IDENTIFICAÇÃO DE FUNÇÕES UTILITÁRIAS EM JAVA E JAVASCRIPT

TAMARA MÁRCIA MENDES

IDENTIFICAÇÃO DE FUNÇÕES UTILITÁRIAS EM JAVA E JAVASCRIPT

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente
Coorientador: André Cavalcante Hora

Belo Horizonte
Agosto de 2016

© 2016, Tamara Márcia Mendes.
Todos os direitos reservados.

Mendes, Tamara Márcia

M538i Identificação de Funções Utilitárias em Java e JavaScript / Tamara Márcia Mendes. — Belo Horizonte, 2016
xxi, 86 f. : il. ;86 cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Marco Túlio de Oliveira Valente
Coorientador: André Cavalcante Hora

1. Computação - Teses. 2. Arquitetura de software. 3. Programação modular - Teses. I. Orientador. II. Coorientador. III. Título.

CDU 519.6*32 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Identificação de funções utilitárias em Java e Javascript

TAMARA MÁRCIA MENDES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Orientador
Departamento de Ciência da Computação - UFMG

PROF. ANDRÉ CAVALCANTE HORA - Coorientador
Faculdade de Computação - UFMS

PROF. EDUARDO MAGNO LAGES FIGUEIREDO
Departamento de Ciência da Computação - UFMG

PROF. MARCELO DE ALMEIDA MAIA
Departamento de Ciência da Computação - UFU

Belo Horizonte, 11 de agosto de 2016.

Agradecimentos

A gratidão é uma forma de retribuir um pouco do muito que nos foi dado. Por isso, agradeço a todas as pessoas que de alguma forma contribuíram para que eu concluísse mais este desafio. Agradeço a ajuda, o apoio, a atenção, o incentivo e a compreensão em minhas ausências.

Expresso gratidão a minha família que é meu alicerce e meu recanto seguro, em especial a meus pais Zélia e Humberto pelo apoio incondicional.

Agradeço a meu namorado, amigo, e companheiro Rafael, que me apoia, me incentiva e me ajudou muito nessa conquista.

Agradeço ao Marco a orientação, as ideias, a compreensão. Foi um prazer aprender e trabalhar com esse excelente professor, que sempre me propunha novos desafios e me tirava da inércia.

Agradeço a meu coorientador André pelo apoio, tempo e conhecimento que contribuíram para a realização deste trabalho.

Obrigada aos amigos, colegas do mestrado e colegas da Concert. Sem vocês essa jornada seria mais difícil e teria menos cor.

Agradeço também a todos profissionais do PPGCC que contribuem todos os dias com seu trabalho e dedicação para formar melhores estudantes e profissionais.

"Eu nunca estou realmente satisfeita em entender algo;
porque, mesmo compreendendo-o tão bem como eu posso,
minha compreensão é apenas uma fração infinitesimal
de tudo que eu quero entender..."
(Ada Lovelace)

Resumo

Funções utilitárias são funções que oferecem serviços genéricos e que são úteis em várias partes de um sistema, como conversões de dados, manipulação de datas e de estruturas, dentre outros. Idealmente, devem ser implementadas em módulos próprios, para facilitar o reuso. No entanto, desenvolvedores frequentemente implementam funções utilitárias em módulos projetados para conter funções de propósito específico. Desta forma, diminuem-se as chances de reuso, causando retrabalho e facilitando duplicação de código. Uma solução para esse problema é mover a função utilitária para um módulo adequado. Por outro lado, esse tipo de refatoração não é trivial para desenvolvedores de grandes sistemas, devido à falta de ferramentas de suporte para identificar tais funções. Para suprir tal necessidade, nesta dissertação propõe-se um conjunto de heurísticas para identificação de funções utilitárias. Inicialmente, foi feita uma investigação para resolver o problema de pesquisa proposto usando aprendizado de máquina. Foram realizadas avaliações por projeto (*intra-project*) e usando vários projetos (*cross-project*) nas fases de treinamento e teste, sendo essa última mais próxima de um cenário de aplicação prática. Apesar do classificador apresentar ótimos resultados na avaliação *intra-project*, eles foram muito inferiores na abordagem *cross-project*. Portanto, decidiu-se propor heurísticas para identificar funções utilitárias, as quais podem ser computadas por meio de análise estática de código fonte em Java e em JavaScript, sem necessidade de treinamento. Avaliando as heurísticas propostas em quatro sistemas proprietários de uma empresa de desenvolvimento de software brasileira, obteve-se uma precisão média de 68%. Além disso, em um *survey* realizado com 33 desenvolvedores, obteve-se uma precisão de 66% e 67%, respectivamente, para Java e JavaScript, considerando uma amostra das funções de sistemas de código aberto do *GitHub* e do *Qualitas Corpus*.

Palavras-chave: Funções utilitárias, Refatoração, Arquitetura de Software, Modularização, Aprendizado de Máquina.

Abstract

Utility functions are functions that offer generic services — such as data conversions, manipulation of dates and structures, among others — and which are useful in many parts of a system. Ideally, they should be implemented in appropriate modules, to facilitate reuse. However, developers often implement utility functions in modules designed to contain specific purpose functions. Thus, this practice decreases opportunities of reuse, causing rework and duplicating code. Moving the utility function to a suitable module is a solution for this problem. However, this type of refactoring is not trivial for large system developers, due to the lack of supporting tools to identify such functions. To address this shortcoming, we proposed a set of heuristics to identify utility functions. Initially, we investigate the use of machine learning to solve the proposed research problem. We conducted evaluations by project (intra-project) and using many projects (cross-project) in training and testing phases. This last approach is closer to a practical application scenario. Despite the classifier has showed excellent results in intra-project evaluation, they were much lower in the cross-project scenario. Therefore, we proposed heuristics to identify utility functions, which can be computed by static analysis of Java and JavaScript source code, with no training phase. We obtained an average precision of 68% evaluating the proposed heuristics in four proprietary systems of a Brazilian software development company. Furthermore, in a survey performed with 33 developers, we obtained a precision of 66% and 67%, respectively, for Java and JavaScript, considering a sample of open source systems functions from *GitHub* and *Qualitas Corpus*.

Keywords: Utility functions, Refactoring, Software Architecture, Modularization, Machine Learning.

Lista de Figuras

2.1 Exemplo de curva ROC. 14
2.2 Funcionamento do algoritmo *Random Forests*. 18

3.1 Distribuição de (a) número de funções, (b) porcentagem de funções utilitárias. 25

4.1 Distribuição das medidas de acurácia. 34
4.2 Distribuição de *ranking* dos cinco melhores preditores. 38
4.3 Distribuição dos melhores preditores do sistema *Apache Collections* (AUC=0.99). 39
4.4 Distribuição dos melhores preditores do sistema *Freemind* (AUC=0.72). . . 40
4.5 Distribuição dos melhores preditores do sistema *Typeahead* (AUC=0.95). . 41
4.6 Distribuição dos melhores preditores do sistema *Mocha* (AUC=0.60). . . . 42
4.7 Distribuição do preditor *this* nos sistemas com maior AUC em JavaScript. 43
4.8 Modelo de entrada de dados de treinamento e de teste para o *Random Forests* em cada sistema. 45
4.9 Resultados da validação *cross-project* comparados às avaliações por projeto. 47

5.1 Resultados da avaliação das heurísticas comparados à validação *cross-project* nos sistemas proprietários. 57
5.2 Experiência dos participantes em desenvolvimento de software. 63
5.3 Experiência dos participantes nas linguagens Java e JavaScript. 63
5.4 Classificação das funções de acordo com as respostas dos formulários. . . . 65

Lista de Tabelas

2.1 Matriz de Confusão. 13
2.2 Exemplo de classificação (baseada em Han et al. [2011]). 14

3.1 Sistemas proprietários utilizados neste trabalho. 25
3.2 Resultados do estudo exploratório. 28

4.1 Lista de preditores para os sistemas em Java — métricas por método. . . . 32
4.2 Lista de preditores para os sistemas em JavaScript — métricas por função. 33
4.3 *Ranking* de preditores para Java organizados pelo teste *Scott-Knott*. 36
4.4 *Ranking* de preditores para JavaScript organizados pelo teste *Scott-Knott*. 37
4.5 Indicador de diferença estatística (*p-value*) entre métodos utilitários e não utilitários para os melhores preditores dos sistemas em Java com maior AUC. 39
4.6 Indicador de diferença estatística (*p-value*) entre métodos utilitários e não utilitários para os melhores preditores dos sistemas em Java com menor AUC. 40
4.7 Indicador de diferença estatística (*p-value*) entre funções utilitárias e não utilitárias para os melhores preditores dos sistemas em JavaScript com maior AUC. 41
4.8 Indicador de diferença estatística (*p-value*) entre funções utilitárias e não utilitárias para os melhores preditores dos sistemas em JavaScript com menor AUC. 42
4.9 Resultados do *Random Forests* nos sistemas proprietários. 44
4.10 Matrizes de confusão do Sistema A (JavaScript). 45
4.11 Matrizes de confusão do Sistema B (Java). 45
4.12 Matrizes de confusão do Sistema C (Java). 46
4.13 Matrizes de confusão do Sistema D (JavaScript). 46
4.14 Resultados da validação *cross-project* nos sistemas proprietários. 48

5.1 Resultado da identificação de funções utilitárias em módulos não utilitários. 56
5.2 Resultado da identificação de funções utilitárias em todo o sistema. 56

5.3 Comparação dos resultados da identificação de funções utilitárias entre a configuração padrão e uma configuração personalizada usando o *Random Forests*. 58
5.4 Número de funções identificadas por meio das heurísticas e de funções avaliadas nos formulários. 62
5.5 Número de formulários enviados, respondidos e de respostas com erro nas funções de controle. 62
5.6 Número de respostas para os pontos de controle. 64

A.1 84 sistemas de código aberto em Java do *Qualitas Corpus*. 71
A.2 22 sistemas de código aberto em JavaScript do *GitHub*. 74

Sumário

Agradecimentos **vii**

Resumo **xi**

Abstract **xiii**

Lista de Figuras **xv**

Lista de Tabelas **xvii**

1 Introdução **1**

1.1 Motivação 1

1.2 Visão Geral da Abordagem Proposta 3

1.3 Organização da Dissertação 4

1.4 Publicações 5

2 Fundamentação Teórica **7**

2.1 Funções Utilitárias 7

2.2 Recomendações de Refatoração 9

2.2.1 Refatoração 9

2.2.2 Sistemas de Recomendação 10

2.3 Aprendizado de Máquina 11

2.3.1 Medidas de Desempenho 12

2.3.2 Validação Cruzada 15

2.4 *Random Forests* 15

2.4.1 Árvores de Decisão 15

2.4.2 Métodos *Ensemble* 16

2.4.3 *Random Forests* 17

2.5 Outros Algoritmos 19

2.5.1 *Naive Bayes* . . . 19
2.5.2 Redes Neurais . . . 19
2.5.3 *k-Nearest-Neighbor* . . . 20
2.5.4 Máquina de Vetores de Suporte . . . 20
2.5.5 Algoritmos Genéticos . . . 20
2.6 Considerações Finais . . . 21

3 Dataset **23**
3.1 Critério para Definição de Módulo Utilitário . . . 23
3.2 Sistemas de Código Aberto . . . 23
3.3 Sistemas Proprietários . . . 25
3.3.1 Estudo Exploratório . . . 26
3.4 Considerações Finais . . . 28

4 Identificação de Funções Utilitárias Usando Aprendizado de Máquina **31**
4.1 Projeto do Estudo . . . 31
4.2 Preditores . . . 32
4.3 Avaliação *Intra-Project* . . . 33
4.3.1 Avaliação nos Sistemas de Código Aberto . . . 34
4.3.2 Avaliação nos Sistemas Proprietários . . . 43
4.4 Validação *Cross-Project* . . . 46
4.4.1 Avaliação nos Sistemas de Código Aberto . . . 47
4.4.2 Avaliação nos Sistemas Proprietários . . . 48
4.5 Ameaças à Validade dos Resultados . . . 48
4.6 Considerações Finais . . . 49

5 Heurísticas para Identificação de Funções Utilitárias **51**
5.1 Heurísticas Propostas . . . 51
5.2 Avaliação nos Sistemas Proprietários . . . 56
5.2.1 Usando o *Random Forests* para Configurar um Recomendador de Funções Utilitárias . . . 57
5.3 Survey com Desenvolvedores . . . 59
5.3.1 Resultados . . . 62
5.4 Ameaças à Validade dos Resultados . . . 65
5.5 Considerações Finais . . . 66

6 Conclusão **67**
6.1 Contribuições . . . 68

6.2 Limitações 68
6.3 Trabalhos Futuros 69

Apêndice A Dataset **71**

Apêndice B Formulário Usado no Survey **75**

Referências Bibliográficas **83**

Capítulo 1

Introdução

Neste capítulo, são expostos o problema e a motivação desta dissertação de mestrado (Seção 1.1). Em seguida, apresenta-se uma visão geral da solução proposta (Seção 1.2). Finalmente, descreve-se a organização da dissertação (Seção 1.3) e listam-se as publicações resultantes da pesquisa (Seção 1.4).

1.1 Motivação

Funções utilitárias são funções de propósito geral, que não implementam regras de negócio e podem ser reusadas em diversas partes de um sistema e de diferentes sistemas. Elas auxiliam outras funções, provendo alguma funcionalidade genérica, como conversões de dados, manipulação de *string*, de datas, de estruturas de dados, dentre outros. Convencionalmente, funções utilitárias são definidas em bibliotecas próprias, para facilitar o reuso. Porém, frequentemente, desenvolvedores implementam funções utilitárias junto a funções de propósito específico. Por exemplo, ao analisar a modularização de uma aplicação bancária com problemas de manutenção (com 25 milhões de linhas de código e mais de 100 instalações em 50 países), Sarkar et al. [2009] reportaram como uma das causas desses problemas, o fato de funções de baixa granularidade, como validação de datas, serem implementadas nas mesmas bibliotecas e arquivos em que se encontram funções de domínio complexas. Consequentemente, dificulta-se o reuso de funcionalidades, causando retrabalho e aumentando as chances de se gerar código duplicado. Alguns exemplos da ocorrência desse problema em sistemas de código aberto do *GitHub* são mostrados nos códigos a seguir.

```
1 /* mocha/lib/browser/events.js */
2 function isArray(obj) {
3     return "[object Array]" == {}.toString.call(obj);
4 }
```

Código 1.1: Exemplo de função utilitária em JavaScript, do sistema *Mocha*, implementada fora de um módulo utilitário.

```
1 /* jgroups/src/org/jgroups/conf/XmlConfigurator.java */
2 public static String replace(String input, final String expr,
3                                                 String replacement) {
4     StringBuilder sb=new StringBuilder();
5     int new_index=0, index=0, len=expr.length(), input_len=input.length();
6
7     while(true) {
8         new_index=input.indexOf(expr, index);
9         if(new_index == -1) {
10            sb.append(input.substring(index, input_len));
11            break;
12        }
13        sb.append(input.substring(index, new_index));
14        sb.append(replacement);
15        index=new_index + len;
16    }
17
18    return sb.toString();
19 }
```

Código 1.2: Exemplo de método utilitário em Java, do sistema *JGroups*, implementado fora de um módulo utilitário.

```
1 /* brackets/src/language/HTMLTokenizer.js */
2 function isWhitespace(c) {
3     return c === " " || c === "\t" || c === "\r" || c === "\n";
4 }
```

Código 1.3: Exemplo de função utilitária em JavaScript, do sistema *Brackets*, implementada fora de módulo utilitário.

O Código 1.1 mostra uma função utilitária para verificar o tipo de um objeto do sistema *Mocha* em JavaScript (um *framework* de testes para JavaScript). Como um segundo exemplo, o Código 1.2 apresenta um método para manipulação de *strings* do sistema *JGroups* em Java (um *toolkit* para comunicação em grupo). O Código 1.3 exibe uma função do sistema *Brackets* em JavaScript (editor de código para JavaScript, HTML e CSS) que verifica se um caractere é um espaço em branco. Essas funções fornecem serviços com propósitos genéricos e podem ser usadas em diferentes tipos de

sistemas[1]. Contudo, as três funções foram implementadas em módulos não utilitários, isto é, junto a funções de propósito específico. Esse problema pode ser visto como uma forma de *Feature Envy* — tipo de *bad smell* no qual as responsabilidades de um método estão mais relacionadas a outra classe do que à própria [Fowler, 1999]. Uma das formas de remover esse *bad smell* é aplicar uma refatoração para mover a função para um módulo mais adequado. No caso das funções genéricas implementadas em módulos de propósito específico, sugere-se que elas sejam movidas para um módulo utilitário. No entanto, identificar essas funções não é uma tarefa trivial para os desenvolvedores em grandes sistemas. Logo, um sistema de recomendação para identificar oportunidades de movimentação de funções utilitárias pode auxiliar os desenvolvedores na identificação desses problemas, principalmente quando estão trabalhando em sistemas com grande quantidade de componentes e funções.

