudo é avaliar a técnica proposta com relação a sua efetividade em gerar automaticamente regras para detecção de Bad Smells que tenham qualidade suficiente para atingir boa performance de classificação. Em particular, este estudo procura responder às seguintes questões de pesquisa:

Q1: O algoritmo de classificação baseado em Indução de Árvores de Decisão (C5.0) é efetivo em detectar Bad Smells?

Q2: É possível melhorar a eficiência da classificação pré-selecionando as métricas com um Algoritmo Genético?

Q3: Regras de detecção de Bad Smells aprendidas em um projeto de software preservam sua qualidade quando aplicadas a outros projetos?

Q4: Como o desempenho das regras geradas pelo C5.0 se compara ao desempenho de ferramentas existentes para a detecção de Bad Smells?

Para responder à questão de pesquisa **Q1**, utilizaremos uma medida de concordância entre modelos de classificação largamente adotada na literatura: A estatística Kappa (ver Apêndice A). Que permitirá verificar a concordância da detecção obtida pelas regras geradas pelo C5.0 com o conhecimento prévio (presente na base de dados utilizada) sobre a existência ou não de cada Bad Smell.

Para responder à questão de pesquisa **Q2**, iremos comparar os resultados da detecção de Bad Smells utilizando regras geradas pelo C5.0 recebendo como entrada todas as métricas de cada classe, com os resultados da detecção quando o C5.0 recebe apenas as métricas pré-selecionadas pelo Algoritmo Genético após vários ciclos de evolução da solução.

Para responder à questão de pesquisa **Q3**, iremos separar o conjunto de dados por projetos de software para realizar um experimento onde o conjunto de treinamento contenha

somente classes de um determinado projeto e os conjuntos de testes contendo classes de outro projeto. Os resultados poderão ser analisados individualmente, através do valor do Kappa obtido, além de comparativamente, com aqueles obtidos a partir do conjunto de dados composto por todos os projetos.

Já a questão de pesquisa **Q4**, será respondida a partir da comparação dos resultados obtidos com as regras geradas pelo C5.0 conforme o Experimento 5.3.1 com o resultado de ferramentas de detecção de Bad Smells disponíveis na literatura. Neste contexto, entendemos como desempenho, a efetividade da técnica em detectar os Bad Smells, o que será avaliado quantitativamente de acordo com métricas como *Precision*, *Recall* e *Kappa*.

5.2 Seleção da Amostra

O conjunto de dados usado nesta pesquisa consiste de dados sobre métricas de software e Bad Smells detectados em quatro projetos de software livre desenvolvidos na linguagem de programação Java: Eclipse [5], Mylyn [6], ArgoUML [44] e Rhino [32]. O projeto Eclipse consiste de um Ambiente de Desenvolvimento Integrado (IDE) para linguagem de programação Java, mas também suporta muitas outras linguagens por meio do uso de plugins. Eclipse é um projeto maduro com extenso suporte para programadores por meio de seu desenvolvimento orientado a plugins. O projeto Eclipse principal tem mais de 3,5 MLOC (milhão de linhas de código) distribuídas ao longo de mais de 10.000 classes.

Mylyn é um plugin para Eclipse projetado para ajudar desenvolvedores a lidar com o problema de sobrecarga de informação ao trabalhar com projetos muito grandes. Ao invés de mostrar a hierarquia completa de classes e arquivos, Mylyn tenta mostrar apenas a informação que é relevante para a tarefa corrente, tal como bug reports, novas funcionalidades e tarefas a fazer (to-do). O projeto Mylyn também é desenvolvido pela comunidade do Eclipse e tem mais de 200KLOCs e mais de 1500 classes.

O projeto ArgoUML é uma ferramenta de modelagem largamente utilizada que inclui suporte para todos os diagramas UML (Linguagem de Modelagem Unificada) e tem aproximadamente 300KLOCs e aproximadamente 2000 classes. Rhino é o menor projeto analisado nesta pesquisa, mas também é bastante maduro. Trata-se de uma implementação do JavaScript (utilizado no Mozilla Firefox [7], por exemplo) cujo desenvolvimento é gerenciado pela Mozilla Foundation, ele tem aproximadamente 70KLOCs e 200 classes. A versão dos projetos analisados foram: Eclipse versão 3.3.1, também conhecido como Europa, Mylyn versão 3.1.1, ArgoUML versão 0.26 e Rhino versão 1.6R6.

A informação sobre os Bad Smells presentes em cada classe das versões acima dos quatro projetos foi derivada da base de dados utilizada em [19], fornecida pelos pesquisadores do grupo de pesquisa Ptidej através do site do grupo [43], composta de vários conjuntos de dados, um para cada versão dos quatro projetos descritos acima, contendo informação sobre

a presença ou não de cada tipo de Bad Smell. Este conjunto de dados contém informação sobre os 12 Bad Smells considerados neste estudo: Antisingleton, Blob, Class Data Should Be Private, Complex Class, Large Class, Lazy Class, Long Method, Long Parameter List, Message Chains, Refused Parent Bequest, Speculative Generality, e Swiss Army Knife. Estes Bad Smells foram descritos na Seção 2.1 deste trabalho.

Para obter as métricas das classes, nós utilizamos duas ferramentas de extração de métricas: CKJM [14] e POM [9], ambas ferramentas são capazes de calcular métricas analisando os arquivos .jar dos projetos. Os arquivos .jar analisados, correspondentes às versões apontadas acima, foram obtidos das páginas de download de cada projeto. A ferramenta CKJM foi utilizada para calcular 18 métricas[1], e a ferramenta POM foi utilizada para calcular outras 44 métricas[2] para cada classe contida nos arquivos .jar, resultando em 62 métricas de software. Embora algumas métricas apareçam de forma repetida no conjunto final, já que foram calculadas pelas duas ferramentas, achamos necessário manter as duas versões, uma vez que são calculadas de maneira diferente por cada ferramenta.

Finalmente, para criar um único conjunto de dados adequado para a utilização nestes experimentos, o conjunto de dados sobre os Bad Smells utilizado em [9] foi unido aos conjuntos correspondentes contendo as 18 métricas calculadas pela ferramenta CKJM, e as 44 métricas calculadas pela ferramenta POM, usando como chave de busca para a união o nome da classe. Após isso, os conjuntos de dados resultantes (um para cada projeto de software) foram concatenados e as linhas repetidas foram removidas deixando somente uma linha para cada nome de classe. Algumas classes encontradas no conjunto de dados de Bad Smells não foram encontradas na saída do CKJM ou POM e vice-versa e portanto precisaram ser removidas. Foram consideradas para este estudo, portanto, somente as 7952 classes presentes em ambos os conjuntos de dados. O esquema final do conjunto de dados está mostrado na Figura 5.1.

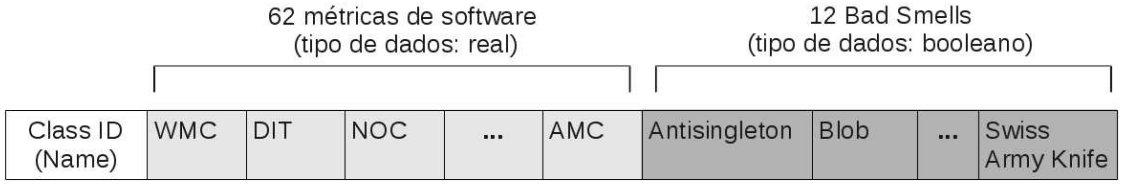

Figura 5.1: Esquema do conjunto de dados utilizado.

[1]Métricas CKJM: WMC, DIT, NOC, CBO, RFC, LCOM, Ca, Ce, NPM, LCOM3, LOC, DAM, MOA, MFA, CAM, IC, CBM e AMC.

[2]Métricas POM: ACAIC, ACMIC, AID, CLD, DCAEC, DCC, DCMEC, DIT, DSC, ICHClass, IR, LCOM1, LCOM2, LCOM5, LOC, MLOCsum, McCabe, NAD, NADExtended, NCM, NMA, NMD, NMDExtended, NMI, NMO, NOA, NOC, NOD, NOF, NOH, NOM, NOP, NOPM, NOParam, NOTC, NOTI, RFC_New, SIX, VGsum, WMC1, WMC_New, cohesionAttributes e connectivity.

5.3 Experimentos

Para cada um dos 12 Bad Smells selecionados para este estudo, três experimentos foram realizados: O primeiro experimento procura responder a questão de pesquisa **Q1**, o segundo experimento concentra-se na questão de pesquisa **Q2** e o terceiro na questão de pesquisa **Q3**. Além desses 3 experimentos, um quarto experimento foi realizado para comparar os resultados obtidos com as regras geradas pela técnica proposta neste trabalho com resultados de ferramentas presentes na literatura, respondendo a questão de pesquisa **Q4**.

5.3.1 Experimento 1

O primeiro experimento usou apenas o classificador C5.0, sem o Algoritmo Genético para pré-selecionar as métricas, de maneira que todas as 62 métricas foram entregues como entrada ao classificador para que este gerasse as regras (Árvores de Decisão) selecionando as métricas por seu próprio método. Neste experimento, algoritmo C5.0 foi configurado para realizar uma validação cruzada com 10-folds. A configuração do conjunto de dados utilizado neste experimento, obedece o esquema da Figura 5.1 e é composto pelas classes dos 4 projetos de softwares estudados, tal como descrito na Seção 5.2. Portanto, os conjuntos de teste e treinamento são compostos por classes dos 4 projetos.

Resultados

A Tabela 5.1 exibe o resultado do Experimento 1, realizado utilizando o C5.0 para gerar as regras (Árvores de Decisão) tendo como entrada os valores de todas as métricas presentes na base e as anotações do conjunto de treinamento. Nesta tabela, é possível visualizar, para cada um dos Bad Smells, a taxa de erro média de classificação, o tamanho médio das árvores, a quantidade de Verdadeiros Negativos, Falso Positivos, Falso Negativos, e Verdadeiro Positivos, e nas últimas três colunas, os valores de Precision, Recall e Kappa.

Tabela 5.1: Resultados do Experimento 1.

Bad Smell	**Erro (%)**	**Tam. Árvore**	**TN**	**FP**	**FN**	**TP**	**Precision**	**Recall**	**Kappa**
AntiSingleton	4.5	29.7	7201	101	256	394	79.596	60.615	0.664
Blob	9.9	47.3	6.838	181	605	328	64.440	35.155	0.406
CDSBP	6.2	57.6	6918	117	377	540	82.192	58.888	0.653
ComplexClass	2.5	34.8	7057	89	109	697	88.677	86.476	0.862
LargeClass	1.5	32.1	7622	41	79	210	83.665	72.664	0.770
LazyClass	4.5	104.8	5862	144	215	1731	92.320	88.952	0.876
LongMethod	13.8	81.1	4488	576	518	2370	80.448	82.064	0.704
LongParameterList	1.1	47.7	6450	29	56	1417	97.994	96.198	0.964
MessageChains	9.5	18.6	6417	287	468	780	73.102	62.500	0.619
RefusedParentRequest	5.9	107.7	6244	187	280	1241	86.905	81.591	0.806
SpeculativeGenerality	2	15.3	7724	64	93	71	52.593	43.293	0.465
SwissArmyKnife	0.4	10.3	7882	16	16	38	70.370	70.370	0.702

5.3.2 Experimento 2

O segundo experimento foi realizado utilizando o Algoritmo Genético para pré-selecionar as métricas, da maneira descrita na Figura 4.2, com a configuração de parâmetros variáveis do algoritmo descrita abaixo. A configuração do conjunto de dados utilizado foi exatamente a mesmo do Experimento 1.

Tamanho da Solução: 62 (O número total de atributos, neste caso, as 62 métricas).

Tipo de dados dos Genes: Booleanos/binários (indicando se uma determinada métrica será utilizada ou não).

Tamanho da população: 20 soluções.

Função de aptidão ou avaliação: O algoritmo C5.0 é executado utilizando apenas as métricas representadas na solução como presentes (bit 1). O alvo da classificação é o Bad Smell selecionado para o experimento corrente. A fim de obter uma melhor avaliação dos resultados, o algoritmo C5.0 foi configurado para realizar uma validação cruzada com 10-folds, e o erro de classificação médio das 10 execuções é considerado para o cálculo da aptidão da solução.

Número limite de ciclos: 25.

Seleção: A estratégia utilizada para a seleção das soluções foi a Seleção Baseada em Torneio, na qual 4 soluções são aleatoriamente selecionadas para uma disputa na qual aquela com a melhor aptidão é selecionada. O processo é repetido até que metade da população esteja preenchida, a outra metade é preenchida aleatoriamente com as soluções remanescentes.

Cruzamento: Foi utilizado cruzamento de um ponto (metade de um cromossomo é unido à metade de um outro cromossomo para gerar um novo indivíduo) com uma taxa de ocorrência de 50%.

Mutação: Inversão de bits com uma taxa de ocorrência de 2%.