1.2 Visão Geral da Abordagem Proposta

Não foram encontradas na literatura analisada soluções ou ferramentas voltadas para modularização de funções utilitárias. Existem sistemas propostos para mover métodos implementados em módulos inadequados, mas eles não são especializados em funções utilitárias [Fokaefs et al., 2007; Sales et al., 2013; Bavota et al., 2014]. Além disso, os trabalhos mencionados são implementados apenas para Java. Apesar de JavaScript ter uma grande comunidade de desenvolvedores e usuários e estar em expansão [Tiwari & Solihin, 2012], existem poucos sistemas de recomendação e refatoração implementados para essa linguagem, quando comparada a outras linguagens mais maduras e estaticamente tipadas, como Java. Isso porque suas características, como tipagem fraca e dinâmica, tornam o desenvolvimento deste tipo de ferramenta mais desafiador [Richards et al., 2010]. Assim, para fornecer uma alternativa ao problema apresentado, nesta dissertação de mestrado propõe-se um conjunto de heurísticas para identificar funções utilitárias implementadas em módulos inadequados de sistemas Java e JavaScript. O objetivo é alertar os desenvolvedores de que essas funções podem ser movidas para módulos utilitários.

Inicialmente, investigou-se a possibilidade de identificar funções utilitárias usando aprendizado de máquina. Um classificador *Random Forests* foi treinado usando uma lista de preditores baseados em métricas estáticas de código e testou-se sua habilidade em classificar funções dos sistemas proprietários e de código aberto. Esse classificador foi avaliado de duas formas: (a) *intra-project*, no qual a classificação é feita por sistema

[1]Nesta dissertação, usa-se a palavra "função" também para denotar métodos Java, visando uniformizar os termos em Java e JavaScript.

e (b) *cross-project*, que usa diferentes sistemas nas fases de treinamento e teste, e é considerada uma abordagem mais próxima de um cenário de aplicação prática. Obteve-se resultados muito bons utilizando a abordagem *intra-project*. No entanto, esses resultados não se mantiveram na avaliação *cross-project*. Logo, a aplicação do classificador em um cenário prático pode ficar comprometida.

Diante dos resultados não satisfatórios obtidos na investigação da técnica de aprendizado de máquina na classificação de funções utilitárias, decidiu-se propor um conjunto de heurísticas baseadas em métricas de código que podem ser obtidas via análise estática. Tais métricas foram selecionadas para representar características comuns a funções utilitárias. Esse conjunto de heurísticas também pode ser facilmente integrado a ferramentas utilizadas nos ambientes de desenvolvimento de software atuais. Como resultado, obteve-se uma precisão média de 68% na identificação de funções utilitárias, por meio das heurísticas propostas, em uma avaliação com quatro sistemas proprietários de uma empresa de desenvolvimento de software brasileira. Além disso, em um *survey* realizado com 33 desenvolvedores, avaliando uma amostra das funções de 84 sistemas em Java e 22 sistemas em JavaScript de código aberto, obteve-se uma precisão de 66% para Java e 67% para JavaScript na identificação de funções utilitárias implementadas em módulos projetados para conter funções de propósito específico.

Concluindo, as principais contribuições são sumarizadas a seguir:

- Classificação manual de funções, qualificadas como utilitárias ou não, em quatro sistemas proprietários de uma empresa de desenvolvimento de software brasileira;
- Investigação da aplicação de aprendizado de máquina para identificar funções utilitárias;
- Um conjunto de heurísticas baseadas em métricas estáticas de código para identificar funções utilitárias em Java e JavaScript;
- Avaliação de duas abordagens na identificação de funções utilitárias em quatro sistemas proprietários e 106 sistemas de código aberto em Java e JavaScript;
- Um *survey* realizado com 33 desenvolvedores que classificaram funções de sistemas de código aberto como utilitárias ou não.

1.3 Organização da Dissertação

O restante desta dissertação é organizado conforme descrito a seguir:

- **Capítulo 2**: descreve conceitos centrais relacionados à dissertação, incluindo definições sobre funções utilitárias e refatoração. Também são relatados trabalhos que propõem recomendações de *Move Method*. Por fim, são apresentados fundamentos na área de Aprendizado de Máquina, com destaque para o classificador *Random Forests*.

- **Capítulo 3**: apresenta o *dataset* usado nesta dissertação, assim como o critério adotado para definir módulos utilitários no mesmo. Um estudo exploratório realizado para verificar hipóteses que viabilizam a pesquisa também é descrito.

- **Capítulo 4**: discute uma investigação sobre a identificação de funções utilitárias usando aprendizado de máquina. São reportadas avaliações *intra-project* e *cross-project* nos sistemas do *dataset* considerado.

- **Capítulo 5**: descreve um conjunto de heurísticas propostas para identificar funções utilitárias em Java e JavaScript. São reportadas as avaliações nos sistemas do *dataset*, incluindo os resultados de um *survey* realizado com 33 desenvolvedores de software.

- **Capítulo 6**: apresenta as considerações finais da dissertação, incluindo as principais contribuições, limitações e trabalhos futuros.

1.4 Publicações

Esta dissertação produziu as seguintes publicações e, portanto, contém material das mesmas:

- Mendes, T.; Valente, M. T.; Hora, A. & Serebrenik, A. (2016b). Identifying utility functions using Random Forests. Em *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pp. 614–618.

- Mendes, T.; Valente, M. T. & Hora, A. (2016a). Identificação de funções utilitárias em Java e Javascript. Em *X Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS)*.

Capítulo 2

Fundamentação Teórica

Neste capítulo, são apresentados alguns conceitos nos quais o trabalho se baseia, como o de funções utilitárias (Seção 2.1), sistemas de recomendação e refatoração (Seção 2.2). Os trabalhos relacionados também são discutidos na Seção 2.2. Além disso, na Seção 2.3 são apresentados fundamentos na área de Aprendizado de Máquina. O *Random Forests*, em especial, é tratado na Seção 2.4. Apresenta-se ainda outros algoritmos de aprendizado de máquina na Seção 2.5. Por fim, são apresentadas as conclusões na Seção 2.6.

2.1 Funções Utilitárias

Funções utilitárias são funções de propósito geral, úteis em diversas partes de um sistema ou de outros sistemas. São usadas para auxiliar outras funções fornecendo alguma funcionalidade comum, como conversões de dados, manipulação de *strings*, de datas e de estruturas. Normalmente são definidas em bibliotecas específicas, para facilitar o reuso.

De acordo com algumas definições propostas na comunidade de programadores *Stack Overflow*[1], funções utilitárias são funções que proveem serviços genéricos a múltiplos clientes. Normalmente, são funções globais que não precisam ser encapsuladas em objetos. Por esse motivo, em Java, costumam ser implementadas em métodos estáticos. Segundo Bloch [2008], não é adequado instanciar classes utilitárias. Porém, não há regras determinando que elas devam ser estáticas.

As características de uma função utilitária variam de acordo com o sistema. No entanto, a maioria delas segue algumas convenções gerais das comunidades de desenvolvimento de software. Por exemplo, funções utilitárias não devem implementar regras

[1] `http://stackoverflow.com/questions/25060976`

de negócio, pois a ideia é que elas disponibilizem serviços genéricos, que possam ser reusados em outros sistemas. Normalmente, todos — ou quase todos — os objetos necessários são passados via parâmetro, portanto, não há necessidade de acessar variáveis globais e funções externas, na maioria das vezes. Quando acessam, geralmente são constantes declaradas no próprio módulo utilitário ou funções de bibliotecas externas. Inclusive, é comum que essas funções sejam usadas para encapsular funcionalidades de bibliotecas externas. Ou seja, as funções utilitárias geralmente têm baixo acoplamento com módulos do sistema, mas podem depender módulos externos. Espera-se, também, que uma função utilitária, após executar alguma funcionalidade, retorne algum resultado. Logo, a maioria das funções possuem retorno diferente de `void` (menos de 35% das funções em módulos utilitários dos sistemas do *dataset* usado nesta dissertação de mestrado não possuem retorno ou são métodos do tipo `void`). Construtores, interfaces, *getters* e *setters* não são considerados utilitários. Alguns exemplos de funções utilitárias em Java e JavaScript são mostrados nos códigos a seguir.

```
1 /* typeahead/src/common/utils.js */
2 toStr: function toStr(s) {
3     return (_.isUndefined(s) || s === null) ? "" : s + "";
4 }
```

Código 2.1: Exemplo de função utilitária em JavaScript do *Typeahead*.

```
1 /* jsXe/src/net/sourceforge/jsxe/MiscUtilities.java */
2 public static String getFileExtension(String name) {
3     int index = name.indexOf(".");
4     if(index == -1)
5         return "";
6     else
7         return name.substring(index);
8 }
```

Código 2.2: Exemplo de método utilitário em Java do *jsXe*.

```
1 /* timelinejs/source/js/Core/Core/VMM.Util.js */
2 toCamelCase: function(s, forceLowerCase) {
3     if(forceLowerCase !== false) forceLowerCase = true;
4
5     var sps = ((forceLowerCase) ? s.toLowerCase() : s).split(" ");
6     for(var i = 0; i < sps.length; i++) {
7         sps[i] = sps[i].substr(0,1).toUpperCase() + sps[i].substr(1);
8     }
9     return sps.join(" ");
10 }
```

Código 2.3: Exemplo de função utilitária em JavaScript do *TimelineJS*.

```
1 /* cobertura/src/net/sourceforge/cobertura/util/StringUtil.java */
2 public static String getPercentValue(double value) {
3     //moved from HTMLReport.getPercentValue()
4     value = Math.floor(value * 100) / 100;
5     return NumberFormat.getPercentInstance().format(value);
6 }
```

Código 2.4: Exemplo de método utilitário em Java do *Cobertura*.

```
1 /* react/src/renderers/dom/client/utils/DOMChildrenOperations.js */
2 function insertChildAt(parentNode, childNode, index) {
3     var beforeChild = index >= parentNode.childNodes.length ? null
4                         : parentNode.childNodes.item(index);
5
6     parentNode.insertBefore(childNode, beforeChild);
7 }
```

Código 2.5: Exemplo de função utilitária em JavaScript do *React*.

O Código 2.1 em JavaScript é uma função do *Typeahead* que converte um objeto em *string*, retornando uma *string* vazia caso o objeto seja nulo. O Código 2.2 mostra um método Java do sistema *jsXe*. Dada uma *string* com o nome de um arquivo, ele retorna a extensão do mesmo. O Código 2.3 em JavaScript do sistema *TimelineJS* converte um conjunto de palavras para o formato *CamelCase*, no qual as palavras se iniciam com letra maiúscula. No Código 2.4 em Java do sistema *Cobertura*, um valor decimal é convertido para o formato de porcentagem em *string*. Nesse método há um comentário interessante mostrando que aquele método foi implementado inicialmente em um módulo não utilitário (*"HTMLReport"*), sendo movido para a classe *"StringUtil"*. Finalmente, o Código 2.5 mostra uma função em JavaScript do sistema *React* que insere um elemento do DOM na posição desejada dentro de outro elemento do DOM.

2.2 Recomendações de Refatoração

2.2.1 Refatoração

Sistemas de software precisam evoluir constantemente para não se tornarem obsoletos. No entanto, à medida que o código de um sistema é modificado, incluindo novos requisitos e corrigindo defeitos, sua complexidade aumenta e sua estrutura se desvia da arquitetura original gradualmente [Lehman, 1980]. Uma das formas de evitar e corrigir esses problemas é realizando refatorações. Segundo Fowler [1999], refatoração é o processo de modificar um software para melhorar sua estrutura interna, preservando

seu comportamento externo. É uma forma disciplinada de melhorar a arquitetura e minimizar as chances de introduzir defeitos.

Renomear classes, métodos e atributos é um dos tipos de refatoração mais comuns, realizada com o objetivo de melhorar a legibilidade e o entendimento do código. Outro exemplo de refatoração é o *Extract Method* que remove parte de um código dentro de um método e insere em um novo método. Assim, diminui-se a complexidade de métodos longos. O *Inline Method* é exatamente o oposto. Substituem-se as chamadas a um método por seu corpo e remove-se o método. Nesta pesquisa, o tipo de refatoração abordado é o *Move Method*, no qual um método é movido para outra classe. Ela é realizada quando as responsabilidades do método estão mais relacionadas a outra classe do que à classe original na qual ele foi implementado, ou quando essa classe tem muitas responsabilidades. Isso ajuda a simplificar a classe, diminuir o acoplamento, e melhorar a coesão [Fowler, 1999].

A refatoração é um processo importante para melhorar a qualidade de um software. Entretanto, é comum que ela não seja praticada devido a problemas com prazo, recursos e orçamento. Por outro lado, negligenciar esse processo pode trazer custos ainda maiores com a manutenção do sistema. Desta forma, ferramentas que automatizam o processo, ou parte dele, são importantes para viabilizar sua prática reduzindo custos [Fowler, 1999].

2.2.2 Sistemas de Recomendação

Não é trivial construir ferramentas completamente automatizadas que realizam refatorações [Tsantalis & Chatzigeorgiou, 2009]. Segundo Terra et al. [2015], implementá-las é uma tarefa complexa mesmo para as refatorações típicas. Os sistemas de recomendação são uma alternativa às soluções plenamente automáticas. São sistemas para ajudar os desenvolvedores a tomarem decisões principalmente quando estão trabalhando com grande quantidade de informação — tecnologias, componentes, classes, etc. Os sistemas de recomendação auxiliam os desenvolvedores em diversos tipos de atividades como, por exemplo, refatoração, reuso de código, histórico de versão, re-modularização, localização e correção de erros, dentre outros [Robillard et al., 2010; Sales et al., 2013; Silva et al., 2014].

Nesta dissertação de mestrado, a proposta é identificar funções utilitárias implementadas em módulos inadequados e então recomendar que elas sejam movidas para módulos apropriados. Esse problema com funções utilitárias pode ser visto como um tipo de *Feature Envy*. *Feature Envy* é uma categoria de *bad smell* na qual um método parece estar mais interessado em outra classe do que na própria, ou seja, as responsa-

bilidades do método estão mais relacionadas a outra classe do sistema. Logo, aplicar a refatoração *Move Method* é uma das formas de resolver esse problema [Fowler, 1999]. Alguns exemplos de sistemas de recomendação voltados para esse tipo de refatoração incluem o *JDeodorant* [Fokaefs et al., 2007], o *JMove* [Sales et al., 2013] e o *Methodbook* [Bavota et al., 2014].

JDeodorant é um sistema que recomenda refatoração de *bad smells* como o *Feature Envy* [Fokaefs et al., 2007]. Quando um método acessa mais atributos e métodos de uma classe do que da própria, ele pode ser recomendado para ser movido. Para que isso seja possível, o *JDeodorant* define um conjunto de precondições para verificar se a refatoração pode ser aplicada sem erros. São verificadas precondições para garantir que o sistema compile corretamente, para certificar que o comportamento do código seja preservado e que métricas de qualidade de projeto não sejam violadas [Tsantalis & Chatzigeorgiou, 2009]. *JMove* [Sales et al., 2013] é um sistema que recomenda refatorações *Move Method* baseado na ideia de que os métodos de uma classe bem estruturada devem ter dependência de tipos similares. De modo simplificado, quando um método possui dependências mais similares a métodos de outras classes do que aos métodos da própria classe, recomenda-se a refatoração. O *Methodbook* [Bavota et al., 2014] é outro sistema de recomendação de *Move Method*. Além de considerar as dependências estruturais entre os métodos (chamadas e atributos), ele também analisa a informação textual presente no código (comentários e identificadores) para encontrar relação semântica entre os métodos.