Resultados

A Tabela 5.2 exibe o resultado do Experimento 2, realizado utilizando algoritmo C5.0 para gerar as regras (Árvores de Decisão) tendo como entrada somente os valores das métricas pré-selecionadas pelo Algoritmo Genético e as anotações do conjunto de treinamento. Esta tabela apresenta a mesma estrutura descrita para a Tabela 5.1 acima.

Tabela 5.2: Resultados do Experimento 2.

Bad Smell	Erro (%)	Tam. Árvore	TN	FP	FN	TP	Precision	Recall	Kappa
AntiSingleton	4	23.1	7225	77	242	408	84.124	62.769	0.698
Blob	9.1	38	6889	130	595	338	72.222	36.227	0.438
CDSBP	5.4	56.3	6958	77	354	563	87.969	61.396	0.694
ComplexClass	2.3	41.7	7059	87	97	709	89.070	87.965	0.872
LargeClass	1	28.8	7632	31	48	241	88.603	83.391	0.854
LazyClass	3.2	101.6	5910	96	162	1784	94.894	91.675	0.911
LongMethod	11.3	105.1	4560	504	393	2495	83.194	86.392	0.758
LongParameterList	0.6	54.3	6460	19	32	1441	98.699	97.828	0.979
MessageChains	8.8	16	6498	206	491	757	78.609	60.657	0.635
RefusedParentRequest	5.5	106	6251	180	260	1261	87.509	82.906	0.817
SpeculativeGenerality	1.6	9.2	7751	37	94	70	65.421	42.683	0.509
SwissArmyKnife	0.3	9.2	7882	16	11	43	72.881	79.630	0.759

Comparativo dos resultados dos Experimentos 1 e 2

Para permitir a comparação dos resultados dos dois experimentos acima, necessária para a resposta à questão de pesquisa **Q2**, exibe-se abaixo uma série de gráficos de barras comparando os valores de Precision (Figura 5.2), Recall (Figura 5.3), Kappa (Figura 5.4) e Tamanho médio da Árvore de Decisão (Figura 5.5), com todas as métricas presentes no conjunto de dados (resultados da Tabela 5.1) e somente com as métricas pré-selecionadas pelo Algoritmo Genético (resultados da Tabela 5.2).

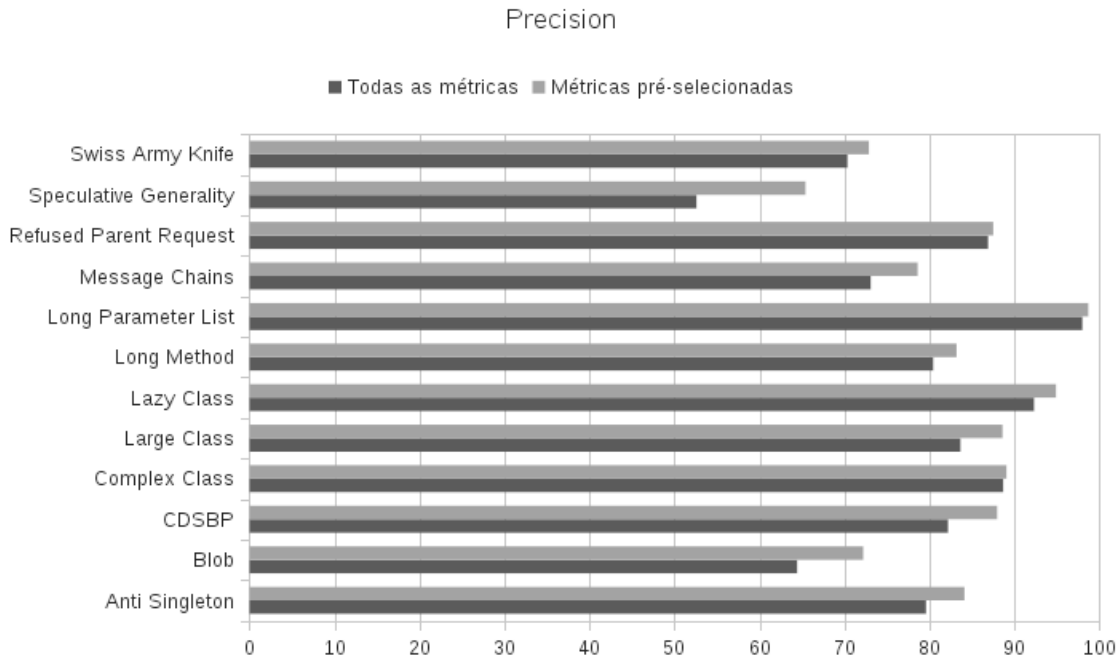

Figura 5.2: Comparação dos valores de Precision para cada Bad Smell com todas as métricas e somente com as métricas pré-selecionadas.

Figura 5.3: Comparação dos valores de Recall para cada Bad Smell com todas as métricas e somente com as métricas pré-selecionadas.

5.3.3 Experimento 3

No terceiro experimento, com o objetivo de responder a questão de pesquisa **Q3**, o conjunto de dados foi separado por projetos de software de modo que foi possível executar o C5.0 tendo como conjunto de treinamento somente classes de um mesmo projeto e, como conjunto de teste, somente classes de um outro projeto. Este experimento foi repetido de forma a contemplar todas as 12 configurações possíveis de pares de conjuntos, tal como descrito na Tabela 5.3. Para cada um desses pares, o C5.0 foi executado 12 vezes, uma para cada Bad Smell.

Resultados

As Tabelas 5.4 a 5.7 exibem os resultados do Experimento 3. Algumas células das tabelas estão marcados com um travessão, indicando que não foi possível efetuar os cálculos de Precision, Recall ou Kappa para aqueles valores de TN, FN, FP e TP devido ao fato do denominador ser nulo, isso ocorre, por exemplo, quando não é detectado nenhum positivo, seja verdadeiro positivo (TP) ou falso positivo (FP).

A Tabela 5.4 apresenta os resultados da execução do C5.0 tomando como conjunto de treinamento as classes do projeto ArgoUML e, como conjuntos de teste, as classes dos outros projetos. Esta primeira tabela cumpre o planejado nas 3 primeiras linhas da Tabela 5.3, apresentada na Seção 5.3.

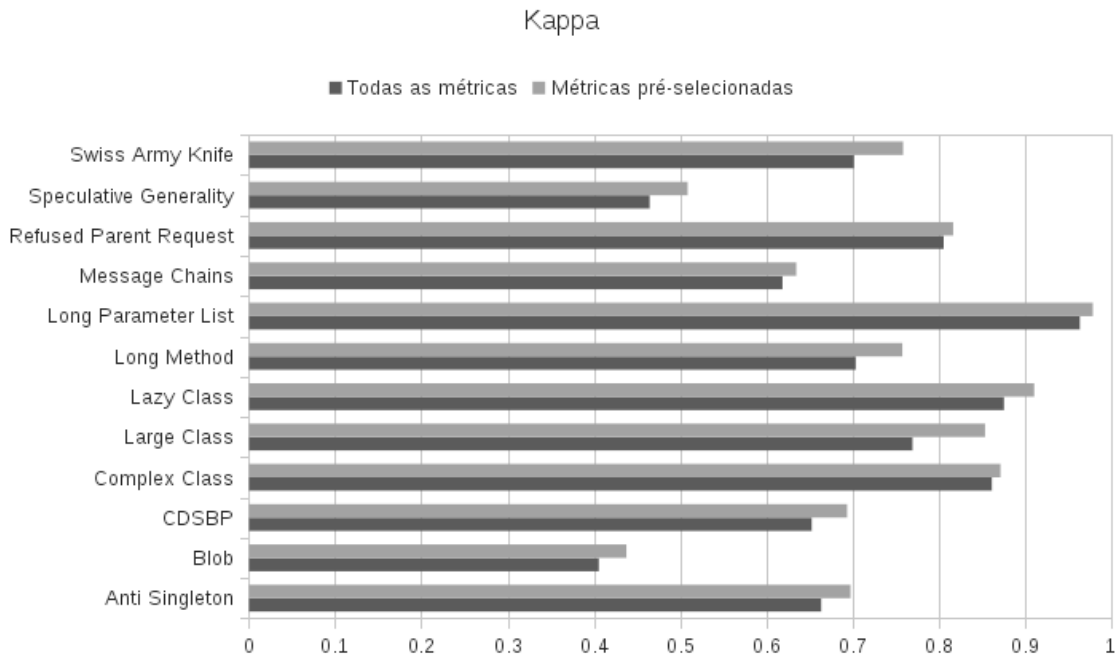

Figura 5.4: Comparação dos valores de Kappa para cada Bad Smell com todas as métricas e somente com as métricas pré-selecionadas.

A Tabela 5.5 apresenta os resultados da execução do C5.0 tomando como conjunto de treinamento as classes do projeto Eclipse e, como conjuntos de teste, as classes dos outros projetos. Esta tabela cumpre o planejado nas linhas de números 4, 5 e 6 da Tabela 5.3.

A Tabela 5.6 apresenta os resultados da execução do C5.0 tomando como conjunto de treinamento as classes do projeto Mylyn e, como conjuntos de teste, as classes dos outros projetos. Esta tabela cumpre o planejado nas linhas de números 7, 8 e 9 da Tabela 5.3.

A Tabela 5.7 apresenta os resultados da execução do C5.0 tomando como conjunto de treinamento as classes do projeto Rhino e, como conjuntos de teste, as classes dos outros projetos. Esta tabela cumpre o planejado nas linhas de números 10, 11 e 12 da Tabela 5.3.

5.3.4 Experimento 4

Para realizar a comparação dos resultados de detecção de Bad Smells das regras geradas pela técnica proposta com os resultados de outras ferramentas capazes de analisar códigos em Java, respondendo a questão de pesquisa **Q4**, realizamos o Experimento 4, utilizando as ferramentas **inCode**[12], **inFusion**[13], **iPlasma**[26] e **PMD**[41]. Para utilizar estas ferramentas, foi necessário o download dos códigos fontes dos quatro projetos, os quais serviram de entrada aos algoritmos de detecção.

As ferramentas inCode e inFusion são ambas produzidas pela empresa Intooitus e prometem a detecção automática de Bad Smells como "God Class", "Data Class", "Code Duplication", "Data Clumps"e outros aspectos. As duas ferramentas são muito parecidas, no

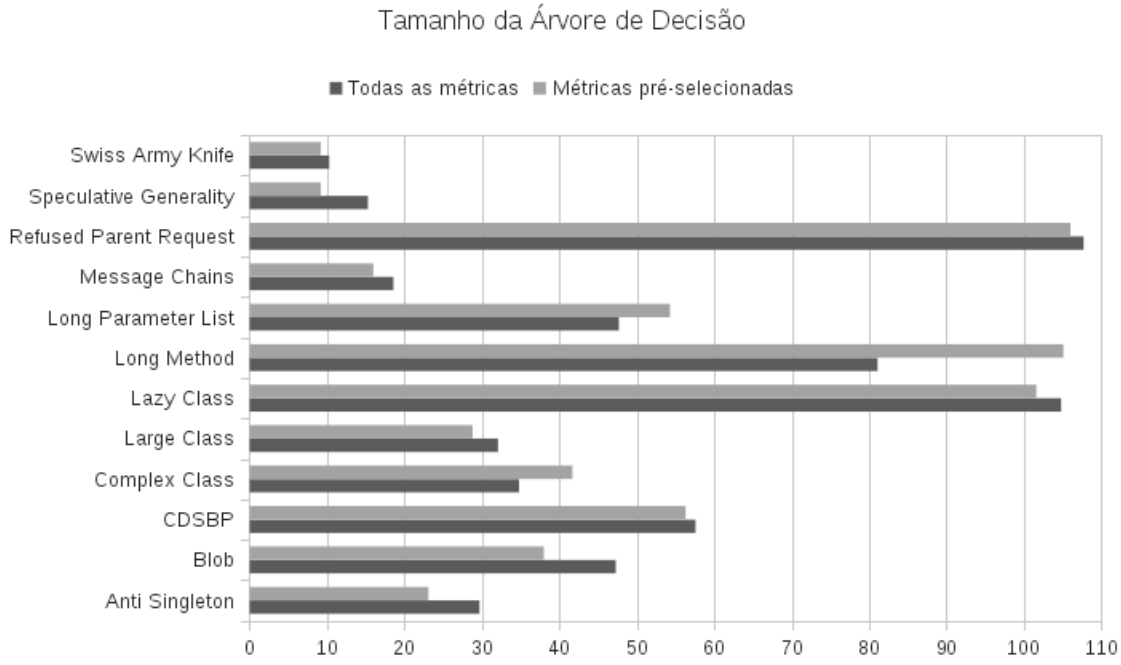

Figura 5.5: Comparação dos tamanhos médios das Árvores de Decisão geradas para cada Bad Smell com todas as métricas e somente com as métricas pré-selecionadas.

entanto, a ferramenta inCode é mais simples e pode ser considerada uma versão mais leve da ferramenta inFusion. Estas ferramentas podem ser integradas ao Eclipse e estão disponíveis comercialmente.

A ferramenta iPlasma, que tem um propósito mais acadêmico e não está disponível comercialmente, está descrita em Marinescu et al. [26] como um ambiente integrado para análise da qualidade de sistemas de software orientados a objetos. Esta ferramenta é capaz de detectar Bad Smells como "God Class", "Refused Parent Bequest"e outros. A ferramenta PMD, descrita como um "analisador de código fonte", encontra problemas de programação como variáveis não utilizadas, blocos "catch"vazios, criação desnecessárias de objetos e outros. Além, é claro, de detectar alguns Bad Smells como "God Class", "Excessive Public Count", "Excessive Parameter List"e outros.

Todas as quatro ferramentas acima baseiam-se em conjuntos de regras, baseadas em métricas de software, para detectar os Bad Smells. Estas regras são pré-definidas mas podem ser customizadas pelo desenvolvedor. Outras ferramentas como Sonar, JDeodorant e outras também foram selecionadas para a análise mas precisaram ser excluídas por motivos como o fato de não detectarem Bad Smells por si só mas dependerem de outras ferramentas que já estavam no conjunto de análise (Sonar) ou por funcionarem exclusivamente como um plugin da IDE Eclipse, dificultando a análise de alguns dos projetos da base de dados, devido à dificuldade de importá-los corretamente para o Eclipse.

A ferramenta DETEX/DECOR não foi analisada devido à dificuldade de executá-la, já

Tabela 5.3: Configuração dos pares dos conjuntos de treinamento e teste para a realização do Experimento 3. Cada linha da tabela representa uma configuração possível.

Config. Num.	Conjunto de Treinamento	Conjunto de Teste
1	ArgoUML	Eclipse
2	ArgoUML	Mylyn
3	ArgoUML	Rhino
4	Eclipse	ArgoUML
5	Eclipse	Mylyn
6	Eclipse	Rhino
7	Mylyn	ArgoUML
8	Mylyn	Eclipse
9	Mylyn	Rhino
10	Rhino	ArgoUML
11	Rhino	Eclipse
12	Rhino	Mylyn

que seus autores não disponibilizam uma versão "stand alone"da ferramenta e um passo-a-passo para sua execução.

Equivalência entre os nomes dos Bad Smells

Vários autores definem o mesmo Bad Smell com diferente nomes ou até mesmo usam um mesmo nome para identificar Bad Smells com descrições que não são totalmente idênticas. Por este fato, a tarefa de comparar os resultados das ferramentas com os resultados descritos no conjunto de dados utilizado requer uma atividade de identificação das equivalências entre os nomes dos Bad Smells descritos na base de dados e os nomes dos Bad Smells detectados pelas ferramentas.

Para tanto, foi necessário estudar a descrição dos Bad Smells tanto no site do grupo Ptidej, do qual provém o conjunto de dados, como nos sites, artigos, ou documentação das ferramentas. Como resultado desse estudo, a Tabela 5.8 apresenta as equivalências que encontramos para utilizar na comparação realizada nesse experimento.

Incompletude da comparação

Como pode-se ver pela Tabela 5.8 apenas 7 dos 12 Bad Smells presentes no conjunto de dados foram encontrados entre aqueles que são analisados por pelo menos uma das quatro ferramentas.

Além disso, como vemos na Tabela 5.9, com as ferramentas inFusion e iPlasma tivemos algumas restrições no que diz respeito aos projetos analisados. No caso da ferramenta iPlasma, não foi possível analisar o projeto Eclipse. Com a ferramenta inFusion, só foi possível analisar o projeto Rhino, o menor dos quatro projetos do conjunto de dados.

Estas restrições ocorreram porque tivemos dificuldades de executar a ferramenta inFusion e iPlasma com um número muito grande de linhas de código. No caso da ferramenta

inFusion, o limite de linhas de código para a licença gratuita para avaliação, a que tivemos acesso, era de 100KLOC, como somente o projeto Rhino está abaixo deste limite, somente este projeto foi analisado. Tentamos obter uma licença completa para uso neste estudo, o que não foi concedido pelos desenvolvedores da ferramenta.

No caso da ferramenta iPlasma, apesar de não apresentar nenhuma limitação quanto ao número máximo de linhas de código, ao executar a análise do projeto Eclipse a mesma não foi capaz de finalizar o processamento por apresentar erros relacionados a limites de memória atingidos pelo Garbage Collector da máquina virtual Java (a ferramenta iPlasma foi escrita em Java). Procuramos alternativas para resolver este problema durante a execução, porém sem sucesso.

Resutados

A Tabela 5.10 exibe os parâmetros calculados a partir do resultado da detecção de Bad Smells utilizando a ferramenta inCode. Com esta ferramenta, foi possível analisar 3 dos 12 Bad Smells do conjunto de dados do estudo: Blob, Long Parameter List, e Message Chains, nos quatro projetos de software.

A Tabela 5.11 exibe os parâmetros calculados a partir do resultado da detecção de Bad Smells utilizando a ferramenta inFusion. Com esta ferramenta, foi possível analisar 4 dos 12 Bad Smells do conjunto de dados do estudo: Blob, Long Parameter List, Message Chains e Refused Parent Bequest, no entanto, somente foi possível analisar o projeto Rhino (ver Seção 5.3.4).

A Tabela 5.12 exibe os parâmetros calculados a partir do resultado da detecção de Bad Smells utilizando a ferramenta iPlasma. Com esta ferramenta, foi possível analisar 2 dos 12 Bad Smells do conjunto de dados do estudo: Blob e Refused Parent Bequest, no entanto, não foi possível analisar o projeto Eclipse (ver Seção 5.3.4).

A Tabela 5.13 exibe os parâmetros calculados a partir do resultado da detecção de Bad Smells utilizando a ferramenta PMD. Com esta ferramenta, foi possível analisar 5 dos 12 Bad Smells do conjunto de dados do estudo: Blob, Class Data Should Be Private, Long Method, Long Parameter List e Speculative Generality nos quatro projetos de software.

Tabela 5.4: Resultados do Experimento 3 utilizando as classes do projeto **ArgoUML** como conjunto de treinamento.

Conjunto de teste	**Bad Smell**	**Erro (%)**	**Tam. da Árvore**	**TN**	**FP**	**FN**	**TP**	**Precision**	**Recall**	**Kappa**
Eclipse	Antisingleton	11.9	1	3855	0	519	0	–	0.000	0.000
Eclipse	Blob	18.1	7	3054	146	646	78	34.821	10.773	0.085
Eclipse	CDSBP	14.5	6	3688	20	613	53	72.603	7.958	0.117
Eclipse	ComplexClass	11.3	8	3076	51	443	174	77.333	28.201	0.357
Eclipse	LargeClass	6.9	10	4072	297	4	1	0.336	20.000	0.004
Eclipse	LazyClass	42.4	5	2497	0	1854	23	100.000	1.225	0.014
Eclipse	LongMethod	42.9	12	2202	14	2862	296	95.484	9.373	0.073
Eclipse	LongParameterList	25.0	5	3049	254	839	232	47.737	21.662	0.171
Eclipse	MessageChains	16.7	8	3528	6	725	115	95.041	13.690	0.201
Eclipse	RefusedParentRequest	12.9	27	3612	116	447	199	63.175	30.805	0.351
Eclipse	SpeculativeGenerality	2.3	4	4258	15	84	17	53.125	16.832	0.247
Eclipse	SwissArmyKnife	1.2	1	4320	0	54	0	–	0.000	0.000
Mylyn	Antisingleton	8.2	1	1414	0	127	0	–	0.000	0.000
Mylyn	Blob	6.7	7	1424	24	80	13	35.135	13.978	0.172
Mylyn	CDSBP	12.0	6	1353	5	180	3	37.500	1.639	0.022
Mylyn	ComplexClass	5.0	8	1451	18	59	13	41.935	18.056	0.231
Mylyn	LargeClass	7.6	10	1407	35	82	17	32.692	17.172	0.189
Mylyn	LazyClass	0.4	5	1523	0	6	12	100.000	66.667	0.798
Mylyn	LongMethod	18.8	12	1162	30	259	90	75.000	25.788	0.303
Mylyn	LongParameterList	10.3	5	1367	79	80	15	15.957	15.789	0.104
Mylyn	MessageChains	11.2	8	1364	0	173	4	100.000	2.260	0.039
Mylyn	RefusedParentRequest	17.7	27	1239	12	260	30	71.429	10.345	0.140
Mylyn	SpeculativeGenerality	2.5	4	1500	2	37	2	50.000	5.128	0.089
Mylyn	SwissArmyKnife	0.0	1	1541	0	0	0	–	–	–
Rhino	Antisingleton	0.5	1	198	0	1	0	–	0.000	0.000
Rhino	Blob	0.0	7	199	0	0	0	–	–	–
Rhino	CDSBP	11.1	6	176	6	16	1	14.286	5.882	0.035
Rhino	ComplexClass	7.0	8	185	0	14	0	–	0.000	0.000
Rhino	LargeClass	9.5	10	180	0	19	0	–	0.000	0.000
Rhino	LazyClass	4.5	5	190	0	9	0	–	0.000	0.000
Rhino	LongMethod	17.6	12	164	0	35	0	–	0.000	0.000
Rhino	LongParameterList	4.0	5	191	0	8	0	–	0.000	0.000
Rhino	MessageChains	32.7	8	134	0	65	0	–	0.000	0.000
Rhino	RefusedParentRequest	4.0	27	187	1	7	4	80.000	36.364	0.482
Rhino	SpeculativeGenerality	1.5	3	196	1	2	0	0.000	0.000	-0.007
Rhino	SwissArmyKnife	0.0	1	199	0	0	0	–	–	–

Tabela 5.5: Resultados do Experimento 3 utilizando as classes do projeto **Eclipse** como conjunto de treinamento.

Conjunto de teste	**Bad Smell**	**Erro (%)**	**Tam. da Árvore**	**TN**	**FP**	**FN**	**TP**	**Precision**	**Recall**	**Kappa**
ArgoUML	Antisingleton	4.9	33	1746	89	1	2	2.198	66.667	0.040
ArgoUML	Blob	11.3	32	1605	117	90	26	18.182	22.414	0.141
ArgoUML	CDSBP	11.3	40	1607	180	28	23	11.330	45.098	0.143
ArgoUML	ComplexClass	5.6	16	1669	66	37	66	50.000	64.078	0.532
ArgoUML	LargeClass	9.0	1	1672	0	166	0	–	0.000	0.000
ArgoUML	LazyClass	70.2	62	506	1290	0	42	3.153	100.000	0.018
ArgoUML	LongMethod	61.2	40	368	1124	0	364	24.462	100.000	0.114
ArgoUML	LongParameterList	19.2	37	1436	103	249	50	32.680	16.722	0.125
ArgoUML	MessageChains	9.1	22	1524	148	20	146	49.660	87.952	0.587
ArgoUML	RefusedParentRequest	26.8	56	1079	185	308	266	58.980	46.341	0.337
ArgoUML	SpeculativeGenerality	1.7	6	1801	15	17	5	25.000	22.727	0.229
ArgoUML	SwissArmyKnife	1.0	12	1819	19	0	0	0.000	–	0.000
Mylyn	Antisingleton	11.2	33	1328	86	86	41	32.283	32.283	0.262
Mylyn	Blob	9.0	32	1358	90	49	44	32.836	47.312	0.341
Mylyn	CDSBP	11.9	40	1287	71	113	70	49.645	38.251	0.367
Mylyn	ComplexClass	5.3	16	1390	79	2	70	46.980	97.222	0.609
Mylyn	LargeClass	6.4	1	1442	0	99	0	–	0.000	0.000
Mylyn	LazyClass	54.6	62	682	841	0	18	2.095	100.000	0.019
Mylyn	LongMethod	29.9	40	748	444	16	333	42.857	95.415	0.406
Mylyn	LongParameterList	22.4	37	1102	344	1	94	21.461	98.947	0.280
Mylyn	MessageChains	11.7	22	1267	97	83	94	49.215	53.107	0.445
Mylyn	RefusedParentRequest	16.9	56	1108	143	118	172	54.603	59.310	0.463
Mylyn	SpeculativeGenerality	3.0	6	1482	20	26	13	39.394	33.333	0.346
Mylyn	SwissArmyKnife	0.5	12	1533	8	0	0	0.000	–	0.000
Rhino	Antisingleton	5.0	33	189	9	1	0	0.000	0.000	-0.009
Rhino	Blob	18.1	32	163	36	0	0	0.000	–	0.000
Rhino	CDSBP	23.6	40	138	44	3	14	24.138	82.353	0.278
Rhino	ComplexClass	29.1	16	127	58	0	14	19.444	100.000	0.236
Rhino	LargeClass	9.5	1	180	0	19	0	–	0.000	0.000
Rhino	LazyClass	26.1	62	138	52	0	2	3.704	100.000	0.052
Rhino	LongMethod	41.2	40	85	79	3	32	28.829	91.429	0.233
Rhino	LongParameterList	60.3	37	71	120	0	8	6.250	100.000	0.045
Rhino	MessageChains	11.6	22	125	9	14	51	85.000	78.462	0.732
Rhino	RefusedParentRequest	7.5	56	176	12	3	8	40.000	72.727	0.479
Rhino	SpeculativeGenerality	1.0	6	195	2	0	2	50.000	100.000	0.662
Rhino	SwissArmyKnife	0.0	12	199	0	0	0	–	–	–

Tabela 5.6: Resultados do Experimento 3 utilizando as classes do projeto **Mylyn** como conjunto de treinamento.