No entanto, esses sistemas foram propostos para mover qualquer tipo de método, sem tratar particularidades de funções utilitárias. Além disso, todos eles são implementados em Java e são fortemente dependentes de informações de tipo. Logo, teriam que ser adaptados para sistemas que usam linguagens dinâmicas, como JavaScript, o que, possivelmente, não é uma tarefa trivial.

2.3 Aprendizado de Máquina

Aprendizado de Máquina é uma área da Inteligência Artificial inspirada na capacidade de aprendizagem dos humanos. Algoritmos de aprendizado são programas que tomam decisões com base em experiências adquiridas a partir de boas soluções de problemas passados [Mitchell, 1997; Russell & Norvig, 1995]. Eles têm sido desenvolvidos e aplicados na prática para resolver vários tipos de problemas em diversas áreas como: identificação de genes e doenças, classificação de câncer e proteínas [Guyon et al., 2002], reconhecimento de fala, predição de mortalidade de pacientes com pneumonia, detec-

ção de uso fraudulento de cartões de crédito, condução automática de veículos, dentre outros [Mitchell, 1997].

Os algoritmos de aprendizado de máquina observam um padrão e aprendem, a partir de **exemplos** de entrada, a tomar decisões reconhecendo esses padrões automaticamente. Geralmente, cada exemplo é composto por:

- ***preditores***: vetor de entrada com determinados atributos;
- ***classe***: saída esperada na classificação, isto é, o valor do qual deseja-se obter previsões.

Formalmente, cada exemplo é um par $(x_i, f(x_i))$ no qual x_i é o vetor de entrada e $f(x_i)$ é a classe de saída da predição. O algoritmo recebe esses exemplos e induz um **classificador** de função $h(x_i)$ que se aproxima de $f(x_i)$. Dadas novas entradas, esse classificador prediz a classe daquele conjunto.

Segundo Mitchell [1997], um programa de computador aprende com a experiência E para alguma classe de tarefas T e medida de performance P se sua performance nas tarefas em T, medida por P, melhora com a experiência E. Por exemplo, em um algoritmo para reconhecer palavras escritas à mão, a classe de tarefas T é reconhecer e classificar palavras escritas a mão disponíveis em imagens. A performance P é a porcentagem de palavras classificadas corretamente. A fonte de experiência E é uma base de dados de imagens de palavras escritas a mão com suas respectivas classificações corretas. Nesta dissertação de mestrado, a classe de tarefas T é reconhecer e classificar funções utilitárias. A performance P é a porcentagem de funções classificadas corretamente. A fonte de experiência E é um conjunto de funções classificadas como utilitárias ou não utilitárias.

2.3.1 Medidas de Desempenho

Para avaliar o desempenho de classificadores, utilizam-se algumas medidas como a matriz de confusão, a curva ROC e a AUC, que serão descritas a seguir.

Matriz de Confusão

A matriz de confusão é uma tabela que mostra, de forma fragmentada, o número de erros e acertos na predição do classificador (ver Tabela 2.1). Verdadeiros positivos são o número de classes positivas classificadas como positivas; falsos negativos são o número de classes positivas indicadas como negativas; falsos positivos são o número

de classes negativas previstas como positivas; e verdadeiros negativos são o número de classes negativas classificadas como negativas [Osman et al., 2013]. A diagonal principal corresponde aos acertos do classificador e os demais elementos representam o número de classes previstas incorretamente.

Tabela 2.1: Matriz de Confusão.

	Predição Positiva	**Predição Negativa**
Classe Positiva	Verdadeiro Positivo	Falso Negativo
Classe Negativa	Falso Positivo	Verdadeiro Negativo

ROC e AUC

A curva ROC (*Receiver Operating Characteristics*) é uma ferramenta visual para comparar classificações [Han et al., 2011]. A taxa de verdadeiro positivo (TVP) forma o eixo vertical do gráfico e a taxa de falso positivo (TFP), o eixo horizontal. Dados VP e FP, respectivamente, o número de exemplos verdadeiro positivo e falso positivo na predição, e P e N o número de exemplos com valor da classe positivo e negativo, temos que:

$$TVP = \frac{VP}{P} \qquad \text{e} \qquad TFP = \frac{FP}{N}$$

Para construir a curva ROC, o classificador deve fornecer a probabilidade de cada exemplo ser verdadeiro, já que a plotagem do gráfico é feita com base na ordenação dos exemplos pela probabilidade de forma decrescente. Para cada exemplo, um ponto é marcado. Na primeira plotagem, considera-se o primeiro exemplo positivo e o restante dos exemplos negativos, calculam-se as taxas e plota-se. Para o segundo ponto, considera-se o primeiro e segundo exemplos positivos e o restante negativos, calculam-se as taxas e plota-se. E assim por diante, até o último exemplo.

Para exemplificar, a Tabela 2.2 exibe uma classificação com 4 exemplos positivos e 4 negativos ordenados pela probabilidade do exemplo ser positivo. A segunda coluna mostra a classe real do exemplo. A terceira coluna mostra a probabilidade de exemplo ser positivo. As colunas VP, FP, VN e FN mostram, respectivamente, o número de verdadeiros positivos, falsos positivos, verdadeiros negativos e falsos negativos considerando que, até aquela linha, todos os exemplos serão classificados como positivos e os demais como negativos. As duas últimas colunas mostram as taxas de verdadeiro positivo e falso positivo.

Tabela 2.2: Exemplo de classificação (baseada em Han et al. [2011]).

Ex.	**Classe**	**Prob.**	**VP**	**FP**	**VN**	**FN**	**TVP**	**TFP**
1	P	0.91	1	0	4	3	0.25	0
2	P	0.85	2	0	4	2	0.5	0
3	N	0.79	2	1	3	2	0.5	0.25
4	P	0.66	3	1	3	1	0.75	0.25
5	N	0.58	3	2	2	1	0.75	0.5
6	N	0.52	3	3	1	1	0.75	0.75
7	P	0.47	4	3	1	0	1	0.75
8	N	0.40	4	4	0	0	1	1

O gráfico na Figura 2.1 exibe a plotagem das duas últimas colunas da Tabela 2.2: taxas de verdadeiro positivo e taxas de falso positivo, que formam a curva ROC para a classificação do exemplo.

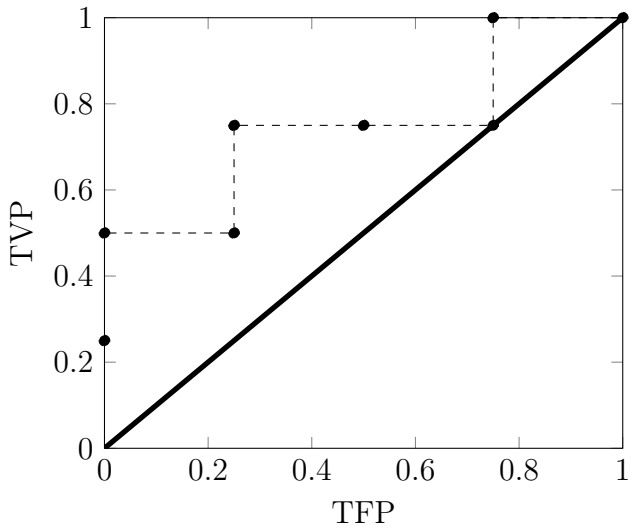


Figura 2.1: Exemplo de curva ROC.

O gráfico (Figura 2.1) também mostra uma linha diagonal na qual as taxas de verdadeiro positivo e falso positivo são iguais. De acordo com Han et al. [2011], quanto mais próxima a curva ROC estiver da diagonal, menos precisa é a classificação, já que ela se aproxima de uma função aleatória. Logo, quanto maior a área sob a curva, melhor é a classificação, ou seja, quando a curva está mais distante e acima da diagonal. A classificação perfeita é aquela que ao ordenar os exemplos, os primeiros são todos positivos e o restante negativos, de forma que a curva seja traçada sobre o eixo y ($x = 0$) e na reta superior ($y = 1$) do gráfico, e a área sob a curva seja igual a 1.

A medida que se obtém calculando a área abaixo da curva ROC é chamada

AUC (*Area Under ROC Curve*). Ela revela a habilidade do algoritmo em classificar corretamente sua entrada. Seus valores variam de 0 a 1 e quanto mais próximo de 1, melhor é o desempenho do classificador [Vuk & Curk, 2006]. Valores abaixo de 0.6 são considerados resultados produzidos aleatoriamente [Osman et al., 2013].

2.3.2 Validação Cruzada

Validação cruzada é um método de amostragem para avaliar quão bem o classificador prevê dados não utilizados no treinamento, separando uma parte dos dados com classe conhecida para testá-lo. A validação (*k-fold cross validation*), de acordo com Mitchell [1997], consiste em dividir n exemplos da entrada do classificador aleatoriamente em k grupos (*folds*), mutuamente exclusivos de tamanho aproximado n/k. Um dos k *folds* é usado para testar e os $k-1$ restantes são usados para treinamento. Esse processo é repetido k vezes de modo que todos os *folds* sejam usados no conjunto de teste.

2.4 *Random Forests*

Dentre os algoritmos de classificação existentes, destaca-se o *Random Forests*. Seu funcionamento será descrito nesta seção. Para isso, serão introduzidos alguns conceitos para facilitar seu entendimento: árvores de decisão, métodos *ensemble* e método *Bagging*.

2.4.1 Árvores de Decisão

Um dos mais simples e eficazes métodos de aprendizado de máquina é por meio de árvores de decisão. Elas representam funções com valores discretos e são similares a um conjunto de regras *if-then*. Têm como entrada um conjunto de atributos — os preditores. Cada nó contém um teste de um preditor. Cada ramo descendente do nó é um possível valor desse preditor. Cada folha é uma classe, ou seja, o resultado da predição. E cada percurso da raiz à folha é uma regra de classificação [Russell & Norvig, 1995].

Para construir uma árvore de decisão, um preditor é escolhido para particionar os exemplos em subconjuntos correspondentes aos valores desse preditor. Para cada subconjunto, se os exemplos dele pertencem à mesma classe, associa-se essa classe a um nó folha. Do contrário, um novo preditor é escolhido para particionar a árvore, repetindo esse processo até que as classes estejam homogêneas em um subconjunto.

Quando o número de preditores é grande e o número de exemplos do conjunto de treinamento é, comparativamente, pequeno, ou quando existem muitas hipóteses, ou seja, os exemplos são muito particionados, pode-se chegar a regularidades sem sentido, e funções muito específicas podem ser geradas para aquele conjunto de dados. Esse problema é conhecido como *overfitting* e acontece nos algoritmos de aprendizado em geral e não apenas em árvores de decisão [Russell & Norvig, 1995; Guyon et al., 2002].

Apesar de ser um método popular, o modelo tradicional de uma única árvore de decisão pode ser instável, isto é, pequenas mudanças no conjunto de treinamento causam grandes mudanças no classificador. Modelos que combinam várias árvores, como serão mostrados nas próximas seções, tentam reduzir ou eliminar esse problema [Dietterich, 2000].

2.4.2 Métodos *Ensemble*

Muitos métodos tradicionais de modelos de aprendizagem assumem que as classes dos dados são bem distribuídas. Todavia, em muitos problemas do mundo real, as classes são desbalanceadas, de forma que a classe de interesse é representada apenas por alguns exemplos do conjunto de dados. Uma das técnicas para lidar com esse problema é o método *ensemble*.

Esse método combina um conjunto de k classificadores $C_1, C_2, ..., C_k$ com o intuito de criar um classificador composto melhorado. Para isso, com um conjunto de dados de entrada para o algoritmo criam-se k conjuntos de treinamento. Cada um deles é usado para formar um modelo de classificação. Quando um novo exemplo é dado para classificar, cada um dos classificadores vota retornando uma classe. Por fim, o algoritmo retorna a classe de predição baseada nos votos desses classificadores [Han et al., 2011].

Exemplos de algoritmos baseados em métodos *ensemble* são *Bagging, Boosting* e *Random Forests*. Han et al. [2011] fazem uma comparação entre diagnóstico médico e o método *Bagging*. Suponha um paciente que tem um diagnóstico feito com base em seus sintomas. O paciente consulta vários médicos ao invés de apenas um. Se certo diagnóstico ocorrer mais do que outros, esse provavelmente é o melhor diagnóstico, pois ele foi o mais votado, considerando que cada médico tem um voto de mesmo peso. Substituindo médicos por classificadores e diagnóstico por classe, tem-se uma ideia básica do algoritmo *Bagging*. Seu nome se origina de *bootstrap aggregation*. Nesse algoritmo, cada conjunto de treinamento é formado por n exemplos selecionados aleatoriamente, com reposição, da base de dados original. Dessa forma, alguns exemplos podem se repetir dentre os k conjuntos de treinamento ou podem ainda não serem

usados em nenhum deles. Gera-se um classificador diferente com cada conjunto de treinamento. Para predizer um exemplo, cada classificador escolhe uma classe e a mais votada é retornada pelo algoritmo. Ele tem precisão significativamente maior do que um único classificador derivado do conjunto de dados de treinamento e lida bem com instabilidade [Breiman, 1996].

2.4.3 *Random Forests*

Random Forests é um método de classificação *ensemble* criado por Breiman [2001]. Cada um dos classificadores é uma árvore de decisão e a coleção de árvores é uma floresta. Ele é construído usando o conceito de *Bagging* em combinação com seleção randômica de atributos.

O algoritmo funciona da seguinte maneira, descrita por Han et al. [2011]: dado um conjunto de dados da base de treinamento, cada uma das k árvores de decisão para o *ensemble* é gerada escolhendo-se d exemplos, com reposição, e formando um subconjunto de treinamento. Isto significa que os subconjuntos podem ter exemplos repetidos e alguns exemplos podem ser excluídos dos treinamentos, diferentemente da árvore de decisão tradicional, na qual todo o conjunto de dados é usado na sua construção. Em cada nó de cada árvore, f preditores, dos p disponíveis no total, são selecionados aleatoriamente para dividir o nó, sendo f um número bem menor do que os p preditores disponíveis. Mais formalmente, cada árvore depende dos valores de um vetor aleatório amostrado independentemente e com a mesma distribuição para todas as árvores da floresta [Breiman, 2001]. As árvores crescem até o tamanho máximo e não são podadas. Durante a classificação, cada uma das árvores formadas vota numa classe e a classe mais votada é retornada pelo algoritmo. A Figura 2.2 ilustra o algoritmo.

O algoritmo é efetivo como preditor, com bom desempenho e precisão. Observa-se que a precisão do algoritmo depende da força dos classificadores individuais e da medida de dependência entre eles (correlação). O ideal é manter a força de cada classificador sem aumentar a correlação. Quanto menor a similaridade entre duas árvores, melhor [Han et al., 2011].

O *Random Forests* está sendo largamente usado nas pesquisas em Engenharia de Software. Lessmann et al. [2008] comparam o desempenho de vários classificadores na tarefa de prever defeitos de software nos módulos de sistemas. O *Random Forests* foi um dos algoritmos com maior precisão em comparação com algoritmos como *Naive Bayes*, *k-Nearest-Neighbor*, Redes Neurais e Máquina de Vetores de Suporte, sendo recomendado para futuros experimentos na área. Abebe et al. [2012] também avaliam a detecção de defeitos de software usando o *Random Forests*, dentre outros algoritmos.