Conjunto de teste	**Bad Smell**	**Erro (%)**	**Tam. da Árvore**	**TN**	**FP**	**FN**	**TP**	**Precision**	**Recall**	**Kappa**
ArgoUML	Antisingleton	4.7	6	1750	85	1	2	2.299	66.667	0.041
ArgoUML	Blob	6.2	11	1695	27	87	29	51.786	25.000	0.309
ArgoUML	CDSBP	12.4	23	1596	191	37	14	6.829	27.451	0.068
ArgoUML	ComplexClass	4.7	8	1715	20	66	37	64.912	35.922	0.440
ArgoUML	LargeClass	7.3	11	1640	32	102	64	66.667	38.554	0.452
ArgoUML	LazyClass	0.3	6	1793	3	2	40	93.023	95.238	0.940
ArgoUML	LongMethod	9.5	18	1121	71	75	274	79.420	78.510	0.728
ArgoUML	LongParameterList	15.6	3	1504	35	252	47	57.317	15.719	0.190
ArgoUML	MessageChains	8.6	12	1559	113	45	121	51.709	72.892	0.558
ArgoUML	RefusedParentRequest	16.8	50	1017	247	61	513	67.500	89.373	0.642
ArgoUML	SpeculativeGenerality	1.6	5	1803	13	17	5	27.778	22.727	0.242
ArgoUML	SwissArmyKnife	0.0	1	1838	0	0	0	–	–	–
Eclipse	Antisingleton	10.4	6	3794	61	394	125	67.204	24.085	0.312
Eclipse	Blob	15.2	11	3524	126	537	187	59.744	25.829	0.290
Eclipse	CDSBP	12.0	23	3585	123	402	264	68.217	39.640	0.439
Eclipse	ComplexClass	7.0	8	3743	14	290	327	95.894	52.998	0.647
Eclipse	LargeClass	10.4	11	3917	452	1	4	0.877	80.000	0.015
Eclipse	LazyClass	42.2	6	2497	0	1847	30	100.000	1.598	0.018
Eclipse	LongMethod	28.8	18	2103	113	1147	1011	89.947	46.849	0.420
Eclipse	LongParameterList	17.5	3	3300	3	761	310	99.042	28.945	0.379
Eclipse	MessageChains	7.7	12	3414	120	451	389	76.424	46.310	0.505
Eclipse	RefusedParentRequest	16.2	50	3251	477	231	415	46.525	64.241	0.444
Eclipse	SpeculativeGenerality	2.6	5	4223	50	65	36	41.860	35.644	0.372
Eclipse	SwissArmyKnife	1.2	1	4320	0	54	0	–	0.000	0.000
Rhino	Antisingleton	6.0	6	187	11	1	0	0.000	0.000	-0.009
Rhino	Blob	18.1	11	163	36	0	0	0.000	–	0.000
Rhino	CDSBP	14.6	23	160	22	7	10	31.250	58.824	0.334
Rhino	ComplexClass	13.6	8	158	27	0	14	34.146	100.000	0.452
Rhino	LargeClass	14.1	11	152	28	0	19	40.426	100.000	0.509
Rhino	LazyClass	4.5	6	190	0	9	0	–	0.000	0.000
Rhino	LongMethod	25.1	18	125	39	11	24	38.095	68.571	0.341
Rhino	LongParameterList	27.6	3	136	55	0	8	12.698	100.000	0.166
Rhino	MessageChains	21.1	12	132	2	40	25	92.593	38.462	0.435
Rhino	RefusedParentRequest	6.0	50	178	10	2	9	47.368	81.818	0.570
Rhino	SpeculativeGenerality	0.5	5	196	1	0	2	66.667	100.000	0.798
Rhino	SwissArmyKnife	0.0	1	199	0	0	0	–	–	–

Tabela 5.7: Resultados do Experimento 3 utilizando as classes do projeto **Rhino** como conjunto de treinamento.

Conjunto de teste	**Bad Smell**	**Erro (%)**	**Tam. da Árvore**	**TN**	**FP**	**FN**	**TP**	**Precision**	**Recall**	**Kappa**
ArgoUML	Antisingleton	0.2	1	1835	0	3	0	–	0	0
ArgoUML	Blob	6.3	1	1722	0	116	0	–	0	0
ArgoUML	CDSBP	4.8	6	1745	42	47	4	8.696	7.843	0.058
ArgoUML	ComplexClass	5.3	4	1729	6	92	11	64.706	10.68	0.17
ArgoUML	LargeClass	9.1	6	1654	18	149	17	48.571	10.241	0.142
ArgoUML	LazyClass	24.9	5	1376	420	38	4	0.943	9.524	-0.025
ArgoUML	LongMethod	21.6	13	1411	81	316	30	27.027	8.671	0.044
ArgoUML	LongParameterList	16.6	3	1533	6	299	0	0	0	-0.006
ArgoUML	MessageChains	10.3	14	1537	135	54	112	45.344	67.47	0.487
ArgoUML	RefusedParentRequest	12.0	5	1202	62	159	415	87.002	72.3	0.707
ArgoUML	SpeculativeGenerality	1.2	1	1816	0	22	0	–	0	0
ArgoUML	SwissArmyKnife	0.0	1	1838	0	0	0	–	–	–
Eclipse	Antisingleton	11.9	1	3855	0	519	0	–	0	0
Eclipse	Blob	16.6	1	3650	0	724	0	–	0	0
Eclipse	CDSBP	14.4	6	3643	65	564	102	61.078	15.315	0.196
Eclipse	ComplexClass	12.9	4	3755	2	563	54	96.429	8.752	0.14
Eclipse	LargeClass	4.0	6	4200	169	4	1	0.588	20	0.009
Eclipse	LazyClass	37.1	5	2482	15	1608	269	94.718	14.331	0.153
Eclipse	LongMethod	42.0	13	2132	84	1753	405	82.822	18.767	0.151
Eclipse	LongParameterList	23.6	3	3297	6	1025	46	88.462	4.295	0.061
Eclipse	MessageChains	13.5	14	3277	257	332	508	66.405	60.476	0.551
Eclipse	RefusedParentRequest	14.2	5	3551	177	446	200	53.05	30.96	0.317
Eclipse	SpeculativeGenerality	2.3	1	4273	0	101	0	–	0	0
Eclipse	SwissArmyKnife	1.2	1	4320	0	54	0	–	0	0
Mylyn	Antisingleton	8.2	1	1414	0	127	0	–	0	0
Mylyn	Blob	6.0	1	1448	0	93	0	–	0	0
Mylyn	CDSBP	13.2	6	1333	25	178	5	16.667	2.732	0.014
Mylyn	ComplexClass	4.2	4	1467	2	63	9	81.818	12.5	0.207
Mylyn	LargeClass	4.8	6	1433	9	65	34	79.07	34.343	0.458
Mylyn	LazyClass	14.5	5	1315	208	16	2	0.952	11.111	-0.004
Mylyn	LongMethod	21.0	13	1135	57	267	82	58.993	23.496	0.238
Mylyn	LongParameterList	5.7	3	1446	0	88	7	100	7.368	0.13
Mylyn	MessageChains	12.2	14	1291	73	115	62	45.926	35.028	0.331
Mylyn	RefusedParentRequest	16.9	5	1224	27	234	56	67.47	19.31	0.236
Mylyn	SpeculativeGenerality	2.5	1	1502	0	39	0	–	0	0
Mylyn	SwissArmyKnife	0.0	1	1541	0	0	0	–	–	–

Tabela 5.8: Equivalência entre os nomes originais dos Bad Smells detectados pelas ferramentas e os nomes dos Bad Smells presentes no conjunto de dados.

Ferramenta	**Nome original**	**Nome no conj. de dados**
inCode	God Class	Blob
inCode	Data Clumps	Long Parameter List
inCode	Message Chains	Message Chains
inFusion	God Class	Blob
inFusion	Data Clumps	Long Parameter List
inFusion	Message Chains	Message Chains
inFusion	Refused Parent Bequest	Refused Parent Bequest
iPlasma	God Class	Blob
iPlasma	Refused Parent Bequest	Refused Parent Bequest
PMD	God Class	Blob
PMD	Excessive Public Count	Class Data Should Be Private
PMD	Excessive Method Length	Long Method
PMD	Excessive Parameter List	Long Parameter List
PMD	Abstract Class Without Any Method	Speculative Generality

Tabela 5.9: Projetos cujos códigos foram analisados por cada ferramenta.

	Projetos			
Ferramenta	**ArgoUML**	**Eclipse**	**Mylyn**	**Rhino**
inCode	X	X	X	X
inFusion				X
iPlasma	X		X	X
PMD	X	X	X	X

Tabela 5.10: Resultados da detecção de Bad Smells utilizando a ferramenta **inCode**.

Bad Smell	**Projeto**	**TN**	**FP**	**FN**	**TP**	**Precision**	**Recall**	**Kappa**
Blob	ArgoUML	2115	7	105	11	61.111	9.483	0.152
Blob	Eclipse	12455	494	1750	437	46.939	19.982	0.212
Blob	Mylyn	2661	10	71	22	68.75	23.656	0.341
Blob	Rhino	437	17	0	0	0	–	0
LPL	ArgoUML	1930	9	280	19	67.857	6.355	0.096
LPL	Eclipse	11842	75	2844	368	83.07	11.457	0.158
LPL	Mylyn	2669	0	95	0	–	0	0
LPL	Rhino	432	14	3	5	26.316	62.5	0.354
Message Chains	ArgoUML	2072	0	166	0	–	0	0
Message Chains	Eclipse	12098	5	3005	27	84.375	0.891	0.013
Message Chains	Mylyn	2583	0	180	1	100	0.552	0.01
Message Chains	Rhino	388	0	66	0	–	0	0

Tabela 5.11: Resultados da detecção de Bad Smells utilizando a ferramenta **inFusion**.

Bad Smell	**Projeto**	**TN**	**FP**	**FN**	**TP**	**Precision**	**Recall**	**Kappa**
Blob	Rhino	437	17	0	0	0	–	0
LPL	Rhino	432	14	3	5	26.316	62.5	0.354
Message Chains	Rhino	388	0	66	0	–	0	0
RPR	Rhino	433	0	11	0	–	0	0

Tabela 5.12: Resultados da detecção de Bad Smells utilizando a ferramenta **iPlasma**.

Bad Smell	**Projeto**	**TN**	**FP**	**FN**	**TP**	**Precision**	**Recall**	**Kappa**
Blob	ArgoUML	2116	6	91	25	80.645	21.552	0.325
Blob	Mylyn	2651	20	61	32	61.538	34.409	0.428
Blob	Rhino	439	15	0	0	0	–	0
RPR	ArgoUML	1659	5	570	4	44.444	0.697	0.006
RPR	Mylyn	2474	3	290	0	0	0	-0.002
RPR	Rhino	433	3	11	0	0	0	-0.011

Tabela 5.13: Resultados da detecção de Bad Smells utilizando a ferramenta **PMD**.

Bad Smell	**Projeto**	**TN**	**FP**	**FN**	**TP**	**Precision**	**Recall**	**Kappa**
Blob	ArgoUML	2094	28	63	53	65.432	45.69	0.518
Blob	Eclipse	12949	0	2187	0	–	0	0
Blob	Mylyn	2611	60	43	50	45.455	53.763	0.473
Blob	Rhino	389	65	0	0	0	–	0
CDSBP	ArgoUML	2181	6	49	2	25	3.922	0.062
CDSBP	Eclipse	12704	154	2141	137	47.079	6.014	0.075
CDSBP	Mylyn	2580	1	182	1	50	0.546	0.009
CDSBP	Rhino	431	6	16	1	14.286	5.882	0.063
Long Method	ArgoUML	1879	13	346	0	0	0	-0.011
Long Method	Eclipse	6911	284	7600	333	53.971	4.198	0.002
Long Method	Mylyn	2410	5	346	3	37.5	0.86	0.011
Long Method	Rhino	405	14	15	20	58.824	57.143	0.545
LPL	ArgoUML	1939	0	299	0	–	0	0
LPL	Eclipse	11902	15	3163	49	76.563	1.526	0.022
LPL	Mylyn	2669	0	93	2	100	2.105	0.04
LPL	Rhino	446	0	6	2	100	25	0.396
SG	ArgoUML	2215	1	22	0	0	0	-0.001
SG	Eclipse	14911	0	228	0	–	0	0
SG	Mylyn	2725	0	39	0	–	0	0
SG	Rhino	452	0	2	0	–	0	0

Capítulo 6

Discussão

Neste capítulo, na seção 6.1 iremos responder as questões de pesquisa apresentadas na Seção 5.1, discutiremos, na Seção 6.2, os benefícios da técnica proposta, e em seguida, na Seção 6.3, serão discutidas as ameaças à validação destes experimentos.

6.1 Respostas às Questões de Pesquisa

6.1.1 Questão 1

Q1: O algoritmo de classificação baseado em Indução de Árvores de Decisão (C5.0) é efetivo em detectar Bad Smells?

Resposta:

Baseado nos resultados do Experimento 1, expostos na Tabela 5.1, especialmente no que diz respeito aos valores de Kappa obtidos, é possível afirmar que o algoritmo C5.0 é capaz de gerar regras efetivas em realizar a detecção de Bad Smells no conjunto de dados estudados. De fato, pode-se ver pela Tabela 6.1 que para todos os 12 Bad Smells analisados, o valor de Kappa esteve acima de 0.4 (concordância moderada) e quase todos (83%) acima de 0.6 (concordância substancial), e ainda uma boa parcela (1/3) obteve Kappa acima de 0.8 que indica concordância quase perfeita com o conjunto de dados anotado.

Tabela 6.1: Distribuição por categorias dos valores de Kappa do Experimento 1.

Valor do Kappa	Interpretação	Porcentagem
$\kappa < 0$	Não concordância	0%
$0 < \kappa < 0.19$	Concordância pobre	0%
$0.20 < \kappa < 0.39$	Concordância razoável	0%
$0.40 < \kappa < 0.59$	Concordância moderada	16.6%
$0.60 < \kappa < 0.79$	Concordância substancial	50.0%
$0.80 < \kappa < 1.00$	Concordância quase perfeita	33.3%

A interpretação dos valores de Kappa na Tabela 6.1 baseia-se no exposto na literatura [22], para mais detalhes, ver Apêndice A.

Ainda da Tabela 5.1, podemos obter que em média, o valor de Kappa é de 0.708. Para dar maior suporte e significado a esta análise, após confirmar com o teste de Shapiro Wilk a normalidade dos valores de Kappa, aplicamos o teste T de Student (considerando $\alpha = 0.05$) com as seguintes hipóteses alternativa e nula:

H_A Em média, o algoritmo C5.0 obtém Kappa superior a 0.6 na comparação com o conjunto de dados anotado.

H_0 Em média, o algoritmo C5.0 obtém Kappa inferior a 0.6 na comparação com o conjunto de dados anotado.

Este teste resultou num p-valor = 0.0218, que por ser menor que α, rejeita a hipótese nula definida acima, H_0. Temos, portanto, subsídios para afirmar que, em média, a concordância é substancial e que, portanto, **o algoritmo de classificação baseado em Indução de Árvores de Decisão (C5.0) é efetivo em detectar Bad Smells.**

6.1.2 Questão 2

Q2: É possível melhorar a eficiência da classificação pré-selecionando as métricas com um Algoritmo Genético?

Resposta:

Conforme resultado do Experimento 2, disposto na Tabela 5.2, para todos os 12 Bad Smells analisados, os valores de Precision e Kappa aumentaram em relação aos resultados do Experimento 1 (Tabela 5.1), e apenas para 2 valores de Recall (nos Bad Smells Speculative Generality e Message Chains) o valor de Recall sofreu redução. Para facilitar a visualização, as Figuras de 5.2 a 5.4 exibem gráficos de barras comparando os valores dessas três medidas para os 12 Bad Smells. Através destes gráficos, é possível notar que há aumento em todos os casos mas que este aumento não aparenta ser muito significativo. Para verificar estatisticamente a significância desta diferença entre os dois experimentos, após verificar a normalidade dos dados com o teste de Shapiro Wilk definimos o seguinte conjunto de hipóteses para aplicação do teste T de Student pareado (também considerando $\alpha = 0.05$):

H_A Em média, a detecção dos Bad Smells com as métricas pré-selecionadas (Experimento 2) obtém maior valor de Precision, Recall e Kappa do que na detecção utilizando todas as métricas (Experimento 1).

H_0 Em média, a detecção dos Bad Smells com as métricas pré-selecionadas (Experimento 2) obtém o mesmo valor de Precision, Recall e Kappa da detecção utilizando todas as métricas (Experimento 1).

Para testar estas hipóteses é necessário a realização de três testes T de Student pareados, um para cada parâmetro (Precision, Recall e Kappa). O resultado da média das diferenças e do p-valor destes testes para cada um dos três parâmetros pode ser visto na Tabela 6.2

Tabela 6.2: Média das diferenças e p-valor obtido a partir do teste T de Student para cada parâmetro.

Parâmetro	**Média das Diferenças**	**p-valor**
Precision	4.241083	0.001679
Recall	2.896083	0.01965
Kappa	0.03608333	0.0001416

Uma vez que os p-valores ficaram todos abaixo do valor de α, podemos rejeitar a hipótese nula e concluir que, sim, **é possível melhorar a eficiência da classificação pré-selecionando as métricas com um Algoritmo Genético.**

Atribuímos esta melhoria na eficiência de detecção ao fato de que, apesar de o algoritmo C5.0 realizar sua própria seleção de métricas, ele não é capaz de fazê-lo com tanta eficácia quanto o Algoritmo Genético que desenvolvemos. Desta forma, ao utilizar todas as 62 métricas presentes na base de dados para realizar a detecção dos Bad Smells, no Experimento 1, uma quantidade excessiva de informação oriunda do conjunto de métricas pode ter deixado o mecanismo de seleção do nó pai de cada subárvore menos eficiente, provavelmente por ter que lidar com valores de entropia muito próximos em alguns casos, dificultando a distinção entre as métricas realmente importantes e as não tão importantes assim.

Por ser um algoritmo evolutivo, após alguns ciclos, o A.G. é capaz de identificar, qual é o subconjunto de métricas que realmente interessa para gerar a regra (Árvore de Decisão) para a detecção de cada Bad Smell.

Há ainda um outro parâmetro decorrente desta análise sobre o qual ainda não discutimos: o tamanho médio da Árvore de Decisão. Era esperado que as regras de detecção geradas no Experimento 2 (Árvores de Decisão) fossem também menores, já que estamos reduzindo o conjunto de métricas. Esta redução do tamanho da árvore também era desejável, para que permitisse detecções mais rápidas (menor custo computacional) em cenários com grande quantidade de classes para analisar. No entanto, aparentemente, o tamanho do conjunto de métricas não está diretamente relacionado ao tamanho da Árvore de Decisão gerada, pois uma mesma métrica pode aparecer várias vezes na árvore.

A Tabela 5.2 e a Figura 5.5 mostram que, na maioria dos casos (9 dos 12 Bad Smells), com a pré-seleção das métricas, o tamanho médio da Árvore de Decisão é, de fato, reduzido. Porém, com o teste T de Student foi possível notar que apesar da média das diferenças ser -0.1916667 (redução), este valor não é significante já que o p-valor resultou em 0.5291 (maior do que α).

6.1.3 Questão 3

Q3: Regras de detecção de Bad Smells aprendidas em um projeto de software preservam sua qualidade quando aplicadas a outros projetos?

Resposta:

Para discutir os resultados do Experimento 3, afim de responder a esta pergunta, inicialmente, a partir das Tabelas 5.4 a 5.7, calculamos as médias dos parâmetros Precision, Recall e Kappa para cada conjunto de treinamento, desconsiderando os valores faltosos (aqueles que não puderam ser calculados devido a divisões por zero) e dispomos estes dados na Tabela 6.3. Note que cada média foi calculada tomando como amostra todos os valores de cada parâmetro para os 3 conjuntos de treinamento. Por exemplo, o valor 56.243, para o conjunto de treinamento ArgoUML, apresentado na primeira linha da Tabela 6.3, é referente à média de todos os valores da coluna Precision na Tabela 5.4. De maneira análoga as outras linhas da Tabela 6.3 foram calculadas.

Tabela 6.3: Média dos parâmetros resultantes do Experimento 3, para cada Conjunto de Treinamento.

Conj. de Treinamento	Parâmetro	Média
ArgoUML	Precision	56.243
ArgoUML	Recall	11.503
ArgoUML	Kappa	0.128
Eclipse	Precision	28.262
Eclipse	Recall	61.005
Eclipse	Kappa	0.243
Mylyn	Precision	52.444
Mylyn	Recall	50.360
Mylyn	Kappa	0.354
Rhino	Precision	53.197
Rhino	Recall	14.868
Rhino	Kappa	0.140

Analisando os dados da Tabela 6.3, podemos perceber que, em relação aos valores de Precision, Recall e Kappa obtidos no Experimento 1, houve uma visível redução nos valores destes parâmetros. Em especial, o valores médios de Kappa mostrados na Tabela 6.3 são bastante inferiores à média dos valores de Kappa (0.708) mostrados na Tabela 5.1.

Para os projetos ArgoUML e Rhino, o valor de Kappa 0.128 pode ser iterpretado, segundo a Tabela A.2, como uma concordância pobre com o conjunto de dados anotado. Enquanto que para os projetos Eclipse e Mylyn, o valor de Kappa pode ser enquadrado como uma concordância razoável. Portanto, temos indícios para afirmar que **regras de detecção de Bad Smells aprendidas em um projeto de software não preservam sua qualidade quando aplicadas a um outro projeto**.

Acreditamos que, pelo fato de o algoritmo de Indução de Árvore de Decisão adaptar as regras geradas aos dados do conjunto de treinamento, elas acabam ganhando uma especificidade muito grande para aquele projeto do conjunto de treinamento e perdem sua eficiência quando aplicadas a um projeto diferente. No Experimento 1, quando a base era composta de todos os projetos, as regras de detecção acabavam se tornando mais genéricas, aplicando-se aos diversos conjuntos de teste com um resultado mais consistente no que diz respeito a qualidade.

6.1.4 Questão 4

Q4: Como o desempenho das regras geradas pelo C5.0 se compara ao desempenho de ferramentas existentes para a detecção de Bad Smells?

Resposta:

Devido à incompletude da analise das ferramentas, conforme discutido na Seção 5.3.4, não temos subsídios para responder de forma contundente a esta pergunta. O que temos são indícios de que as ferramentas analisadas não conseguem obter a mesma precisão, nem consistência, na detecção de Bad Smells apresentada pela técnica proposta. O que segue-se abaixo é uma exposição destes indícios por meio da análise dos resultados do experimento.

As Tabelas 5.10 a 5.13 apresentam o resultado do Experimento 4 onde utilizamos quatro ferramentas para realizar a detecção de Bad Smells no conjunto de dados utilizado neste trabalho. A partir das tabelas é possível ver, além de outros parâmetros, os valores de Precision, Recall e Kappa referentes à detecção de cada Bad Smell analisado pela ferramenta em questão.

A Tabela 5.10 apresenta o resultado da detecção de Bad Smells realizada pela ferramenta inCode. Podemos ver que os valores de Kappa variaram de 0 (concordância pobre) a 0.354 (concordância razoável). Estes valores estão abaixo do desejável para uma detecção que seja suficientemente confiável.

Na Tabela 5.11, que apresenta o resultado da detecção utilizando a ferramenta inFusion somente no projeto Rhino, podemos observar que dos quatro Bad Smells estudados, apenas para um, Long Parameter List a ferramenta obteve um resultado significativo que foi um valor de Kappa de 0.354, ainda assim estando na faixa da concordância razoável.

Para o Bad Smell Blob, que na realidade não ocorre (segundo o conjunto de dados anotado) no projeto Rhino, a ferramenta ainda detectou, erroneamente, 17 positivos, no entanto, como a quantidade de Verdadeiros Positivos foi nula (já que não há este Bad Smell no projeto Rhino), o Kappa foi calculado como zero. O mesmo ocorre para os Bad Smells Message Chains e Refused Parent Bequest, sendo que estes dois Bad Smells estão presentes no projeto Rhino e não foram detectados pela ferramenta.

Para a ferramenta iPlasma, cujos resultados da detecção estão apresentados na Tabela 5.12, tivemos uma pequena melhora no valor de Kappa para o Bad Smell Blob, que no projeto Mylyn, atingiu o valor de 0.428 (concordância moderada) e no projeto ArgoUML atingiu o valor de 0.325 (concordância razoável). No entanto, para o outro Bad Smell analisado por esta ferramenta, Refused Parent Bequest, os resultados foram muito abaixo do desejável, apresentando valores de Kappa que indicam que não houve concordância com o conjunto de dados anotados.

Finalmente, a Tabela 5.13 apresenta os resultados da detecção com a ferramenta PMD. Com esta ferramenta, foi possível analisar os quatro projetos e uma quantidade maior de Bad Smells, tendo sido analisados Blob, Class Data Should Be Private, Long Method, Long Parameter List e Specultive Generality. Nesta tabela podemos notar que dos 20 valores de Kappa calculados, 6 foram nulos (não concordância), 10 foram bem abaixo de 0.19 (concordância pobre) e os outros quatro tiveram uma avaliação melhor, que pode ser considerada como concordância moderada: 0.396 (LPL no projeto Rhino), 0.473 (Blob no projeto Mylyn), 0.518 (Blob no projeto ArgoUML) e 0.545 (Long Method no projeto Rhino).