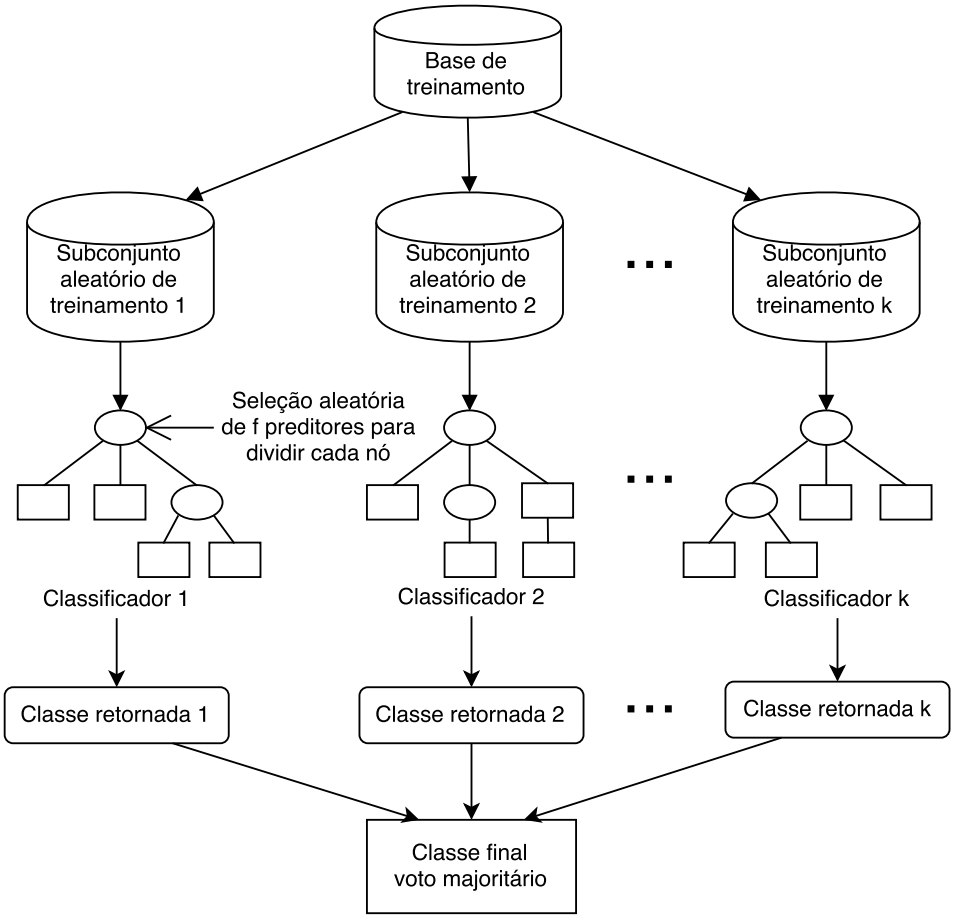


Figura 2.2: Funcionamento do algoritmo *Random Forests*.

Eles usaram métricas de *Bad Smells* léxicos (relacionados principalmente a nomes de identificadores inadequados) e métricas estruturais. Peters et al. [2013] é outro trabalho sobre detecção de defeitos que usa, dentre outros algoritmos, o *Random Forests* e faz um cruzamento de dados de treinamento com outros projetos para prever defeitos de software. Costa et al. [2014] construíram modelos com o *Random Forests* para prever o tempo de integração de *issues* resolvidas, e indicar em qual *release* ela acontecerá. Tian et al. [2015] usam o classificador para identificar quais características mais influenciam no sucesso de um aplicativo *Android*. Dias et al. [2015] fazem um estudo sobre o problema de modificações não relacionadas serem salvas em um mesmo *commit* — *tangled commit*. Para granular essas mudanças, eles usaram regressão logística, *Naive Bayes* e o *Random Forests* e os melhores resultados foram obtidos com esse último.

Os motivos para seu sucesso são, possivelmente, suas várias vantagens: é robusto a dados de entrada com ruído e *outliers* [Breiman, 2001]; *overfitting* não é um problema [Han et al., 2011]; a seleção randômica dos preditores em cada nó diminui a correlação entre as árvores, diminuindo assim sua taxa de erro [Archer & Kimes, 2008];

o número de preditores não é limitado e é eficiente em grandes bases de dados [Han et al., 2011]; e quando comparado a outros métodos, como redes neurais artificiais, *Bagging* e *Naive Bayes*, é considerado um dos mais precisos [Caruana et al., 2008].

2.5 Outros Algoritmos

Nesta seção serão descritos brevemente outros algoritmos de aprendizado de máquina.

2.5.1 *Naive Bayes*

O *Naive Bayes* é um classificador estatístico baseado no teorema de *Bayes*. Ele calcula a probabilidade de um exemplo pertencer a uma determinada classe [Han et al., 2011]. O classificador assume que os preditores são variáveis independentes [Osman et al., 2013], o que é considerado uma suposição irrealista que pode afetar sua precisão [Caruana & Niculescu-Mizil, 2006]. Contudo, estudos mostram que mesmo nos casos em que as variáveis possuem forte dependência entre si, o algoritmo é efetivo e tem bons resultados [Zhang, 2004]. Ainda segundo Zhang [2004], o desempenho é influenciado pela distribuição de dependência.

2.5.2 Redes Neurais

A área de redes neurais se originou da tentativa de psicólogos e neurocientistas de desenvolver neurônios computacionais. Segundo Han et al. [2011], uma rede neural é um conjunto de unidades de entradas e saídas conectadas, na qual cada conexão tem um peso associado. Durante a fase de aprendizado, a rede aprende ajustando os pesos até conseguir predizer a classe correta do exemplo de entrada.

Existem diversos algoritmos de redes neurais e um dos mais populares é o *Back-propagation*. Nesse algoritmo, a rede é uma estrutura fixa correspondente a um grafo direcionado e o aprendizado se baseia em escolher pesos para cada aresta do grafo [Mitchell, 1997].

As redes neurais são tolerantes a ruído e conseguem classificar padrões não treinados, podendo ser usadas em situações nas quais se tem pouco conhecimento das relações entre os atributos e classes. Os tempos de treinamento são longos, logo, são mais apropriadas para aplicações nas quais esse tempo é factível. Outra desvantagem é que os algoritmos são difíceis de serem interpretados por humanos [Han et al., 2011].

2.5.3 *k-Nearest-Neighbor*

Os classificadores *k-Nearest-Neighbor* (vizinhos mais próximos) são baseados no aprendizado por analogia, isto é, os exemplos de teste são comparados com exemplos de treinamento similares. Cada exemplo é representado por um ponto em um espaço n-dimensional, para n preditores convertidos em uma representação numérica caso não sejam de outro tipo, como valores booleanos ou cores. Para classificar um exemplo de teste, o algoritmo busca os k exemplos de treinamento mais próximos por métrica de distância (euclidiana, por exemplo), e atribui a classe mais comum entre os k vizinhos. Quando a predição é numérica, ou seja, a classe é um valor numérico, a média dos valores das classes dos k vizinhos é retornada.

2.5.4 Máquina de Vetores de Suporte

Nas máquinas de vetores de suporte (*Support Vector Machines*) os exemplos (vetores de entrada) são transformados em pontos num espaço de características (preditores) de dimensão alta o suficiente para que sejam separados por uma superfície. Assim, com um mapeamento não linear, ele separa os dados do treinamento em duas classes, de forma que a distância entre os dois grupos seja a maior possível. Na fase de teste, os novos exemplos são divididos em cada espaço e a classe é definida de acordo com o lado que eles ocupam [Han et al., 2011; Cortes & Vapnik, 1995].

2.5.5 Algoritmos Genéticos

Algoritmos Genéticos, criados originalmente por Friedberg [1958], são baseados na ideia de evolução natural. Na natureza, organismos que não se adaptam bem a um ambiente morrem e os sobreviventes reproduzem gerando descendentes que carregam características similares às de seus antepassados [Russell & Norvig, 1995].

Com uma ideia semelhante, no algoritmo, os preditores e classes do conjunto de treinamento são transformados em sequências de bits chamadas regras, e estas regras são usadas como os indivíduos da população. Suponha, por exemplo, um conjunto de treinamento com dois preditores booleanos $P1$ e $P2$, e duas classes $C1$ e $C2$. A regra "se $P1$ e não $P2$ então $C2$ " pode ser codificada como "100", na qual os dois primeiros bits representam os preditores e o último bit representa a classe. Similarmente, a regra "se não $P1$ e não $P2$ então $C1$" tem código "001". Usando regras aleatórias desse tipo, cria-se uma população inicial e a partir dela formam-se novas gerações, escolhendo os elementos "mais aptos". A noção de adaptação geralmente é definida pela precisão num conjunto de exemplos treinados. Os descendentes são gerados aplicando

operações genéticas como *crossover* e mutações aleatórias. O processo de gerar novas populações continua até encontrar uma população em que cada regra satisfaça um limite predefinido [Han et al., 2011].

2.6 Considerações Finais

Este capítulo apresentou conceitos sobre funções utilitárias, refatoração e sistemas de recomendação, assim como os trabalhos relacionados. Apresentou-se uma visão geral da área de Aprendizado de Máquina bem como conceitos de avaliação e desenvolvimento necessários para o entendimento dos métodos utilizados neste trabalho. Diversos algoritmos de aprendizado de máquina foram apresentados, com destaque para o *Random Forests*, utilizado neste trabalho para reconhecer funções utilitárias.

Capítulo 3

Dataset

Neste capítulo é apresentado o conjunto de dados usado nesta dissertação de mestrado. A Seção 3.1 descreve o critério adotado para definir módulos utilitários no *dataset* proposto. Ele é formado por sistemas de código aberto Java e JavaScript, detalhados na Seção 3.2, e sistemas proprietários de uma empresa de desenvolvimento de software, também em Java e JavaScript, descritos na Seção 3.3. Um estudo exploratório realizado para verificar hipóteses que viabilizam a pesquisa é apresentado na Subseção 3.3.1.

3.1 Critério para Definição de Módulo Utilitário

Idealmente, funções utilitárias devem ser implementadas em módulos específicos — bibliotecas utilitárias — de forma a facilitar o reuso. Para automatizar a identificação desses módulos, assumiu-se que as bibliotecas utilitárias são todos os arquivos que contêm a palavra *"util"* em seu nome ou diretório, como, por exemplo, `src/util/ByteConverter.java`, `src/ng-services/UtilDOM.js` e `src/util/ArquivoUtils.java`. Os demais módulos dos sistemas foram considerados de propósito específico, ou seja, não utilitários. O *dataset* foi filtrado levando esse critério em consideração, já que os sistemas devem possuir essas bibliotecas, viabilizando os experimentos. Os detalhes desses dados e da filtragem são mostrados a seguir.

3.2 Sistemas de Código Aberto

Parte do *dataset* usado na pesquisa é formado por 106 sistemas de código aberto em Java e JavaScript. Para Java, inicialmente, foram considerados 111 sistemas do *Qualitas.class Corpus* [Terra et al., 2013]. Para JavaScript, foram usados 100 projetos

populares do *GitHub*[1], selecionados por ordem decrescente de estrelas. No *GitHub*, as estrelas são uma medida de popularidade entre os repositórios disponíveis na plataforma. Quanto mais estrelas, melhor avaliado pelos desenvolvedores é o projeto. Os repositórios foram baixados em Agosto de 2015.

Os sistemas foram filtrados pelo fato de possuir ou não módulos utilitários. De todos os sistemas, foram utilizados apenas aqueles que contêm, no mínimo, 25 funções em bibliotecas utilitárias — segundo o critério de possuir a palavra *"util"* em seu diretório. Assim, evitam-se sistemas muito pequenos e com poucas funções utilitárias, possibilitando a verificação da acurácia na identificação das mesmas.

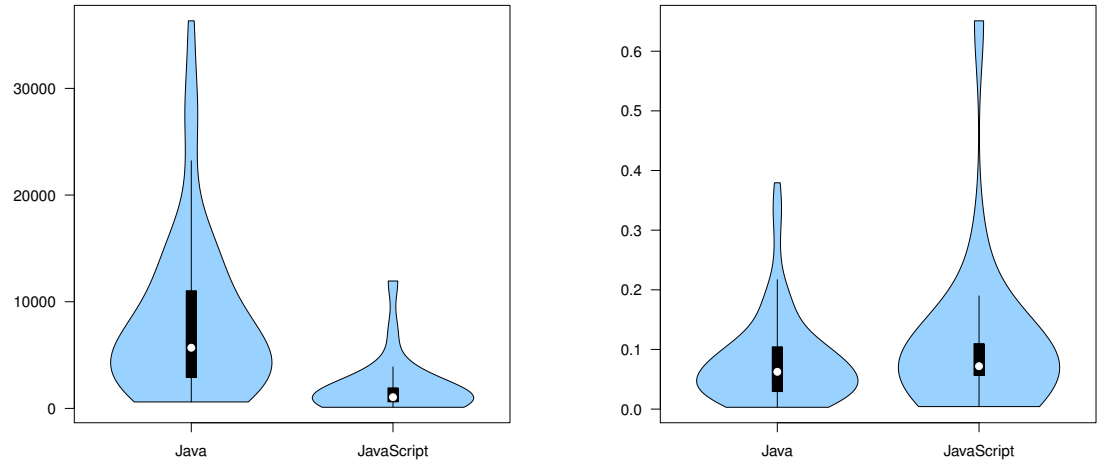
Foram encontradas bibliotecas utilitárias em 95 dos 111 sistemas em Java. Desses, foram selecionados 84 que possuíam pelo menos 25 métodos localizados em bibliotecas utilitárias. Métodos de classes de teste e interfaces foram ignorados. Para JavaScript, foram encontradas bibliotecas utilitárias em 42 sistemas, dos quais 22 continham no mínimo 25 funções utilitárias. Pelo fato de JavaScript ser uma linguagem interpretada, bibliotecas de terceiros e arquivos do próprio projeto não possuem diferenciação formal. Foi usada, então, a ferramenta *Linguist*[2], que implementa uma heurística para remover as bibliotecas de terceiros dos projetos. Essa ferramenta já foi usada com o mesmo propósito em outros trabalhos [Avelino et al., 2016]. A lista completa dos sistemas usados nesta pesquisa encontra-se no Apêndice A.

A Figura 3.1 mostra a distribuição do número de funções e da porcentagem de funções utilitárias nos 106 sistemas selecionados em Java e JavaScript. Os sistemas em Java têm mais funções do que os sistemas em JavaScript (medianas 5687 e 1049, respectivamente). No entanto, em termos relativos, os sistemas em JavaScript possuem mais funções utilitárias (mediana 6.25% para Java e 7.19% para JavaScript).

Em Java, o maior sistema em linhas de código e número de métodos é o *Azureus* com 36k métodos e 518.5k linhas de código. O menor sistema é o *jsXe* com 606 métodos e 9k linhas de código. O sistema com maior proporção de métodos utilitários é o *SandMark*, no qual 38% dos métodos, um total de 2.4k, são utilitários. *JChempaint* é o sistema com menos métodos utilitários — apenas 33, que representam 0.3% do total de métodos.

Em JavaScript, o sistema com maior número de funções é o *Node Inspector* com 12k funções e 119k linhas de código. O sistema com mais linhas de código, igual a 229k, é o *Ace*, que possui 7k funções, e também possui menor taxa de funções utilitárias — 0.4%, totalizando 30 funções. O menor sistema é o *Material-UI*, com 1k linhas de

[1] `http://github.com`
[2] `http://github.com/github/linguist`



(a) Número de Funções (b) % de Funções Utilitárias

Figura 3.1: Distribuição de (a) número de funções, (b) porcentagem de funções utilitárias.

código e 106 funções. Dessas, 69 são utilitárias, sendo esse o sistema com maior taxa dessas funções, de 65%.

3.3 Sistemas Proprietários

O conjunto de dados da pesquisa também conta com quatro sistemas da indústria, descritos na Tabela 3.1. Eles tiveram seus nomes reais omitidos por questões de confidencialidade.

Tabela 3.1: Sistemas proprietários utilizados neste trabalho.

Nome	**Descrição**	**Linguagem**
Sistema A	Visualização e localização de foguetes	JavaScript
Sistema B	Análise de dados de localização de foguetes	Java
Sistema C	Gestão de ativos	Java
Sistema D	Controle de infraestrutura elétrica	JavaScript

O sistema B, em Java, é o maior deles, com 60k linhas de código e o sistema D, em JavaScript é o menor com 9k linhas de código. O sistema D tem 26% das funções em bibliotecas utilitárias, sendo essa a maior proporção. O sistema que possui menos funções em módulos utilitários é o sistema C, em Java, representando apenas 1% das funções. Os quatro sistemas foram analisados manualmente e os detalhes dessa análise são discutidos na Seção 3.3.1.

3.3.1 Estudo Exploratório

Para construir a solução proposta nessa dissertação de mestrado, duas suposições foram verificadas:

- *Suposição 1*: Existem funções utilitárias que *não* são implementadas em bibliotecas utilitárias.
- *Suposição 2*: A maioria das funções implementadas em bibliotecas utilitárias são, de fato, utilitárias.