Na Tabela 6.4, exibimos as médias dos valores do parâmetro Kappa para cada ferramenta, separando por projeto e no total. Já na Tabela 6.5, exibimos as médias do mesmo parâmetro, sendo que para cada Bad Smell, separando por projeto e no total. Para os cálculos das médias destas duas tabelas, excluímos o Bad Smell Blob no projeto Rhino, já que este projeto não apresenta este Bad Smell e invariavelmente o Kappa resulta em zero.

Na Tabela 6.5 é possível observar que das 7 médias de valores de Kappa por Bad Smells (linhas onde lê-se "**Total**"), 6 estão abaixo de 0.19 indicando uma concordância pobre com o conjunto de dados anotados. Para o Bad Smell Blob, a média do Kappa resultou em 0.284, indicando uma concordância razoável.

Se comparamos estes resultados com os do Experimento 1, onde 83% dos Bad Smells analisados obtiveram Kappa acima de 0.6 (concordância substancial) podemos inferir que **as ferramentas analisadas são inferiores, na detecção de Bad Smells, às regras geradas pelo algoritmo indutor de Árvore de Decisão C5.0**, quando aplicadas a estes quatro projetos de software levando em conta as anotações do conjunto de dados utilizado.

Acreditamos que este resultado tão pobre das quatros ferramentas analisadas ocorre principalmente porque elas dependem de regras baseadas na comparação do valores das métricas de software com limiares pré-definidos e estáticos, ou seja, que não se adaptam automaticamente ao projeto de software que está sendo analisado. Usuários destas ferramentas, em geral, precisam adaptar estes limiares de acordo com as particularidades do projeto em questão.

Como o algoritmo C5.0 é capaz de inferir valores para os limiares utilizados nas regras (Árvores de Decisão) geradas por meio de aprendizagem de máquina, estas regras adaptam-se ao conjunto de treinamento e tornam-se mais eficazes quando testadas em um conjunto de testes contendo classes dos mesmo projetos onde foram treinadas.

Tabela 6.4: Média dos valores de Kappa resultantes do Experimento 4, para cada ferramenta, por projeto.

Ferramenta	Projeto	Kappa médio
inCode	ArgoUML	0.083
inCode	Eclipse	0.128
inCode	Mylyn	0.117
inCode	Rhino	0.177
	Total	0.111
inFusion	Rhino	0.118
	Total	0.118
iPlasma	ArgoUML	0.166
iPlasma	Mylyn	0.213
iPlasma	Rhino	-0.01
	Total	0.123
PMD	ArgoUML	0.114
PMD	Eclipse	0.020
PMD	Mylyn	0.107
PMD	Rhino	0.251
	Total	0.123

6.2 Benefícios da Técnica Proposta

O principal benefício do uso da técnica de Indução de Árvore de Decisão para a descoberta automática de regras para detecção de Bad Smells, é a eficiência das regras geradas. Já que a Indução de Árvore de Decisão é um algoritmo de aprendizagem de máquina, é capaz de adaptar-se ao conjunto de treinamento gerando regras específicas e mais precisas para o(s) projeto(s) de software contido(s) no conjunto.

Frequentemente as regras geradas atingem valores relativamente altos de Precision e Recall e ótimos valores de Kappa (acima de 0.6) indicando que a técnica é bastante adequada para este domínio.

Um outro aspecto que vale ressaltar é que, o modelo de detecção resultante, na forma de uma Árvore de Decisão, permite a inspeção e fácil compreensão pelo Desenvolvedor ou Engenheiro de Software. Com esta análise, o profissional pode inclusive melhorar o modelo, propondo pré-seleção das métricas existentes ou até mesmo a inclusão de novas métricas além de outras modificações que podem ser feitas na Árvore de Decisão diretamente. Este tipo de transparência do modelo, não ocorre em outras técnicas que produzem modelos do tipo "caixa preta".

A utilização de um Algoritmo Genético para melhorar a eficiência de classificação também mostrou-se como uma boa complementação ao C5.0 por representar um mecanismo de pré-seleção da melhores métricas para cada Bad Smell, gerando regras ainda mais eficientes e frequentemente menores.

6.3 Ameaças à Validade

Imprecisão na definição dos Bad Smells

Devido aos diferentes nomes dados a um mesmo Bad Smells por diferentes autores, no Experimento 4, mais precisamente na Seção 5.3.4, foi necessário estabelecer algumas equivalências entre os nomes para permitir aferir a eficiência das ferramentas em detectar os Bad Smells anotados no conjunto de dados utilizado no trabalho. Para realizar estre trabalho, estudamos a descrição textual que cada ferramenta dava a cada Bad Smell para verificar se havia equivalência com as descrições de um dos 12 Bad Smells do conjunto de dados. Pode-se ver, então, que pelo fato de ser uma tarefa de interpretação de texto, muitas vezes contendo ambiguidades ou imprecisões (palavras como "pouco", "muito", etc.) estas equivalências podem não estar totalmente corretas, e neste caso, estaríamos comparando a detecção de Bad Smells diferentes, o que poderia explicar, em parte, a má performance das ferramentas estudadas.

Incompletude na análise das ferramentas

A comparação das regras geradas pela técnica com ferramentas existentes para detectar Bad Smells pode ser considerada incompleta por termos abrangido apenas 4 ferramentas e além disso, conforme discutido na Seção 5.3.4, a análise destas ferramentas também teve suas limitações técnicas no que diz respeito a: O número de Bad Smells em comum com os 12 analisados neste trabalho, que se somados os analisados por todas as ferramentas resulta em pouco mais da metade (apenas 7); A aplicação das ferramentas aos projetos estudados, onde apenas duas ferramentas foram capazes de analisar os quatro projetos, e uma delas só conseguiu analisar um deles.

Devido a estas limitações, consideramos os resultados do Experimento 4 apenas como um resultado isolado, que por esta análise, não pode ser generalizado.

Overfitting

Overfitting diz respeito ao sobreajustamento de um modelo aos dados treinamento ou teste, tornando-o menos eficaz para outros conjuntos além dos estudados. No nosso caso, é possível que as regras geradas sofram de overfitting no conjunto de dados de treinamento, no entanto para minimizar este efeito, nos Experimentos 1 e 2 nós utilizamos a validação cruzada com 10-folds, de forma que os conjuntos de treinamento e teste sofrem um revesamento constante.

O Experimento 3, onde utilizamos conjuntos de treinamento e de teste oriundos de projetos diferentes, dá indícios de que o problema de overfitting pode ter causado a superespecificação das regras geradas, tornando-as ineficazes em outros projetos.

De toda maneira, a técnica que propomos é para a descoberta automática das regras de detecção específicas para o projeto de software em questão. Não é o objetivo deste trabalho gerar regras reutilizáveis, e sim propor uma técnica para descoberta automática de regras específicas e mais precisas.

Validação do conjunto de dados

O conjunto de dados utilizado, oriundo do trabalho de Khomh et al. [19], que contém as anotações sobre a presença dos 12 Bad Smells em cada uma das classes dos 4 projetos influencia diretamente na qualidade final das regras geradas, de modo que as regras geradas só podem ser, no máximo, tão realistas quanto forem as anotações.

Tabela 6.5: Média dos valores de Kappa resultantes do Experimento 4, para cada Bad Smell, por projeto.

BadSmell	**Projeto**	**Kappa médio**
Blob	ArgoUML	0.331
Blob	Eclipse	0.106
Blob	Mylyn	0.414
Blob	Rhino	–
	Total	0.284
CDSBP	ArgoUML	0.062
CDSBP	Eclipse	0.075
CDSBP	Mylyn	0.009
CDSBP	Rhino	0.063
	Total	0.052
Long Method	ArgoUML	-0.011
Long Method	Eclipse	0.002
Long Method	Mylyn	0.011
Long Method	Rhino	0.545
	Total	0.137
LPL	ArgoUML	0.048
LPL	Eclipse	0.090
LPL	Mylyn	0.020
LPL	Rhino	0.368
	Total	0.132
Message Chains	ArgoUML	0.000
Message Chains	Eclipse	0.013
Message Chains	Mylyn	0.552
Message Chains	Rhino	0.000
	Total	0.141
RPB	ArgoUML	0.006
RPB	Eclipse	–
RPB	Mylyn	-0.002
RPB	Rhino	-0.005
	Total	-0.002
SG	ArgoUML	-0.001
SG	Eclipse	0.000
SG	Mylyn	0.000
SG	Rhino	0.000
	Total	0.000

Capítulo 7

Trabalhos Relacionados

Neste capítulo, realizamos uma breve revisão da literatura relacionada, dando ênfase aos trabalhos cujo foco esteja principalmente na detecção de Bad Smells. Organizamos o capítulo em três seções: Técnicas Tradicionais, Técnicas Formais e Técnicas inteligentes.

7.1 Técnicas tradicionais

Tradicionalmente, a detecção de Bad Smells tem sido feita através de regras baseadas em métricas de software. Em Marinescu [27], este tipo de abordagem é explorada para detectar os Bad Smells no código de um sistema industrial de tamanho médio, escrito em C++ segundo o paradigma de orientação a objetos, cuja arquitetura apresentava sinais de degradação levando os desenvolvedores a sentir a necessidade de refatorar o código, justificado a necessidade de encontrar as falhas de design. Para tanto, define-se uma abordagem de detecção baseada em métricas que consiste numa sequência de passos:

Análise Quantitativa do Bad Smell Nesta etapa procura-se descrever o Bad Smell de maneira quantitativa, reduzindo a imprecisão da descrição.

Seleção das Métricas Com base na descrição quantitativa, seleciona-se as métricas mais apropriadas para medir as características das estruturas com falhas de design. Em seguida, uma combinação específica das métricas selecionadas (regra) descreve a estratégia de detecção.

Detecção dos Suspeitos O terceiro passo é medir o sistema com as métricas selecionadas e avaliar, de acordo com a regra definida, os fragmentos suspeitos de estarem infectados pelo Bad Smell em questão.

Exame dos Suspeitos Um exame manual é realizado, com base em código fonte e outras informações de contexto, para verificar se os fragmentos suspeitos estão, de fato, com

falha ou se houve uma decisão consciente, de projeto, que fez com que o design do fragmento apresentasse essa combinação de métricas.

Em [28], os mesmo autor investe mais na abordagem proposta em [27], refinando-a, acrescentando uma etapa de filtragem de dados, para reduzir a dimensionalidade do conjunto de dados, e expandindo-a para a detecção de uma diversidade maior de Bad Smells, além de outras melhorias. Uma análise da técnica em um sistema de médio porte é realizada, obtendo resultados de detecção com valores de acurácia média acima de 67%.

Schumacher et al. [39] utiliza regras definidas por Marinescu em [28] em uma ferramenta automática de detecção e compara os resultados com aquele obtidos por inspeção manual realizada pelos desenvolvedores do sistema analisado. Os resultados mostram que, mesmo quando os indivíduos descrevem a atividade de detecção do Bad Smell "God Class"como uma tarefa fácil, a concordância com a classificação automática é baixa. Os autores concluem que uma pré-seleção automática dos fragmentos, baseada nos valores da métricas, pode reduzir o esforço de classificação manual e que a detecção automática acompanhada de revisão manual aumenta a confiança nos resultados de classificadores baseados em métricas.

Em Moha et al. [29], os autores também utilizam uma estratégia dividida em etapas, mas concentram-se na melhoria da etapa que concerne à definição das regras. Uma técnica de detecção DETEX (*DETection EXpert*) é desenvolvida com a finalidade de permitir que engenheiros de software especifiquem os Bad Smells em um alto nível de abstração, utilizando um vocabulário unificado e uma linguagem de domínio específico[1] que permite a geração dos algoritmos de detecção.

A DSL definida permite a epsecificação dos Bad Smells de forma declarativa, como composição de regras em estruturas chamadas de *rule cards*. Na Figura 7.1 reproduzimos um dos exemplos de *rule card* apresentados em [29] para a detecção do Bad Smell "Spaghetti Code". Segundo os autores, Spaghetti Code é revelado por "classes sem estrutura, que declaram métodos longos, sem parâmetros, e utilizam variáveis globais. Os nomes das classes e métodos podem sugerir programação procedural". Além disso, um código com Spaghetti Code tende a "evitar o uso dos mecanismos de orientação a objeto, polimorfismo e herança.

Com base nesta descrição textual do Spaghetti Code, podemos compreender o código presente na Figura 7.1: A regra é definida usando a intersecção de de seis regras (linha 2). Uma classe será considerada SC se declarar métodos com número alto de linhas de código (linha 3), sem parâmetros (linha 4), se não fizer uso de herança (linha 5) e polimorfismo (linha 6), se tiver um nome que remeta a programação procedural (linha 7) e se declarar variáveis globais (linha 8).