A primeira suposição se baseia na ideia de que o problema de pesquisa ocorre na prática, ou seja, que os desenvolvedores algumas vezes implementam funções utilitárias em módulos não utilitários. A segunda suposição pretende mostrar a viabilidade de resolver o problema de pesquisa treinando um classificador com as funções implementadas em bibliotecas utilitárias e testando sua habilidade em identificar funções similares implementadas em outros módulos. Para verificar essas suposições, um estudo exploratório foi realizado, analisando manualmente funções dos quatro sistemas proprietários do *dataset*. Os sistemas foram analisados pela própria autora desta dissertação de mestrado — desenvolvedora que trabalhou nos quatro projetos, com experiência de mais de três anos nos sistemas A e B, um ano no sistema C e seis meses no sistema D; ela trabalha há mais de cinco anos com Java e mais de três anos com JavaScript. Durante a análise, foram classificadas como utilitárias as funções que seguiam os seguintes critérios:

(a) não possuem regras de negócio;

(b) podem ser facilmente removidas do sistema e reusadas em outro;

(c) não dependem de outras classes ou funções definidas em outros arquivos do sistema, exceto quando não interferem diretamente no funcionamento da função, como, por exemplo, funções de *log* ou de lançamento de exceção;

(d) podem usar bibliotecas externas ou nativas da linguagem.

Construtores, interfaces, *getters*, *setters* e demais funções que não se encaixam nos critérios acima foram classificadas como não utilitárias. Alguns exemplos dos códigos analisados são mostrados a seguir.

```
1 /* systemA/src/utilmath.js */
2 function toInt(val) {
3     var number = parseInt(val);
4     if(isNaN(number)){
5         return 0;
6     }
7     return number;
8 }
```

Código 3.1: Exemplo de função utilitária no sistema A.

```
1 /* systemB/src/util/StagesTrajectoriesGenerator.java */
2 public List<Coordinate> trajectoryStageCalculator(long stageEvent,
3     List<Coordinate> coordinates) {
4     this.trajectory = coordinates;
5     this.cadenceCalculator();
6     this.vxAndVyCalculator();
7     int eventPosition = (int) ((stageEvent / 1000.0) / this.cadence);
8     return this.stageTrajectory(eventPosition);
9 }
```

Código 3.2: Exemplo de método não utilitário no sistema B.

```
1 /* systemC/src/relatorios/RelatorioFalhas.java */
2 public Integer diferencaEntreDatasEmDias(Date d1, Date d2) {
3     long newDate;
4     if (d1.after(d2)) {
5         newDate = d1.getTime() - d2.getTime();
6     } else {
7         newDate = d2.getTime() - d1.getTime();
8     }
9     return (int) (newDate / 1000 / 60 / 60 / 24);
10 }
```

Código 3.3: Exemplo de método utilitário no sistema C.

O Código 3.1 mostra uma função utilitária em JavaScript, que converte uma variável para inteiro, definida em uma biblioteca utilitária, isto é, possui a palavra "*util*" em seu nome (*utilmath.js*). O Código 3.2 exibe um método definido em uma biblioteca utilitária, mas que é de propósito específico — calcula a trajetória de um estágio de foguete. Logo, para os fins desta pesquisa, esse método pode ser considerado como um falso positivo. Já o Código 3.3 exibe um método utilitário, definido fora de bibliotecas utilitárias, que retorna a quantidade de dias entre duas datas. Assim, o objetivo central dessa dissertação de mestrado é identificar métodos como esse.

Os resultados da análise são sumarizados na Tabela 3.2. Para cada sistema, mostra-se o número de linhas de código (LOC); o número total de funções (NF); o número de funções implementadas em bibliotecas utilitárias (NFU); o número de falsos positivos (FP), ou seja, funções implementadas em bibliotecas utilitárias que não são

funções utilitárias; e o número de falsos negativos (FN), isto é, funções utilitárias não implementadas em bibliotecas utilitárias.

Tabela 3.2: Resultados do estudo exploratório.

Sistema	**LOC**	**NF**	**NFU**	**FP**	**FN**
Sistema A	12,212	1,334	199	11	16
Sistema B	60,184	6,905	388	17	14
Sistema C	38,015	7,371	70	16	46
Sistema D	8,827	298	78	25	2

Observa-se que, para esses sistemas, as duas suposições enunciadas no início desta seção são confirmadas. Respectivamente para os sistemas A, B, C e D, temos 16, 14, 46 e 2 funções utilitárias que *não* foram implementadas em bibliotecas utilitárias (coluna FN). Esses valores correspondem entre 0.2% e 1.4% das funções em módulos não utilitários, confirmando assim a suposição 1. Os dados também mostram que a maioria das funções em bibliotecas utilitárias é utilitária de fato, já que, respectivamente nos sistemas A, B, C e D, 94%, 95%, 77% e 67% das funções em módulos utilitários são realmente utilitárias (colunas NFU e FP). Apenas o sistema D apresentou muitos falsos positivos. Ainda assim, considerou-se verdadeira a suposição 2, pois o resultado obtido foi maior que 50%.

Ameaças à Validade dos Resultados

A classificação manual das funções utilitárias está sujeita a falhas, pois é uma análise parcialmente subjetiva, apesar de ter sido feita por uma especialista nos sistemas e nas linguagens. O risco dessas falhas pode ser diminuído incluindo a opinião de mais especialistas. Além disso, a definição de módulo utilitário adotada não é precisa. Desta forma, o número de falsos positivos e falsos negativos reportados no estudo pode ser afetado por essa definição.

3.4 Considerações Finais

Este capítulo apresentou o conjunto de sistemas usado nesta pesquisa. Ele é formado por 84 sistemas Java e 22 sistemas em JavaScript de código aberto, dois sistemas proprietários em Java e dois sistemas proprietários em JavaScript. Discutiu-se também o critério adotado para definir módulos utilitários. Por fim, um estudo exploratório mostrou que o problema de pesquisa desta dissertação ocorre na prática, ou seja, exis-

tem funções utilitárias implementadas fora de módulos utilitários. Conclui-se ainda com esse estudo que a maioria das funções definidas em módulos utilitários é, de fato, utilitária.

Capítulo 4

Identificação de Funções Utilitárias Usando Aprendizado de Máquina

Neste capítulo é detalhado um estudo para identificação de funções utilitárias via aprendizado de máquina. O projeto desse estudo é apresentado na Seção 4.1. Os preditores usados na classificação são descritos na Seção 4.2. As avaliações realizadas são discutidas nas Seções 4.3 e 4.4, detalhando as fases de treinamento e teste no *dataset*. Por fim, a Seção 4.6 apresenta as considerações finais.

4.1 Projeto do Estudo

Na abordagem de aprendizado de máquina, um classificador foi treinado e testado com conjuntos de métricas de código de funções utilitárias e não utilitárias. O algoritmo escolhido foi o *Random Forests*, já que ele é largamente usado em outros trabalhos na área de Engenharia de Software [Lessmann et al., 2008; Peters et al., 2013; Costa et al., 2014; Dias et al., 2015] e possui diversas vantagens discutidas no Capítulo 2. Utilizou-se a implementação disponível no pacote *randomForest* [Liaw & Wiener, 2002] da ferramenta R, configurado com 500 árvores, que é o número padrão do pacote. Como mencionado no Capítulo 3, o critério para definir módulos utilitários foi identificar a palavra *"util"* em seu nome ou diretório. Com esse critério, foi possível determinar automaticamente a classe de cada exemplo de entrada para o classificador — valor booleano definindo se a função é utilitária ou não. Os preditores usados na classificação, bem como detalhes do treinamento e teste são mostrados a seguir.

4.2 Preditores

Os preditores usados como entrada para o algoritmo foram métricas de código estáticas calculadas em nível de método para Java e em nível de função para JavaScript. Foram usados 23 preditores para os sistemas em Java, e 20 preditores para os sistemas em JavaScript. As ferramentas adotadas para extrair as métricas foram o *parser* do *Eclipse JDT*[1] para Java e o *Esprima*[2] para JavaScript. As listas de preditores são mostradas nas Tabelas 4.1 e 4.2 para Java e JavaScript, respectivamente.

Tabela 4.1: Lista de preditores para os sistemas em Java — métricas por método.

Preditor	**Descrição**	**Tipo**
loc	Número de linhas de código	int
complexity	Complexidade ciclomática	int
parameters	№ de parâmetros formais	int
return	Retorna um tipo diferente de `void`	boolean
isStatic	É estático	boolean
modifier	Modificador (`private`/`public`/`protected`/`default`)	string
isFinal	Usa a palavra-chave `final`	boolean
isAbstract	É abstrato	boolean
this	№ de referências ao `this`	int
isGetSet	É um método `get`/`set`	boolean
subclass	É implementado em uma subclasse	boolean
overwrite	Sobrescreve método de super classe	boolean
exception	Lança uma exceção	boolean
staticVar	№ de referências a variáveis estáticas	int
finalVar	№ de referências a variáveis `final`	int
fields	№ de referências a atributos da classe	int
systemMethods	№ de chamadas a métodos do sistema	int
bibMethods	№ de chamadas a métodos de bibliotecas externas e da API do Java	int
systemClasses	№ de referências a classes do sistema	int
bibClasses	№ de referências a API do Java e a classes externas	int
methodCallers	№ de chamadas a métodos externos à classe	int
classCallers	№ de classes que chamam o método	int
innerClass	É implementado numa classe aninhada	boolean

Os preditores listados se baseiam em características intuitivas de funções utilitárias. Por exemplo, normalmente uma função utilitária não usa a palavra-chave `this`, não referencia variáveis globais e não depende de outras funções que não sejam nativas da linguagem. Além dessas características, os preditores foram selecionados para

[1] `http://www.eclipse.org/jdt`
[2] `http://esprima.org`

Tabela 4.2: Lista de preditores para os sistemas em JavaScript — métricas por função.

Preditor	**Descrição**	**Tipo**
loc	Número de linhas de código	int
complexity	Complexidade ciclomática	int
parameters	№ de parâmetros	int
return	Inclui a declaração de `return`	boolean
prototype	Usa `prototype`	boolean
arguments	Usa a variável `arguments`	boolean
nested	É aninhada a uma função não anônima	boolean
empty	Tem corpo vazio	boolean
returnBoolean	Apenas retorna uma constante booleana	boolean
this	№ de referências ao `this`	int
dom	№ de usos do DOM	int
getById	№ de chamadas à função `getElementById` do DOM usando uma *string* constante como argumento	int
globalVar	№ de referências a variáveis globais	int
nativeVar	№ de referências a variáveis nativas (ex.: Math)	int
funcCall	№ de chamadas a funções	int
localFunc	№ de definições de funções aninhadas (locais)	int
localCall	№ de chamadas a funções locais	int
propCall	№ de chamadas a funções via propriedade de variáveis (ex.: obj.func())	int
nativeCall	№ de chamadas a funções nativas (ex.: Math.sin())	int
globalCall	№ de chamadas a funções não nativas, não locais e que não sejam via propriedade	int

incluir métricas básicas como número de linhas de código e complexidade ciclomática; dados fornecidos diretamente pela AST (*Abstract Syntax Tree*), como número de parâmetros e uso do modificador `static`; e métricas de acoplamento, já que, idealmente, funções utilitárias não devem depender de funções do sistema, mas podem depender de bibliotecas externas. Além disso, espera-se que o classificador identifique automaticamente os preditores mais relevantes. Logo, incluiu-se também preditores que não possuem relevância intuitiva para diferenciar funções utilitárias das demais.

4.3 Avaliação *Intra-Project*

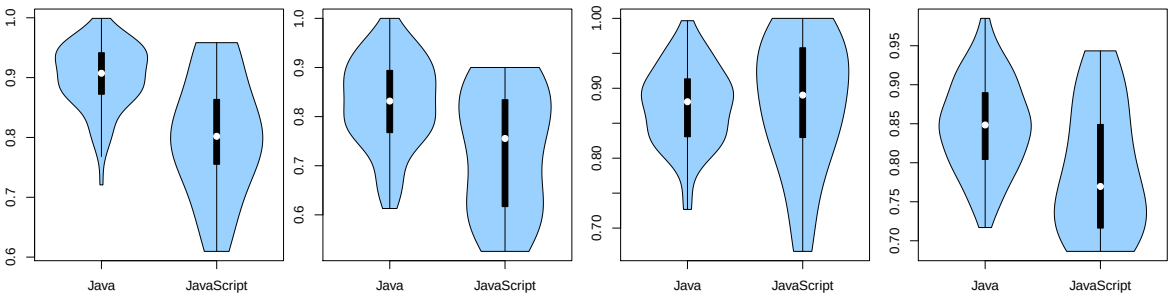
Para cada sistema do *dataset*, os preditores (definidos na Seção 4.2) foram computados para dois conjuntos A e B. O conjunto A possui todas as funções implementadas em bibliotecas utilitárias. O conjunto B é formado pelas funções definidas em bibliotecas não utilitárias selecionadas randomicamente, para ter o mesmo número de funções do

conjunto A. Desta forma, a entrada para o algoritmo é balanceada, possuindo o mesmo número de funções utilitárias e não utilitárias, e conta com pelo menos 50 exemplos. É importante ressaltar que normalmente existe ruído em ambos os conjuntos. De um lado, no conjunto A, podem existir funções que não são utilitárias; por outro lado, no conjunto B, podem existir funções utilitárias. Porém, como mostrado no estudo exploratório (Seção 3.3.1 do Capítulo 3), ambos os casos normalmente não dominam os conjuntos. Além disso, o classificador *Random Forests* é robusto a ruído nos dados [Breiman, 2001].

Usando esses dados, cada sistema foi treinado e testado separadamente com o classificador *Random Forests* por meio de uma *10-fold cross validation*. Os exemplos do sistema foram randomicamente divididos em dez grupos e foram executados dez ciclos de treinamento e teste. Em cada ciclo, nove grupos foram usados para treinamento e o grupo restante para testes, de forma que, ao final, todos os exemplos fossem usados no treinamento e no teste. Depois dos dez ciclos, os resultados de acurácia foram agregados em quatro medições: AUC, precisão, *recall* e *F-measure* (média harmônica ponderada do *recall* e da precisão).

4.3.1 Avaliação nos Sistemas de Código Aberto

Para cada um dos 106 sistemas de código aberto do *dataset* um classificador foi treinado e testado usando *10-fold cross validation*. A Figura 4.1 mostra a distribuição dos valores das medidas de acurácia para cada sistema, por linguagem. Os resultados dessa figura são sumarizados a seguir.



(a) AUC (b) Precisão (c) *Recall* (d) *F-measure*

Figura 4.1: Distribuição das medidas de acurácia.

AUC: Para Java, os resultados de AUC para o primeiro, segundo e terceiro quartis foram 0.87, 0.90 e 0.94, respectivamente. *Apache Collections* foi o sistema com maior medida de AUC (0.99) e *Freemind* foi o sistema com menor medida (0.72). Para

JavaScript, os resultados de AUC para o primeiro, segundo e terceiro quartis foram 0.75, 0.80 e 0.86, respectivamente. *Typeahead* foi o sistema com maior medida de AUC (0.95) e *Mocha* foi o sistema com menor medida (0.60).

Precisão: Para Java, os resultados de precisão para o primeiro, segundo e terceiro quartis foram 0.76, 0.83 e 0.89, respectivamente. *Checkstyle* foi o sistema com maior medida de precisão (1.0) e *Freemind* foi o sistema com menor medida (0.61). Para JavaScript, os resultados de precisão para o primeiro, segundo e terceiro quartis foram 0.61, 0.75 e 0.83, respectivamente. *React Native* foi o sistema com maior medida de precisão (0.9) e *ReactJS* foi o sistema com menor medida (0.52).

Recall: Para Java, os resultados de *recall* para o primeiro, segundo e terceiro quartis foram 0.83, 0.88 e 0.91, respectivamente. *Apache Collections* foi o sistema com maior medida de *recall* (0.99) e *Azureus* foi o sistema com menor medida (0.72). Para JavaScript, os resultados de *recall* para o primeiro, segundo e terceiro quartis foram 0.82, 0.89 e 0.95, respectivamente. *React* e *Typeahead* foram os sistemas com maior medida de *recall* (1.0) e *React Native* foi o sistema com menor medida (0.66).

F-measure: Para Java, os resultados de *f-measure* para o primeiro, segundo e terceiro quartis foram 0.80, 0.84 e 0.88, respectivamente. *Apache Collections* foi o sistema com maior medida de *f-measure* (0.98) e *Freemind* foi o sistema com menor medida (0.71). Para JavaScript, os resultados de *f-measure* para o primeiro, segundo e terceiro quartis foram 0.71, 0.76 e 0.84, respectivamente. *Typeahead* foi o sistema com maior medida de *f-measure* (0.94) e *Material* foi o sistema com menor medida (0.68).

Resumo: esses resultados mostram que o classificador *Random Forests* obteve alta precisão ao identificar funções utilitárias usando os preditores selecionados e uma entrada balanceada por classe. Contudo, os resultados para os sistemas em JavaScript são piores quando comparados aos dos sistemas em Java (*f-measure* mediano igual a 0.84 e 0.76 para Java e JavaScript, respectivamente). Esse resultado é intuitivamente esperado, pois JavaScript é uma linguagem dinâmica e fracamente tipada.

4.3.1.1 Análise dos Preditores

Com o objetivo de entender melhor os resultados obtidos na classificação, algumas análises foram feitas com os preditores usados no trabalho. Nesta seção, serão discutidos alguns pontos como a ordem de importância, a distribuição de acordo com os sistemas e as diferenças entre os melhores e piores preditores.

A classificação dos preditores é feita de acordo com sua importância. Ela é uma medida obtida pela função *importance* do pacote *randomForest* [Liaw & Wiener, 2002] em R, que usa o índice *Gini* para calculá-la. O índice de *Gini* é uma medida de impureza dos nós da árvore de decisão [Han et al., 2011]. Ela é calculada de acordo com a quantidade de exemplos positivos e negativos do nó. Quanto mais exemplos de diferentes classes ele possui, maior a impureza. Ela é igual a zero quando todos os exemplos do nó pertencem a uma mesma classe [Mitchell, 1997].

Para cada um dos dez *folds* na classificação é gerado um *ranking* de preditores. O *ranking* em cada sistema é gerado calculando-se a média das posições de cada preditor nos dez *folds*. O teste *Scott-Knott* [Scott & Knott, 1974] foi aplicado em tais *rankings*, agrupando os preditores pela importância. Ele identifica grupos de variáveis estatisticamente diferentes umas das outras. Foi usada a implementação disponível no pacote *ScottKnott*[3] da ferramenta R. O resultado do teste é mostrado nas Tabelas 4.3 e 4.4.

Tabela 4.3: *Ranking* de preditores para Java organizados pelo teste *Scott-Knott*.

Grupo	**Preditor**	**Posição média**	**Melhor posição**	**Pior posição**
G1	loc	1.9	1	6
G2	fields	5.34	1	12
	systemClasses	6.35	2	13
G3	isStatic	7	1	18
	systemMethods	7.2	2	14
	subclass	7.27	1	20
	parameters	7.36	2	15
	bibClasses	7.67	2	16
G4	bibMethods	8.83	1	17
	methodCallers	9.46	2	15
G5	complexity	10.8	3	17
	classCallers	10.96	1	16
	modifier	11.33	2	20
G6	finalVar	12.54	1	23
G7	isGetSet	14.6	2	20
	return	15.11	4	20
G8	staticVar	16.44	3	21
	exception	16.75	2	23
	this	17.04	2	22
G9	innerClass	19.14	5	23
	isFinal	19.92	8	23
G10	overwrite	21.02	4	23
	isAbstract	21.85	16	23

[3] `http://cran.r-project.org/web/packages/ScottKnott/ScottKnott.pdf`

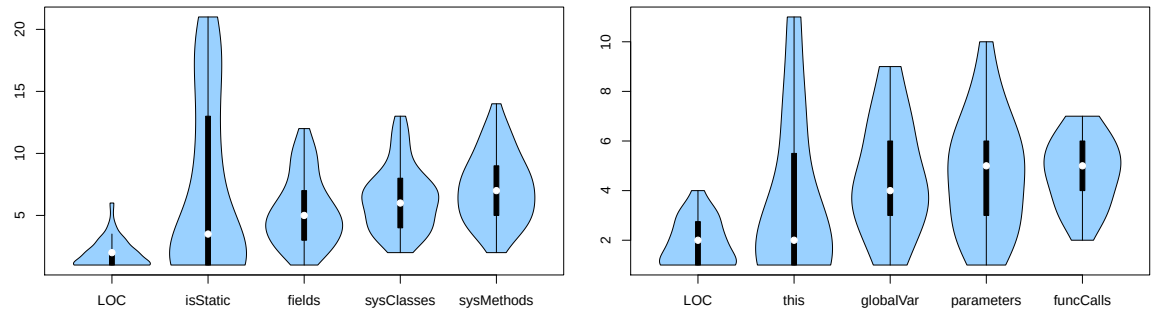
Tabela 4.4: *Ranking* de preditores para JavaScript organizados pelo teste *Scott-Knott*.

Grupo	Preditor	Posição média	Melhor posição	Pior posição
G1	loc	1.86	1	3
G2	this	3.54	1	12
G3	globalVar	4.54	1	8
	parameters	4.68	1	9
	funcCall	4.72	2	7
	propCall	5.54	2	7
	complexity	6.18	3	10
G4	nativeVar	8.95	8	13
	return	9.81	3	20
	globalCall	10.09	5	15
G5	localFunc	11.18	7	15
	nested	11.31	2	16
	prototype	12.59	2	18
G6	nativeCall	13.95	11	17
	arguments	14.77	11	18
G7	dom	15.27	10	19
	localCall	16.0	11	20
	empty	16.77	12	20
G8	returnBoolean	18.59	15	20
	getById	19.59	18	20

As Tabelas 4.3 e 4.4 mostram, respectivamente para os sistemas Java e JavaScript, os *rankings* dos preditores de acordo com sua importância na classificação. Eles estão ordenados pela posição média nos sistemas e separados em grupos com diferenças estatisticamente significativas. São mostradas também a melhor e pior posições obtidas. Observa-se que, para ambas as linguagens, o melhor preditor foi o LOC. Na sequência, para Java, o número de referências a atributos da classe (*fields*) e o número de referências a classes do próprio sistema (*systemClasses*) foram os melhores; e para JavaScript, os melhores preditores foram o número de referências à palavra-chave `this` (*this*) e às variáveis globais (*globalVar*). Para os sistemas em JavaScript, foi possível notar que, provavelmente, os preditores dos grupos G6, G7 e G8 não são importantes, considerando que os mesmos não ocupam uma das nove primeiras posições em nenhum sistema.

A Figura 4.2 mostra a distribuição de posições no *ranking* dos cinco melhores preditores para todos os sistemas no *dataset*, de acordo com a mediana. As posições variam consideravelmente dependendo do sistema. Por exemplo, a posição do preditor *isStatic* varia de 1 (para 29 sistemas do *dataset*) até 21 (para 3 sistemas). LOC é

o preditor com menor dispersão para ambas as linguagens. Esses resultados sugerem que os classificadores para funções utilitárias devem ser gerados para cada sistema e, provavelmente, uma abordagem *cross-project*, no qual o treinamento reúne vários sistemas, deve apresentar menor acurácia. Nota-se também que os sistemas em JavaScript apresentaram menor dispersão. Isso aconteceu, possivelmente, pelo fato de o número de sistemas no *dataset* ser pequeno (22 sistemas).



(a) Java (b) JavaScript

Figura 4.2: Distribuição de *ranking* dos cinco melhores preditores.

A seguir, são detalhados os valores de preditores de alguns sistemas para analisar aqueles com maior e menor importância e entender melhor as características das funções utilitárias que levaram aos resultados obtidos. Foram analisados os três sistemas com maior AUC e os três com menor AUC, para cada linguagem. Além disso, em cada sistema, mostram-se os três melhores preditores, de acordo com o índice *Gini*. O teste U de *Mann-Whitney* [Mann & Whitney, 1947] foi aplicado em cada preditor para calcular o *p-value*. O objetivo é mostrar se existe ou não uma diferença estatística entre os valores para as funções utilitárias e não utilitárias que justifique o desempenho do classificador. Um *p-value* abaixo de 0.05 significa que os dados são estatisticamente diferentes. Nas Tabelas 4.5, 4.6, 4.7 e 4.8 tais valores foram destacados em negrito.

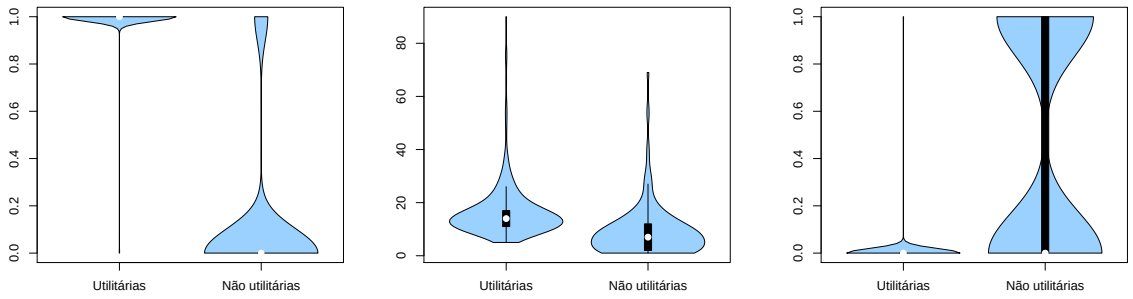
Análise para Java

A Tabela 4.5 mostra os *p-values* para os três sistemas em Java que obtiveram maior AUC. Nota-se que todos os preditores possuem uma diferença estatística significativa para funções utilitárias e não utilitárias, o que justifica a boa capacidade do classificador em identificá-las.

Tabela 4.5: Indicador de diferença estatística (*p-value*) entre métodos utilitários e não utilitários para os melhores preditores dos sistemas em Java com maior AUC.

Sistema	**AUC**	**Preditor**	***p-value***
Collections	0.99	isStatic	**<0.01**
		loc	**<0.01**
		subclass	**<0.01**
Jext	0.97	isStatic	**<0.01**
		fields	**<0.01**
		exception	**<0.01**
Cobertura	0.97	isStatic	**<0.01**
		bibClasses	**<0.01**
		bibMethods	**<0.01**

A Figura 4.3 mostra a distribuição dos melhores preditores para o *Apache Collections*, que obteve melhor AUC. Confirmando os resultados estatísticos, os valores são visivelmente diferentes para métodos utilitários e não utilitários. Para esse sistema, praticamente todos os métodos utilitários são estáticos; para a maioria, o número de linhas de código é um pouco maior que nos demais métodos; e poucos são implementados em subclasses. Nos gráficos, os preditores booleanos foram definidos como zero ou um (*isStatic* e *subclass*).



(a) isStatic (p-value<0.01) (b) loc (p-value<0.01) (c) subclass (p-value<0.01)

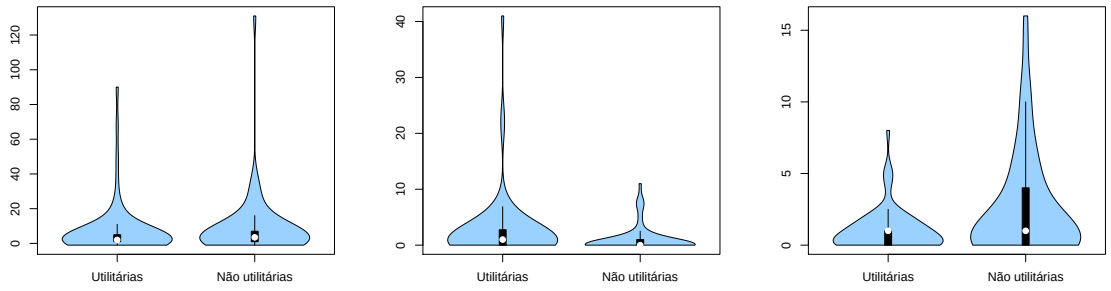
Figura 4.3: Distribuição dos melhores preditores do sistema *Apache Collections* (AUC=0.99).

A Tabela 4.6 mostra os *p-values* para os sistemas em Java com menor AUC. Apenas um preditor de cada um dos três sistemas (*bibMethods* e *bibClasses*) possuem uma diferença estatística significativa para funções utilitárias e não utilitárias, logo, fica mais difícil classificar as funções com esses preditores, resultando em um AUC máximo de 0.78.

Tabela 4.6: Indicador de diferença estatística (*p-value*) entre métodos utilitários e não utilitários para os melhores preditores dos sistemas em Java com menor AUC.

Sistema	**AUC**	**Preditor**	***p-value***
Freemind	0.72	loc	0.54
		bibMethods	**<0.01**
		systemMethods	0.07
JRefactory	0.77	loc	0.55
		bibMethods	**0.02**
		systemMethods	0.06
JGraphpad	0.78	loc	0.59
		bibClasses	0.58
		subclass	**<0.01**

A Figura 4.4 mostra a distribuição dos melhores preditores para o *Freemind*, que obteve o pior AUC. Graficamente, nota-se pouca diferença no principal preditor (LOC) para funções utilitárias e não utilitárias. Já o número de referências a métodos de bibliotecas (*bibMethods*) possui uma diferença significativa. Se o modelo do classificador tivesse considerado *bibMethods* como principal preditor, possivelmente, o resultado seria melhor.



(a) loc (p-value=0.54)

(b) bibMethods (p-value<0.01)

(c) systemMethods (p-value=0.07)

Figura 4.4: Distribuição dos melhores preditores do sistema *Freemind* (AUC=0.72).

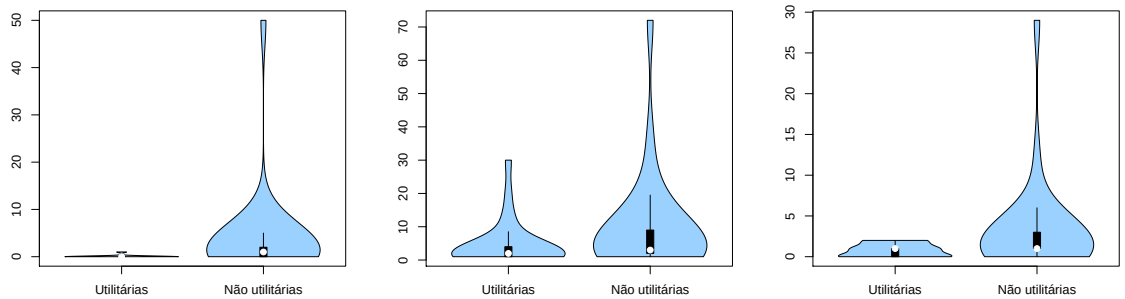
Análise para JavaScript

A Tabela 4.7 mostra os *p-values* dos preditores para os sistemas em JavaScript com maior AUC. Somente um preditor (*parameters*) não possui diferença estatística significativa para funções utilitárias e não utilitárias, resultando em bons valores de AUC, todos acima de 0.90.

Tabela 4.7: Indicador de diferença estatística (*p-value*) entre funções utilitárias e não utilitárias para os melhores preditores dos sistemas em JavaScript com maior AUC.

Sistema	**AUC**	**Preditor**	***p-value***
Typeahead	0.95	this	**<0.01**
		loc	**0.01**
		propCall	**0.04**
GitBook	0.94	loc	**<0.01**
		this	**<0.01**
		parameters	0.34
ECharts	0.94	this	**<0.01**
		loc	**<0.01**
		globalCall	**<0.01**

A Figura 4.5 mostra a distribuição dos melhores preditores para o *Typeahead*, que obteve melhor AUC (0.95). É possível observar graficamente a diferença entre os valores para as funções utilitárias e não utilitárias, verificando os dados estatísticos. Para esse sistema, a grande maioria das funções utilitárias não usa a palavra-chave `this`, possui o número linhas de código menor que as demais e faz poucas chamadas a funções via propriedades de objetos (*propCall* mais baixo).



(a) this (p-value<0.01) (b) loc (p-value=0.01) (c) propCall (p-value=0.04)

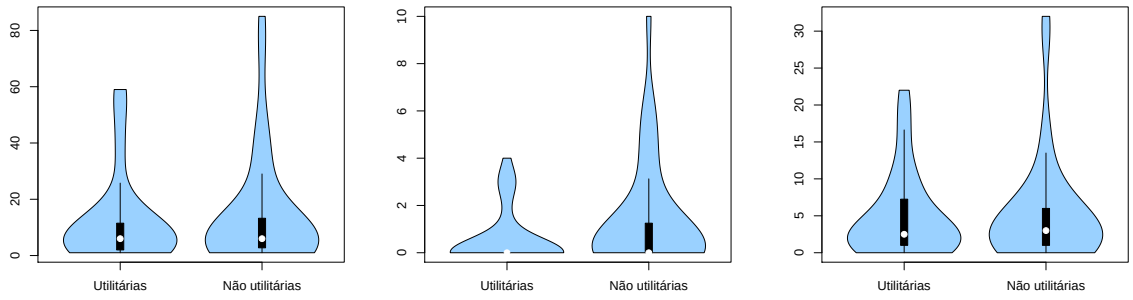
Figura 4.5: Distribuição dos melhores preditores do sistema *Typeahead* (AUC=0.95).