Com regras parecidas com o *Rule Card* apresentado na Figura 7.1, outros Bad Smells também têm suas regras de detecção definidas em [29]. Os autores aplicaram estas regras de

[1] DSL - Domain Specific Languge

```
1  RULE_CARD:SpaghettiCode {
2    RULE:SpaghettiCode
           { INTER LongMethod NoParamete NoInheritance
           NoPolymorphism ProceduralName UseGlobalVariable };
3    RULE:LongMethod       { METRIC LOC_METHOD VERY_HIGH 10.0 };
4    RULE:NoParameter      { METRIC NMNOPARAM VERY_HIGH 5.0 };
5    RULE:NoInheritance    { METRIC DIT 1 0.0 };
6    RULE:NoPolymorphism   { STRUCT NO_POLYMORPHISM };
7    RULE:ProceduralName   { LEXIC CLASS_NAME
                                 (Make, Create, Exec...) };
8    RULE:UseGlobalVariable { STRUCT USE_GLOBAL_VARIABLE };
9  };
```

Figura 7.1: *Rule card* do algoritmo de detecção do Bad Smell "Spaghetti Code"(SC) expresso na DSL definida em [29].

detecção em sistemas de código aberto e obtiveram resultados de recall de 100% e precision maior que 50% em um dos sistemas estudados.

O trabalho de Fontana et al. [4] analisa e compara uma série de ferramentas para detecção de Bad Smells que utilizam este tipo de abordagem baseada em métricas. Eles aplicaram as ferramentas para a detecção de cinco Bad Smells diferentes (Duplicated Code, Feature Envy, God Class, Large Class, Long Method, Long Parameter List) em um sistema de código aberto escrito em Java, GanttProject. Os resultados mostram que, em geral, o nível de concordância entre as ferramentas é muito baixa apesar de todas elas basearem-se no mesmo tipo de técnica de detecção. Isso parece ocorrer, principalmente, pela maneira imprecisa com que os Bad Smells são descritos, o que dá margem a diversas interpretações e consequentemente diferentes implementações.

Com o exposto, constata-se que apesar do uso de métricas de software ter se mostrado promissor para a detecção de vários Bad Smells[28, 29], especialmente aquele cuja definição está mais relacionadas com os aspectos quantitativos, que podem ser capturados pelas métricas, o principal problema ainda está na definição das regras, que além de ser uma tarefa manual e trabalhosa, é extremamente dependente da forma como os Bad Smells são definidos e interpretados pelos engenheiros de software.

7.2 Técnicas formais

As técnicas de Lógica Proposicional, Meta Programação Lógica e OCL estão entre as técnicas formais utilizadas para a detecção de Bad Smells. Abaixo descrevemos alguns trabalhos que as utilizam.

Em Tourwé and Mens [45], utiliza-se a técnica de Meta Programação Lógica (Logic Meta Programming - LMP) para definir regras de detecção de Bad Smells. As regras para detecção precisam ser definidas manualmente e especificamente, numa linguagem batizada de SOUL, para o Bad Smell que se deseja detectar. Estas regras investigam aspectos da progra-

mação orientada a objetos a fim de identificar a presença do Bad Smell.

Os autores mostram dois exemplos onde realizam a definição das regras para a detecção de dois Bad Smells: Parâmetros Obsoletos e Interfaces Inapropriadas. Os autores definem regras na linguagem SOUL, que é uma variante de Prolog, por eles desenvolvida. Na Figura 7.2 reproduzimos o código referente ao algoritmo de detecção de Parâmetros Obsoletos apresentado em [45].

```
obsoleteParameter(?class, ?selector,?parameter) :-
[1] classImplements(?class,?selector),
[2] parameterOf(?class,?selector,?parameter),
[3] forall(subclassImplements(?class,?selector,?subclass),
[4]        not(selectorUsesParameter(?subclass,?selector,
                                     ?parameter)))
```

Figura 7.2: Algoritmo para detecção do Bad Smell Parâmetro Obsoleto expresso em LMP, segundo Tourwé and Mens [45].

Entre os aspectos analisados, a linguagem suporta verificar se uma determinada classe é uma implementação de uma outra; se um determinado parâmetro pertence a uma determinada classe; ou se um parâmetro é utilizado em uma certa subclasse.

Estas informações são bastante úteis para a detecção de Bad Smells que estejam relacionados com estes aspectos, mas podem não ser tao úteis para a detecção de Bad Smells com definições menos precisas, ou menos formais, tais como Blob, Swiss Army Knife e outros que dependem de conceitos relacionados a quantidades relativas. É possível perceber, no entanto, que os dois Bad Smells utilizados tem definições relativamente precisas, o que permite a definição de regras deste tipo com certa facilidade.

No trabalho de Piveta et al. [36], os autores combinam um conjunto de abordagens de refatoramento e fornecem um conjunto de mecanismos para representar "oportunidades de refatoramento"(incluindo os Bad Smells). O formalismo escolhido para expressar as condições de refatoramento é a Lógica Proposicional, implementada utilizando construções nativas da linguagem ou meta-dados, quando disponíveis, para expressar predicados.

Em Kim et al. [20], para elaborar as regras de detecção de Bad Smells, faz-se uso de OCL, *Object Constraint Language*, que é uma linguagem formal usada para descrever expressões em modelos UML. As regras geradas em OCL, portanto, são capazes de detectar Bad Smells por meio da análise da estrutura do modelo UML ao invés da utilização de métricas. Na Figura 7.3 reproduzimos o código apresentado pelos autores para a detecção do Bad Smell "Refused Bequest".

Analisando os três trabalhos apresentados acima, podemos constatar que as abordagens por técnicas formais podem atingir boa precisão de especificação das regras, no entanto também dependem da definição precisa do Bad Smell que está sendo estudado em termos que possam ser expressos, manualmente, na abordagem escolhida. A definição dos Algorit-

```
context Project
def : getRefusedBequestDetection():
Sequence(Tuple(superClass : String, field : String, subClass : String)) =
self.getSuperClasses()->collect(sType | sType.getFieldDeclarationKey()->collect( sFieldKey |
if self.getUsedClassOfSimpleName(sFieldKey)->includes(sType.getTypeKey) then null
else if self.getTypeOfQualifiedName(sFieldKey)->includes(sType.getTypeKey) then null
else if self.getUsedClassOfSimpleName(sFieldKey)->includesAll(self.getSubclassTypeKey(sType)) then null
else
  (self.getUsedClassOfSimpleName(sFieldKey)->intersection(self.getSubclassTypeKey(sType))
  ->union(self.getTypeOfQualifiedName(sFieldKey))-self.getSubclassTypes(sType)
  ->iterate(acc:TypeDeclaration ;
  result : Set(String) = Set{} | if acc.getFieldNames()->includes(self.getVariableName(sFieldKey)) then
    acc.getTypeKey->union(result) else result endif
  ))->collect(target | Tuple{superClass = sType.getClassName(), field = self.getVariableName(sFieldKey),
  subClass = self.getClassNameByKey(target)}) endif endif endif
))->excluding(null)->asSet()->asSequence()
```

Figura 7.3: Algoritmo para detecção do Bad Smell "Refused Bequest"em OCL, como definido em Kim et al. [20].

mos de Detecção, seja em LMP, Lógica Proposicional ou OCL, também são tarefas manuais complexas.

7.3 Técnicas inteligentes

Para reduzir o esforço humano na tarefa de detecção de Bad Smells, algumas abordagens têm sido propostas para a geração automática de modelos de detecção. Tais técnicas eliminam a necessidade da especificação manual de regras baseada na interpretação e no conhecimento do engenheiro ou desenvolvedor de software sobre determinado Bad Smell que se deseja detectar.

Para tanto, essas abordagens baseiam-se em técnicas de Inteligência Artificial, mais precisamente, Aprendizagem de Máquina, onde um conjunto de fragmentos de códigos anotados (rotulados com "verdadeiro"ou "falso"para determinado Bad Smell) é utilizado como conjunto de treinamento, com o qual o algoritmo aprenderá algum tipo de modelo de classificação, que pode ser compreendido como a regra de detecção do Bad Smell.

O trabalho de Kreimer [21], utiliza um algoritmo de classificação Indutor de Árvore de Decisão (C4.5) para gerar as regras de detecção, no entanto, o trabalho peca no que diz respeito à validação, pois o conjunto de treinamento foi composto de apenas 20 exemplos.

Em Kessentini et al. [17], os autores utilizam Programação Genética para a extração das regras de forma evolutiva. Programação Genética é uma técnica de busca inspirada na seleção natural. Programas são gerados e modificados de forma evolutiva até que se encontre uma solução satisfatória. Neste caso, os programas são as regras de detecção. Como entrada ao algoritmo, são dados uma base de exemplos composta de fragmentos de códigos manualmente inspecionados e anotados (conjunto de treinamento) e um conjunto de métricas.

Com estas informações, a Programação Genética então trabalha para gerar regras, que são combinações de métricas de software com limiares, que satisfaçam os exemplos no conjunto de treinamento, maximizando o número de defeitos encontrados em comparação com a real quantidade da base de exemplos. Com a técnica proposta, quatro sistemas de código aberto são analisados e os resultados (recall e precision acima de 80%) da detecção dos Bad Smells indicam que a técnica é bastante promissora. Abordagens muito similares, também com Programação Genética são utilizadas em outros trabalhos [25, 15, 34].

Já em Kessentini et al. [16], ao invés de utilizar um conjunto de treinamento contendo elementos de código do próprio projeto a ser analisado, os autores utilizaram o código de um framework, JHotDraw, utilizado para construção de editores gráficos como uma base de código "bem feito", de forma que, para detectar os Bad Smells. Os autores justificam o uso deste código como exemplo alegando que o código JHotDraw contém muito pouco defeito e que trabalhos anteriores não detectaram nenhum Bad Smell do tipo "Blob"neste código e este é um dos três Bad Smells que foi analisado por eles.

A técnica proposta mede o quão distante está o atual fragmento de código daquilo que está posto na base (código do JHotDraw) como código adequado, quanto maior esta distância (menor semelhança) mais "arriscado"é considerado o fragmento de código. A técnica é baseada numa metáfora de sistemas biológicos imunes, chamada de Sistemas Imunes Artificiais. Com esta abordagem os autores conseguiram constatar que 90% dos fragmentos de código mais arriscados, eram realmente defeitos.

As técnicas de geração automática das regras têm a grande vantagem de dispensarem o esforço humano de análise da descrição do Bad Smell e especificação das regras, já que encontram automaticamente regras que ajustam-se bem aos conjuntos de treinamento. Por outro lado, necessitam de conjuntos de exemplos manualmente analisados e anotados. As regras geradas serão, no máximo, apenas tão boas em detectar os Bad Smells quanto foram os engenheiros de software que anotaram o conjunto de exemplos.

Capítulo 8

Conclusão

Este trabalho propôs um método para a descoberta automática de regras, baseadas em métricas de software, para a detecção de Bad Smells por meio de indução de árvores de decisão. Além disso, visando otimizar o desempenho das regras geradas pelo método proposto, implementamos um Algoritmo Genético para pré-selecionar as métricas de software mais informativas para cada Bad Smell.

Para demonstrar a efetividade do método proposto, utilizamos um conjunto de dados com informações sobre métricas de software calculadas para 4 projetos de código aberto, além do conjunto de dados anotado, e realizamos a geração automática das regras as quais foram avaliadas quanto à sua qualidade em realizar a detecção dos Bad Smells.

Com os experimentos realizados, vimos que a técnica de Indução de Árvore de Decisão é capaz de gerar regras efetivas em realizar a detecção de Bad Smells de forma bastante satisfatória (Kappa quase sempre acima de 0.6) e ainda mostramos que é possível diminuir o erro de classificação além de, muitas vezes, reduzir o tamanho das regras geradas, utilizando um Algoritmo Genético para a pré-seleção das métricas mais informativas para cada Bad Smell. Em comparação com ferramentas existentes para detecção de Bad Smells, mostramos indícios de que as regras geradas pela técnica proposta apresentam bom desempenho.

Contudo, o método proposto apresenta algumas limitações: A especificidade das regras geradas e a necessidade de um conjunto de dados devidamente anotados. No tocante à primeira limitação, o que ocorre é que, por utilizarmos conjuntos de treinamento contendo informações sobre projetos específicos, o algorítimo de classificação tende a gerar regras que funcionam muito bem, sendo que especificamente no caso do projeto apresentado, um dos fatores que contribuem para isso é *overfitting*, conforme discutido na Seção 6.3. Em [21], tenta-se reduzir este problema por meio da utilização de retroalimentação no algoritmo de aprendizagem, no entanto, a eficácia não é comprovada.

De qualquer maneira, o objetivo da proposta não era o de gerar regras genéricas que pudessem ser reutilizadas em outros projetos de software, mas isso seria desejável. Quanto à necessidade de conjuntos de dados anotados, vimos na Seção 7.3, que não somente o mé-

todo que propomos, mas todos aqueles que utilizam técnicas de aprendizagem de máquina dependem de um conjunto de treinamento suficientemente extenso e bem definido, com anotações válidas, ou no mínimo confiáveis, a respeito da existência dos Bad Smells, para gerar regras de boa qualidade.