A Tabela 4.8 mostra os *p-values* para os preditores nos sistemas em JavaScript com menor AUC. Apenas um preditor (*nested*) no sistema *Ionic* possui diferença estatística significativa entre funções utilitárias e não utilitárias. Inclusive, tal sistema obteve o melhor AUC dentre os três analisados. Logo, o classificador gerou piores resultados — AUC abaixo de 0.70 nos três sistemas.

Tabela 4.8: Indicador de diferença estatística (*p-value*) entre funções utilitárias e não utilitárias para os melhores preditores dos sistemas em JavaScript com menor AUC.

Sistema	**AUC**	**Preditor**	***p-value***
Mocha	0.60	loc	0.90
		globalVar	0.05
		funcCall	0.92
React	0.64	loc	0.46
		globalVar	0.88
		parameters	0.28
Ionic	0.67	nested	**<0.01**
		loc	0.39
		globalVar	0.90

A Figura 4.6 mostra a distribuição dos melhores preditores para o *Mocha*, sistema que obteve menor AUC. Observa-se que para os preditores que representam o número de linhas de código e de chamadas a funções, os dados são visivelmente parecidos, confirmando os altos *p-values*.



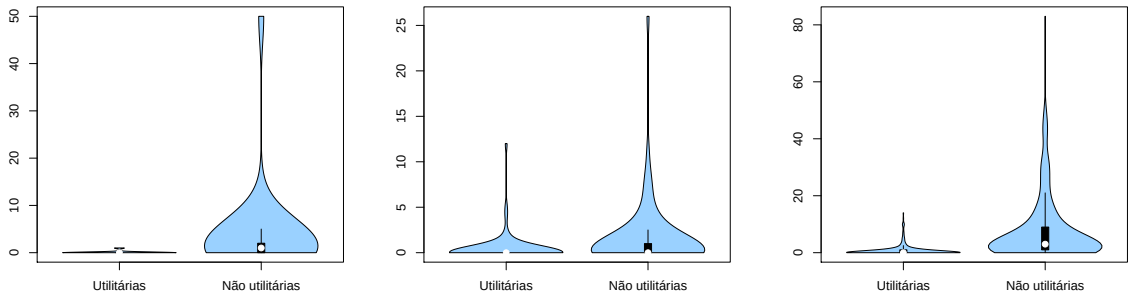
(a) loc (p-value=0.90) (b) globalVar (p-value=0.05) (c) funcCall (p-value=0.92)

Figura 4.6: Distribuição dos melhores preditores do sistema *Mocha* (AUC=0.60).

Discussão dos Resultados

A seguir, algumas observações são feitas a partir dos resultados apresentados anteriormente. A propriedade *static* dos métodos foi predominante na classificação dos sistemas em Java com melhores resultados. Os métodos utilitários geralmente são estáticos e a maioria dos demais não, o que faz dessa propriedade um preditor importante. Porém, é preciso lembrar que o objetivo da pesquisa é encontrar funções utilitárias fora de bibliotecas utilitárias, sendo assim, possivelmente o desenvolvedor que não define um método utilitário no devido módulo também pode não declará-lo

estático. Assim, se o classificador considera esse preditor determinante para rotular os métodos, provavelmente não identificará os métodos utilitários definidos em módulos inapropriados. Para JavaScript, a palavra-chave `this` também obteve destaque, isto porque, a maioria das funções utilitárias não a usa. Isso aconteceu para os três sistemas com maior AUC, como mostra a Figura 4.7.



(a) Typeahead (b) GitBook (c) ECharts

Figura 4.7: Distribuição do preditor *this* nos sistemas com maior AUC em JavaScript.

Observou-se também que para alguns sistemas Java com AUC extremos (Cobertura, Freemind e JRefactory), os métodos utilitários têm um número de chamadas a métodos de bibliotecas externas (*bibMethods*) maior que os demais. Isso pode estar relacionado ao costume de encapsular funcionalidades dessas bibliotecas em métodos utilitários. Notou-se também, que o número de referências a propriedades da classe (*fields*) no *Jext* é consideravelmente menor para funções utilitárias. Essa é uma característica intuitivamente esperada nessas funções.

Resumo: os principais preditores observados nesses experimentos foram LOC, *isStatic* (Java) e *this* (JavaScript). Eles representam as características mais relevantes para a identificação das funções utilitárias no conjunto de dados analisado usando aprendizado de máquina. Apesar disso, as características das funções utilitárias normalmente mudam de um sistema para outro. Por exemplo, nos sistemas *ECharts* e *Typeahead*, o LOC é menor nas funções utilitárias, já no *GitBook* e *Apache Collections*, o LOC é maior. Logo, não se pode concluir que é possível identificar funções utilitárias usando preditores universais, porque cada sistema possui um perfil próprio.

4.3.2 Avaliação nos Sistemas Proprietários

Um classificador foi gerado para os quatro sistemas proprietários da mesma forma descrita na Seção 4.3. Os resultados da *10-fold-cross validation*, usando como entrada

o mesmo número de dados de funções utilitárias e não utilitárias, são mostrados na Tabela 4.9.

Tabela 4.9: Resultados do *Random Forests* nos sistemas proprietários.

Sistema	**Precisão**	***Recall***	**F-measure**	**AUC**
Sistema A	0.86	0.93	0.90	0.94
Sistema B	0.90	0.67	0.76	0.90
Sistema C	0.72	0.42	0.52	0.75
Sistema D	0.89	0.91	0.90	0.96

A maior precisão foi de 0.90 para o sistema B e a menor de 0.72 para o sistema C. O maior *recall* foi de 0.93 para o sistema A e o menor foi de 0.42 para o sistema C. Os sistemas A e D tiveram o maior *f-measure*, no valor de 0.90, e o sistema C obteve o pior resultado, no valor de 0.52. O melhor AUC foi de 0.96 para o sistema D e o pior foi de 0.75 para o sistema C.

Resumo: esses resultados são parecidos com aqueles obtidos nos sistemas de código aberto e mostram uma alta acurácia para identificar funções utilitárias usando dados balanceados (precisão média de 84% e *recall* médio de 73%).

4.3.2.1 Aplicação Prática

Para avaliar a aplicação prática do classificador é preciso verificar se o mesmo consegue identificar funções utilitárias fora dos módulos utilitários. De acordo com os dados obtidos no estudo exploratório mostrados na Seção 3.3.1 do Capítulo 3, nos quatro projetos existem funções utilitárias fora de bibliotecas utilitárias. Essas funções são candidatas a serem movidas para bibliotecas utilitárias. Nesta avaliação, o classificador foi treinado com as funções dos módulos utilitários e com o mesmo número de funções escolhidas aleatoriamente dos módulos não utilitários. Ressalta-se que o treinamento foi realizado com ruído, já que existem funções não utilitárias nos módulos utilitários e vice-versa, simulando um ambiente mais próximo do real. Após o treinamento, o classificador foi aplicado às funções dos módulos não utilitários que não foram escolhidas para treinamento e verifica-se se as funções utilitárias identificadas manualmente, como mostrado no estudo exploratório (Seção 3.3.1 do Capítulo 3), são identificadas (ver Figura 4.8). Como as funções de módulos não utilitários são escolhidas aleatoriamente, eventualmente algumas das funções utilitárias que pretende-se identificar podem ser selecionadas no conjunto de teste como ruído.

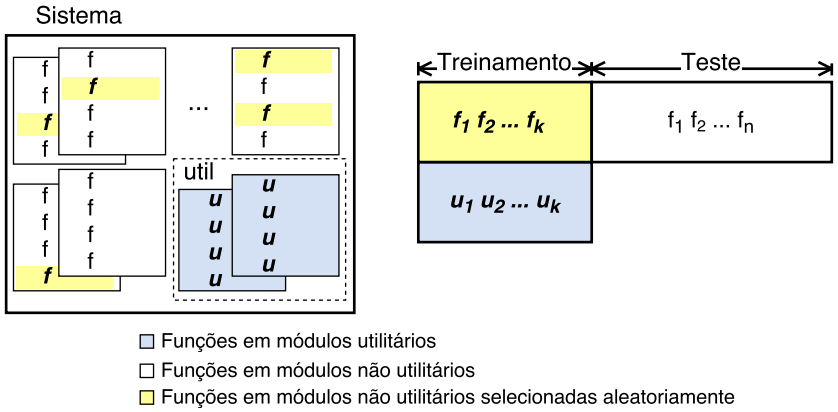


Figura 4.8: Modelo de entrada de dados de treinamento e de teste para o *Random Forests* em cada sistema.

Ao aplicar o classificador ao conjunto de teste, o mesmo retorna a probabilidade de cada função ser utilitária ou não. Logo, aplica-se um limiar para classificar as funções. Os resultados são exibidos em três matrizes de confusão, uma para cada um dos limiares de 60%, 70% e 80%. Ou seja, se a probabilidade de a função ser utilitária, retornada pelo classificador, for maior que o limiar, ela é considerada utilitária. As Tabelas 4.10, 4.11, 4.12 e 4.13 mostram as matrizes de confusão, a precisão e o *recall*, para os quatro sistemas.

Predição		Classe
util	não-util	real
0	13	util
141	782	não-util

(a) Prob. > 60%
prec=0 rec=0

Predição		Classe
util	não-util	real
0	13	util
110	813	não-util

(b) Prob. > 70%
prec=0 rec=0

Predição		Classe
util	não-util	real
0	13	util
82	841	não-util

(c) Prob. > 80%
prec=0 rec=0

Tabela 4.10: Matrizes de confusão do Sistema A (JavaScript).

Predição		Classe
util	não-util	real
8	6	util
264	5851	não-util

(a) Prob. > 60%
prec=0.03 rec=0.57

Predição		Classe
util	não-util	real
4	10	util
105	6010	não-util

(b) Prob. > 70%
prec=0.04 rec=0.28

Predição		Classe
util	não-util	real
1	13	util
38	6077	não-util

(c) Prob. > 80%
prec=0.02 rec=0.07

Tabela 4.11: Matrizes de confusão do Sistema B (Java).

A precisão e o *recall* nos sistemas A e D foram iguais a zero para os três limites de probabilidades. Para o sistema B a precisão também é muito baixa, alcançando o valor

Predição		Classe
util	não-util	real
28	18	util
325	6860	não-util

(a) Prob. > 60%
prec=0.08 rec=0.61

Predição		Classe
util	não-util	real
24	22	util
79	7106	não-util

(b) Prob. > 70%
prec=0.30 rec=0.52

Predição		Classe
util	não-util	real
19	27	util
36	7149	não-util

(c) Prob. > 80%
prec=0.52 rec=0.41

Tabela 4.12: Matrizes de confusão do Sistema C (Java).

Predição		Classe
util	não-util	real
0	1	util
5	136	não-util

(a) Prob. > 60%
prec=0 rec=0

Predição		Classe
util	não-util	real
0	1	util
3	138	não-util

(b) Prob. > 70%
prec=0 rec=0

Predição		Classe
util	não-util	real
0	1	util
1	140	não-util

(c) Prob. > 80%
prec=0 rec=0

Tabela 4.13: Matrizes de confusão do Sistema D (JavaScript).

máximo de 0.04 para probabilidade de ser utilitária superior a 70%. O *recall* diminuiu à medida que o limiar da probabilidade foi aumentado, tendo o valor máximo igual a 0.57 e o menor igual a 0.07. Os melhores resultados foram observados no sistema C, no qual, para a probabilidade acima de 80% de ser utilitária, obteve-se precisão igual a 0.52 e *recall* igual a 0.41. O melhor *recall* foi de 0.61 com a probabilidade mínima de 60%, mas com precisão de 0.08.

Resumo: apesar do classificador treinado e testado em um conjunto de dados balanceados usando *10-fold-cross validation* apresentar um bom desempenho, ele não manteve esses resultados para identificar funções utilitárias fora dos módulos utilitários, em uma abordagem mais próxima de um cenário de aplicação prática. Obteve-se uma precisão média de 8.5% e *recall* médio de 20%, usando uma probabilidade maior que 70%.

4.4 Validação *Cross-Project*

Uma vez que o objetivo desta pesquisa é identificar funções utilitárias definidas em módulos inapropriados, uma das formas de realizar tal identificação, via aprendizado de máquina, é treinar o classificador com alguns sistemas e testar no sistema que se pretende identificar as funções utilitárias. Para avaliar essa abordagem, foi feita uma validação *cross-project* nos sistemas proprietários e de código aberto, conforme descrito nas seções a seguir.

4.4.1 Avaliação nos Sistemas de Código Aberto

Na validação *cross-project* dos sistemas de código aberto, todas as funções de todos os sistemas foram adicionadas à entrada do *Random Forest* e foi executada a *10-fold-cross validation*. Desta forma, o classificador é treinado e testado com funções de vários projetos ao mesmo tempo.

Para os 84 sistemas em Java, a precisão foi de 0.39, o *recall* foi de 0.34, o *f-measure* foi de 0.36 e o AUC foi de 0.74. Para os 22 sistemas em JavaScript, a precisão foi de 0.23, o *recall* foi de 0.25, o *f-measure* foi de 0.24 e o valor do AUC de 0.70.

Para Java, o preditor mais significativo foi LOC. Na sequência, os melhores preditores foram o método ser estático ou não, o número de acessos a campos da classe, a complexidade ciclomática e o número de parâmetros. Para os sistemas em JavaScript, o preditor mais significativo também foi LOC. Na sequência das primeiras posições, mas com índice *Gini* bem menor, os melhores preditores foram o número de chamadas de função, o número de chamadas de função via propriedade, o número de referências a variáveis globais e a complexidade ciclomática.

A Figura 4.9 ilustra a diferença dos resultados entre a abordagem por projeto, com entrada balanceada e a validação *cross-project*. Apesar dos valores de AUC serem próximos, a precisão, *recall* e *f-measure* são consideravelmente inferiores na abordagem *cross-project*. Uma possível razão para tal diferença é que as características das funções utilitárias podem mudar de acordo com o sistema. Logo, o treinamento realizado com base nessas propriedades deixa de ser efetivo para classificar funções de outro sistema.

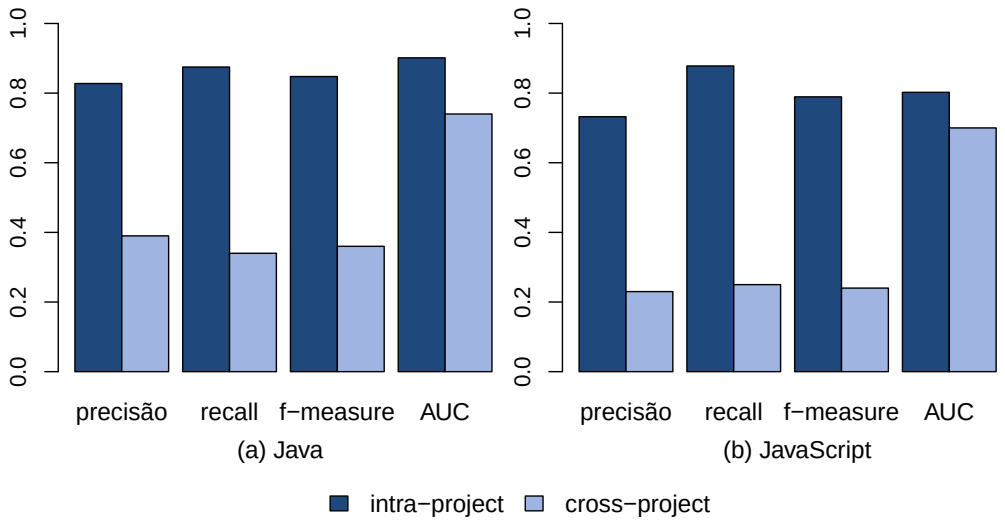


Figura 4.9: Resultados da validação *cross-project* comparados às avaliações por projeto.

Resumo: os resultados mostram que o algoritmo não foi capaz de generalizar a identificação de funções nos sistemas de código aberto usando uma abordagem *cross-project*. Obteve-se precisão de 39% para Java e 23% para JavaScript, valores muito inferiores aos resultados da classificação por projeto. Acredita-se que o principal motivo é que os preditores mais significativos do modelo variam de acordo com o projeto.