Em trabalhos futuros pretendemos utilizar conjuntos de dados maiores, com maior diversidade de projetos de softwares e, idealmente, com anotações realizadas pelos próprios desenvolvedores. Acreditamos que com uma base suficientemente grande poderemos gerar regras com melhor qualidade e confiabilidade. Além disso, pretendemos desenvolver, utilizando o método proposto, uma ferramenta para a detectar Bad Smells durante o desenvolvimento do projeto de software. Tal ferramenta poderá ser implementada por meio da API de *plugins* do Eclipse. Com isso, poderemos experimentar a inclusão de um mecanismo de retroalimentação no algoritmo, onde o desenvolvedor possa dar *feedback* sobre a detecção, melhorando a aprendizagem do modelo.

Referências

[1] L. Breiman. *Classification and regression trees*. The Wadsworth and Brooks-Cole statistics-probability series. Chapman & Hall, 1984. ISBN 9780412048418.

[2] Leonard A. Breslow and David W. Aha. Simplifying decision trees: A survey. *Knowl. Eng. Rev.*, 12(1):1–40, January 1997. ISSN 0269-8889. doi: 10.1017/S0269888997000015.

[3] W. J. Brown, R. C. Malveau, W. H. Brown, McCormick III, and T. J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1st edition, March 1998.

[4] F A Fontana, P Braione, and M Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2), 2012.

[5] Eclipse Foundation. The eclipse foundation open source community website. Disponível em http://www.eclipse.org, 2013. Acessado em Outubro de 2013.

[6] Eclipse Foundation. Eclipse mylyn open source project. Disponível em http://www.eclipse.org/mylyn, 2013. Acessado em Outubro de 2013.

[7] Mozilla Foundation. Dowload firefox - free web browser - mozilla. Disponível em http://www.mozilla.org/firefox, 2014. Acessado em Março de 2014.

[8] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[9] Yann-Gael Guehneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. *2013 20th Working Conference on Reverse Engineering (WCRE)*, 0:172–181, 2004. ISSN 1095-1350. doi: http://doi.ieeecomputersociety.org/10.1109/WCRE.2004.21.

[10] Jiawei Han and Micheline Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 1-55860-489-8.

[11] E. B Hunt, J. Marin, and P. J. Stone. *Experiments in Induction*. Academic Press, 1966.

[12] Intooitus. incode | intooitus - source code analysis and quality assesment tools. Disponível em http://www.intooitus.com/products/incode, 2013. Acessado em Outubro de 2013.

[13] Intooitus. infusion | intooitus - source code analysis and quality assesment tools. Disponível em http://www.intooitus.com/products/infusion, 2013. Acessado em Outubro de 2013.

[14] Marian Jureczko and Diomidis Spinellis. *Using Object-Oriented Design Metrics to Predict Software Defects*, volume Models and Methodology of System Dependability of *Monographs of System Dependability*, pages 69–81. Oficyna Wydawnicza Politechniki Wroclawskiej, Wroclaw, Poland, 2010. ISBN 978-83-7493-526-5.

[15] M Kessentini, Wael Kessentini, and Abdelkarim Erradi. Example-based Design Defects Detection and Correction. *Journal of Automated Software Engineering*, pages 1–32, 2011.

[16] Marouane Kessentini, Stéphane Vaucher, and Houari Sahraoui. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*, page 113, New York, New York, USA, 2010. ACM Press. ISBN 9781450301169. doi: 10.1145/1858996.1859015.

[17] Marouane Kessentini, Wael Kessentini, Houari Sahraoui, Mounir Boukadoum, and Ali Ouni. Design Defects Detection and Correction by Example. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 81–90. Ieee, June 2011. ISBN 978-1-61284-308-7. doi: 10.1109/ICPC.2011.22.

[18] F. Khomh, M. Di Penta, and Y. Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pages 75–84, Oct 2009. doi: 10.1109/WCRE.2009.28.

[19] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Softw. Engg.*, 17(3):243–275, June 2012. ISSN 1382-3256. doi: 10.1007/s10664-011-9171-y.

[20] TW Kim, TG Kim, and JH Seu. Specification and Detection of Code Smells using OCL. *Software Technology*, pages 55–60, 2012.

[21] Jochen Kreimer. Adaptive Detection of Design Flaws. *Electronic Notes in Theoretical Computer Science*, 141(4):117–136, December 2005. ISSN 15710661. doi: 10.1016/j.entcs.2005.02.059.

[22] J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):pp. 159–174, 1977. ISSN 0006341X.

[23] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-39538-6. doi: 10.1007/3-540-39538-5.

[24] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. Von Staa. On the relevance of code anomalies for identifying architecture degradation symptoms. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 277–286, March 2012. doi: 10.1109/CSMR.2012.35.

[25] U. Mansoor, M. Kessentini, S. Bechikh, and K. Deb. Code-Smells Detection using Good and Bad Software Design Examples. Technical report, University of Michigan, Michigan, USA, 01 2014.

[26] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *In ICSM (Industrial and Tool Volume*, pages 77–80. Society Press, 2005.

[27] Radu Marinescu. Detecting Design Flaws via Metrics in Object-Oriented Systems A Metrics-Based Approach for Problem Detection. *International Conference and Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 173–182, 2001.

[28] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2213-0.

[29] N. Moha, Y.-G. Gueheneuc, L. Duchien, and a. F. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, January 2010. ISSN 0098-5589. doi: 10.1109/TSE.2009.50.

[30] M.J. Munro. Product metrics for automatic identification of "bad smell"design problems in java source-code. In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 15–15, Sept 2005. doi: 10.1109/METRICS.2005.38.

[31] Mika Mäntylä. Bad smells in software – a taxonomy and an empirical study. MSc, Helsinki University of Technology, Software Business and Engineering Institute, 2003.

[32] Mozilla Developer Network and individual contributors. Rhino | mdn. Disponível em http://www.mozilla.org/rhino, 2013. Acessado em Outubro de 2013.

[33] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *Proceedings of the*

2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM '09, pages 390–400, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-4842-5. doi: 10.1109/ESEM.2009.5314231.

[34] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, 20(1):47–79, January 2012. ISSN 0928-8910. doi: 10.1007/s10515-011-0098-8.

[35] Thair Nu Phyu. Survey of classification techniques in data mining. *Proceedings of the International MultiConference of Engineers and Computer Scientists*, 1, 2009.

[36] Eduardo Piveta, Marcelo Pimenta, João Araújo, Ana Moreira, Pedro Guerreiro, and R. Tom Price. Representing refactoring opportunities. In *Proceedings of the 2009 ACM symposium on Applied Computing - SAC '09*, page 1867, New York, New York, USA, 2009. ACM Press. ISBN 9781605581668. doi: 10.1145/1529282.1529701.

[37] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1-55860-238-0.

[38] RuleQuest. Data mininig tools see5 and c5.0. Disponível em http://rulequest.com/see5-info.html, 2013. Acessado em Outubro de 2013.

[39] Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw. Building empirical support for automated code smell detection. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '10*, page 1, 2010. doi: 10.1145/1852786.1852797.

[40] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321313798.

[41] Sourceforge. Pmd. Disponível em http://pmd.sourceforge.net/, 2013. Acessado em Outubro de 2013.

[42] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005. ISBN 0321321367.

[43] Ptidej Team. An exploratory study of the impact of antipatterns on class change- and fault-proneness. Disponível em http://www.ptidej.net/downloads/replications/emse10/, 2010. Acessado em Outubro de 2013.

[44] Tigris Open Source Software Engineering Tools. argouml.tigris.org. Disponível em http://argouml.tigris.org, 2013. Acessado em Outubro de 2013.

[45] T Tourwé and T Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *European Conference on Software Maintenance and Reengineering*, 2003.

Apêndice A

Avaliação do Modelo Classificador

O comportamento do modelo classificador quando aplicado ao conjunto de teste pode ser representado por meio de uma matriz de confusão (ver Tabela A.1), que apresenta, basicamente, na diagonal principal, a quantidade de vezes em que as amostras foram classificadas corretamente, e no restante da matriz, a quantidade de vezes em que as amostras foram classificadas incorretamente.

		Classe Predita	
		Classe = 1	Classe = 0
Classe Real	Classe = 1	TP	FN
	Classe = 0	FP	TN

Tabela A.1: Matriz de confusão para um problema com duas classes. Adaptado de [42].

Na matriz acima, TP e TN representam respecitvamente as quantidades de "verdadeiros postitivos"(*true positive*) e de "verdadeiros negativos"(*true negative*), ou seja, as classificações que resultaram em concordância entre o modelo e o que está anotado no conjunto de testes. Já FP e FN representam respecitvamente as quantidades de "falsos postitivos"(*false positive*) e de "falsos negativos"(*false negative*), ou seja, as classificações que resultaram em disconcordância.

A partir da matriz de confusão é possível calcular três métricas de performance que nos interessam sobre a qualidade do modelo do modelo classificador: **Taxa de erro**, **Precision** e **Recall** da seguinte forma:

$$\text{Taxa de erro} = \frac{\text{Número de predições incorretas}}{\text{Número total de predições}} = \frac{FN+FP}{TP+TN+FP+FN}$$

$$\text{Precision} = \frac{\text{Número de predições corretas de positivos}}{\text{Número total de predições de positivos}} = \frac{100 \times TP}{FP+TP}$$

$$\text{Recall} = \frac{\text{Número de predições corretas de positivos}}{\text{Número total de positivos}} = \frac{100 \times TP}{FN+TP}$$

O fator multiplicador, 100, presente no numerador de Precision e Recall tem o único objetivo de alterar a escala para valores de 0 a 100, de forma que possam ser entendidos como uma porcentagem.

A simplicidade dessas medidas as tornam deficientes quando se quer saber a real utilidade do classificador se comparado a um simples sorteio, que é portanto, regido apenas pelo acaso. Existe, no entanto, uma medida mais sensível, chamada de índice Kappa, que mede o nível de concordância entre diferentes observadores.

O índice Kappa foi desenvolvido para ser utilizado em problemas onde há mais de um juiz indicando o resultado e precisa-se, portanto, saber o nível de concordância entre os juízes. No caso da avaliação de um modelo classificador, consideramos que temos dois "juízes": A classificação predita pelo modelo e a classificação real da instância no conjunto de teste. Dessa forma podemos aferir o nível de concordância entre o modelo e o conjunto de teste. A partir da matriz de confusão mostrada na Tabela A.1, o índice Kappa pode ser calculado da seguinte maneira:

$$\text{Kappa} = \frac{P_r(a) \text{ - } P_r(e)}{1 \text{ - } P_r(e)}$$

Onde $P_r(a)$ é a probabilidade de concordância, calculada pela razão entre o número de predições corretas e o número total de predições, e $P_r(e)$ é a probabilidade de concordância ao acaso e podem ser calculados como segue:

$$P_r(a) = \frac{TP + TN}{TP + TN + FP + FN}$$

Para calcular $P_r(e)$, faz-se necessário o cálculo da probabilidade real das classes i e j, $P_{\text{real}}\,i$ e $P_{\text{real}}\,j$, e a probabilidade das classes i e j serem preditas pelo modelo $P_{\text{pred}}\,i$ e $P_{\text{pred}}\,j$ que são:

$$P_{\text{real}}\,i = \frac{TP + FN}{TP + TN + FP + FN}$$

$$P_{\text{real}}\,j = \frac{FP + TN}{TP + TN + FP + FN}$$

$$P_{\text{pred}}\,i = \frac{TP + FP}{TP + TN + FP + FN}$$

$$P_{\text{pred}}\,j = \frac{FN + TN}{TP + TN + FP + FN}$$

Com isso, podemos calcular $P_r(e) = P_{\text{real}}\,i \cdot P_{\text{pred}}\,i + P_{\text{real}}\,j \cdot P_{\text{pred}}\,j$.

Como pode-se ver, o índice Kappa tem valor máximo 1, que representa total concordância entre o modelo e a realidade representada pelo conjunto de teste. Um valor nulo indica nenhuma concordância, ou que a concordância é apenas tão boa quanto a do acaso, um va-

lor menor que zero significa, a princípio, que não há concordância, mas o módulo do valor não tem uma semântica definida e não pode ser interpretado como o nível de discordância. Na Tabela A.2 podemos ver uma interpretação dos intervalos de valores do índice Kappa.

Tabela A.2: Interpretação dos valores de Kappa de acordo com Landis and Koch [22].

Valor do Kappa	Interpretação
$\kappa < 0$	Não concordância
$0 < \kappa < 0.19$	Concordância pobre
$0.20 < \kappa < 0.39$	Concordância razoável
$0.40 < \kappa < 0.59$	Concordância moderada
$0.60 < \kappa < 0.79$	Concordância substancial
$0.80 < \kappa < 1.00$	Concordância quase perfeita