4.4.2 Avaliação nos Sistemas Proprietários

Nos sistemas proprietários, o classificador foi gerado treinando em um sistema e testando em outro, simulando a aplicação real do mesmo. Para Java, os resultados do sistema B foram obtidos treinando o classificador com o sistema C e vice-versa. De forma análoga, os resultados do sistema A foram obtidos treinando o classificador com o sistema D e vice-versa, ambos em JavaScript. As fases de treinamento e teste foram baseadas na classificação manual das funções feita por um especialista, como mostrado no Capítulo 3. A Tabela 4.14 mostra os valores de verdadeiro positivo (VP), falso positivo (FP), verdadeiro negativo (VN), falso negativo (FN), precisão e *recall* para os quatro sistemas. O sistema A obteve precisão de 3% e *recall* de 0.4%, com apenas um verdadeiro positivo. Os demais sistemas tiveram precisão e *recall* nulos.

Tabela 4.14: Resultados da validação *cross-project* nos sistemas proprietários.

Sistema	**VP**	**FP**	**VN**	**FN**	**Precisão**	***Recall***
Sistema A	1	29	1101	203	0.03	0.004
Sistema B	0	5	6,515	385	0	0
Sistema C	0	5	7,266	100	0	0
Sistema D	0	6	237	55	0	0

Resumo: esses resultados mostram que o *Random Forests* apresentou desempenho ruim na validação *cross-project* nos sistemas proprietários. Obteve-se uma precisão média de 0.75% e recall médio de 0.1%, indicando que o algoritmo não foi capaz de generalizar a identificação, o que dificulta sua utilização em um cenário de aplicação prática.

4.5 Ameaças à Validade dos Resultados

Assumiu-se que as bibliotecas utilitárias têm a palavra *"util"* em seu diretório. No entanto, existem bibliotecas utilitárias nos sistemas que não seguem esse padrão adotado. Logo, elas serão apontadas como falsos positivos quando forem classificadas. Alguns exemplos dessas bibliotecas foram

`aspectj/runtime/internal/Conversions.java` do sistema *AspectJ* em Java, que possui métodos de conversões de tipo; `aoi/artofillusion/math/FastMath.java` do sistema *AOI* que faz cálculos matemáticos simples como arredondamentos, potenciação, etc; e `/brackets/src/LiveDevelopment/Agents/DOMHelpers.js` do sistema *Brackets* em JavaScript que possui funções genéricas para manipular o DOM. Além disso, usando esse critério, o treinamento do algoritmo é realizado com ruído. Acredita-se que considerar um critério mais preciso para selecionar as bibliotecas utilitárias, pode melhorar os resultados.

Em JavaScript, geralmente a importação de bibliotecas de terceiros se dá pela inclusão de um arquivo JavaScript contendo todo o código da biblioteca unificado. A remoção dessas bibliotecas dos projetos JavaScript de código aberto foi feita através de uma heurística implementada na ferramenta *Linguist*. No entanto, não é garantido que esses arquivos sejam completamente removidos. O problema disso é que as funções não estão dividas em módulos, logo as funções utilitárias estarão junto com todas as outras funções, mesmo que, originalmente, estivessem em módulos utilitários. Isso pode prejudicar o resultado das avaliações que se baseiam na suposição de que a maioria das funções utilitárias estão em arquivos contendo a palavra "*util*" em seu nome ou diretório.

4.6 Considerações Finais

Este capítulo apresentou uma proposta de identificação de funções utilitárias via aprendizado de máquina. Os resultados mostraram que essa abordagem não é viável para identificar funções utilitárias fora de bibliotecas utilitárias. Apesar de obter resultados muito bons na identificação de funções nos módulos utilitários de cada sistema, com treinamento balanceado, esses resultados não se mantêm em uma validação *cross-project*, a qual é mais próxima de um cenário de aplicação prática. O *Random Forests* seleciona características para diferenciar as funções que são específicas de cada projeto, perde a capacidade de generalizar e não identifica funções fora das bibliotecas utilitárias. No próximo capítulo, será discutido um conjunto de heurísticas para identificação de funções utilitárias.

Capítulo 5

Heurísticas para Identificação de Funções Utilitárias

Neste capítulo, propõe-se um conjunto de heurísticas para identificar funções utilitárias em Java e JavaScript. Na Seção 5.1, são apresentadas em detalhes as heurísticas para encontrar funções utilitárias. As avaliações nos sistemas proprietários são mostradas na Seção 5.2. Um estudo sobre a configuração de um sistema de recomendação usando aprendizado de máquina é mostrado na Subseção 5.2.1. A Seção 5.3 descreve um *survey* realizado para avaliar a identificação de funções utilitárias nos sistemas de código aberto. As ameaças à validade são discutidas na Seção 5.4, e por fim, são apresentadas as conclusões na Seção 5.5.

5.1 Heurísticas Propostas

As heurísticas propostas para identificar funções utilitárias são baseadas em propriedades comuns a essas funções, que podem ser computadas por meio de análise estática. Um conjunto de características intuitivamente inerentes a funções utilitárias foi levantado e aperfeiçoado observando os erros e acertos ao investigar tais características em alguns sistemas do *dataset*. Para classificar uma função como utilitária, deve-se verificar se a função possui todas essas características.

As heurísticas propostas para identificar funções utilitárias são baseadas na ideia de que elas tendem a ter um baixo acoplamento com as demais funções. Normalmente, elas apenas processam e retornam algum dado. Elas têm propósito genérico e não implementam regras de negócio. Também não são implementadas usando determinados padrões e mecanismos como herança, por exemplo.

As heurísticas foram implementadas usando o analisador proposto para extrair os preditores usados na abordagem de aprendizado de máquina (Capítulo 4). Usou-se o *parser* do *Eclipse JDT*[1] para Java e o *Esprima*[2] para JavaScript, para fazer a análise estática do código. Depois de coletadas, as métricas são usadas em estruturas condicionais que definem se a função é utilitária ou não, implementadas em um protótipo de ferramenta para identificação de funções utilitárias. Essas regras são listadas a seguir, assim como alguns exemplos de código extraídos do *dataset* (Capítulo 3).

Um método utilitário em Java *não* deve possuir as seguintes características:

(a) possuir corpo vazio;

(b) ser um método `get/set` usado apenas para recuperar e alterar atributos de uma classe, identificados por meio da seguinte heurística: seu nome se inicia com *"set"* e recebe um parâmetro, ou seu nome se inicia com *"get"*, não possui parâmetros e não possui tipo de retorno;

(c) possuir retorno do tipo `void`, pois espera-se que uma função utilitária retorne algo;

(d) apenas retornar o literal `null`, ou um literal booleano, numérico, *string*, ou um único caractere (um exemplo é mostrado no Código 5.1);

(e) apenas lançar uma exceção (um exemplo é mostrado no Código 5.2);

(f) ser abstrato, isto é, não possuir implementação;

(g) sobrescrever um método, pois é um mecanismo comumente utilizado em entidades, que, por sua vez, representam regras de negócio;

(h) ser implementado em uma subclasse, pois é uma estrutura normalmente utilizada para relacionar entidades, que, por sua vez, representam regras de negócio;

(i) fazer referência à palavra-chave `this`, já que esta é usada normalmente para acessar atributos e métodos da própria classe e, idealmente, uma função utilitária não deve usar elementos externos;

(j) fazer referência a campos da classe, indicando que o método está acoplado à mesma;

[1] `http://www.eclipse.org/jdt`
[2] `http://esprima.org`

(k) lançar uma exceção, pois espera-se que um método utilitário encapsule o serviço oferecido, inclusive tratando os erros que podem ocorrer, evitando assim um acoplamento desnecessário entre o método e seus clientes, já que esse tipo de exceção (verificada) deve ser tratado em tempo de compilação[3];

(l) possuir modificador diferente de `public`, pois, um método utilitário deve ser acessado por qualquer módulo;

(m) instanciar classes do sistema, que é um indicativo de dependência do sistema e das regras de negócio do mesmo;

(n) fazer chamadas a métodos do sistema, que é um indicativo de dependência do sistema e das regras de negócio do mesmo.

```
1 /* apache/tools/ant/ProjectHelper.java */
2 protected Object initialValue(){
3   return ".";
4 }
```

Código 5.1: Exemplo de método não utilitário do sistema *Apache Collections* que apenas retorna um literal do tipo *string*.

```
1 /* argouml/kernel/MemberList.java */
2 public boolean containsAll(Collection<?> arg0){
3   throw new UnsupportedOperationException();
4 }
```

Código 5.2: Exemplo de método não utilitário do sistema *ArgoUML* que apenas lança uma exceção.

Uma função utilitária em JavaScript *não* deve possuir as seguintes características:

(a) ser anônima, pois ela pode ser chamada em outros pontos do código (um exemplo é mostrado no Código 5.3);

(b) ser aninhada a funções não anônimas, pois uma função local tende a realizar uma tarefa específica para aquele contexto (um exemplo é mostrado no Código 5.4);

(c) ser uma função `get/set`, ou seja, seu nome se inicia com *"get"* ou *"set"*;

(d) possuir o corpo vazio (um exemplo é mostrado no Código 5.5);

(e) apenas retornar o literal `null` ou um literal booleano (conforme exemplo mostrado no Código 5.6);

[3]Apesar dessa característica sobrescrever a definida em (e), quando desconsiderada em uma possível configuração da ferramenta, produz resultados diferentes.

(f) fazer referência à palavra-chave `this`, pois subentende-se a existência de um construtor para o objeto e, geralmente, uma função utilitária não precisa de construtor;

(g) acessar elementos específicos do DOM (*Document Object Model*) usando o comando `document.getElementById('idDefinido')` ou uma seleção *jQuery*[4] do tipo `$('#idDefinido')`, passando como parâmetro uma *string* literal e não uma variável (um exemplo é mostrado no Código 5.7);

(h) usar variáveis globais não nativas da linguagem, que é um indicativo de dependência do sistema e das regras de negócio do mesmo;

(i) fazer chamadas a funções, exceto nos seguintes casos: as funções são nativas da linguagem (por exemplo, `Math.sin()` e `parseInt()`); ou são locais (mostra-se um exemplo no Código 5.4); ou são chamadas via propriedade (conforme exemplo no Código 5.8); ou são utilitárias, ou seja, obedecem a todos os critérios acima.

```
1 /* ace/lib/ace/ace.js */
2 editor.on("destroy", function() {
3     event.removeListener(window, "resize", env.onResize);
4     env.editor.container.env = null;
5 });
```

Código 5.3: Exemplo de função não utilitária e anônima do sistema *Ace* a qual é passada como parâmetro na chamada da função `on`.

```
1  /* brackets/src/editor/EditorManager.js */
2  function _handleRemoveFromPaneView(e, removedFiles) {
3      var handleFileRemoved = function (file) {
4          var doc = DocumentManager.getOpenDocumentForPath(file.fullPath);
5          if (doc) {
6              MainViewManager._destroyEditorIfNotNeeded(doc);
7          }
8      };
9      if ($.isArray(removedFiles)) {
10         removedFiles.forEach(function (removedFile) {
11             handleFileRemoved(removedFile);
12         });
13     } else {
14         handleFileRemoved(removedFiles);
15     }
16 }
```

Código 5.4: Exemplo de função não utilitária pois é aninhada e chamada localmente no sistema *Brackets*. A função local `handleFileRemoved` (linha 3) encontra-se aninhada na função `_handleRemoveFromPaneView` (linha 2) e é chamada localmente em dois pontos (linhas 11 e 14).

[4] `http://jquery.com`

```
1 /* mocha/lib/context.js */
2 function Context(){}
```

Código 5.5: Exemplo de função não utilitária com corpo vazio no sistema *Mocha*.

```
1 /* pdf/src/core/colorspace.js */
2 isPassthrough: function ColorSpace_isPassthrough(bits) {
3     return false;
4 }
```

Código 5.6: Exemplo de função não utilitária no sistema *PDF* que apenas retorna um literal booleano.

```
1 /* ace/lib/ace/ext/settings_menu.js */
2 function showSettingsMenu(editor) {
3     var sm = document.getElementById("ace_settingsmenu");
4     if (!sm)
5     overlayPage(editor, generateSettingsMenu(editor), "0", "0", "0");
6 }
```

Código 5.7: Exemplo de função não utilitária do sistema *Ace* que acessa o DOM com um literal *string* passado via parâmetro (linha 3).

```
1 /* leaflet/src/layer/tile/GridLayer.js */
2 createTile: function () {
3     return document.createElement("div");
4 }
```

Código 5.8: Exemplo de função do sistema *Leaflet* que faz uma chamada de função via propriedade. O objeto é `document` e a função é `createElement`.

As restrições apresentadas foram definidas para evitar falsas recomendações. Mesmo que a solução proposta deixe de recomendar algumas funções, o objetivo é diminuir o número de falsos positivos. Apesar disso, alguns sistemas apresentarão melhores resultados se determinadas restrições forem desconsideradas. Isso ocorre porque as principais propriedades de funções utilitárias podem mudar significativamente de um projeto para outro. Por exemplo, cada sistema pode depender de diferentes bibliotecas; em alguns projetos Java, as funções utilitárias podem ser estáticas, mas em outros, não; sistemas em JavaScript podem seguir paradigmas de modularização diferentes, desde programação procedural até orientação a objeto baseada em classes [Silva et al., 2015]. Por essa razão, o ideal é que um recomendador de funções utilitárias seja configurável, permitindo desabilitar algumas regras para melhor se adaptar ao contexto de um determinado sistema.

5.2 Avaliação nos Sistemas Proprietários

A avaliação conduzida nos sistemas proprietários do *dataset* permite uma análise mais realista e resultados mais precisos, já que os mesmos foram analisados manualmente para classificar cada função. Como os sistemas são conhecidos, foi possível também escolher a melhor configuração para os mesmos. Um protótipo de ferramenta, que usa as heurísticas apresentadas, foi aplicado com a configuração padrão nos sistemas A e B. No sistema C ele foi configurado desabilitando a restrição de não lançar exceção (k), pois muitos métodos dos módulos utilitários desse sistema possuem tal comportamento. O sistema D, em JavaScript, possui a característica de definir os módulos dentro de funções; logo, a restrição que desconsidera funções aninhadas (b) impacta significativamente nos resultados. Por esse motivo, a restrição de não ser aninhada à função não anônima foi desabilitada na configuração desse sistema. Os resultados obtidos são mostrados nas Tabelas 5.1 e 5.2.

Tabela 5.1: Resultado da identificação de funções utilitárias em módulos não utilitários.

Sistema	VP	FP	VN	FN	Precisão	Recall
Sistema A (JavaScript)	14	31	1088	2	0.31	0.87
Sistema B (Java)	4	4	6499	10	0.50	0.28
Sistema C (Java)	5	6	7249	41	0.45	0.11
Sistema D (JavaScript)	1	24	194	1	0.04	0.50

A Tabela 5.1 mostra os resultados relativos à identificação de funções utilitárias fora dos módulos utilitários. Já a Tabela 5.2 mostra os resultados da identificação de todas as funções utilitárias do sistema, incluindo aquelas definidas em módulos utilitários. Reporta-se o número de verdadeiros positivos (VP), falsos positivos (FP), verdadeiros negativos (VN), falsos negativos (FN) e os valores de precisão e de *recall*.

Tabela 5.2: Resultado da identificação de funções utilitárias em todo o sistema.

Sistema	VP	FP	VN	FN	Precisão	Recall
Sistema A (JavaScript)	56	36	1094	148	0.60	0.27
Sistema B (Java)	64	4	6516	321	0.94	0.17
Sistema C (Java)	16	6	7265	84	0.72	0.16
Sistema D (JavaScript)	31	32	211	24	0.49	0.56

Na identificação de funções utilitárias em módulos não utilitários (Tabela 5.1), a melhor precisão foi de 0.50 no sistema B e a pior foi de 0.04 no sistema D. Já o melhor *recall* foi de 0.87 no sistema A e o mais baixo foi de 0.11 no sistema C. Na identificação

das funções incluindo os módulos utilitários (Tabela 5.2), a precisão é melhor e o *recall* é menor. A melhor precisão obtida foi de 0.94 no sistema B e a pior foi de 0.49 no sistema D. O melhor *recall* foi de 0.56 para o sistema D e o pior foi obtido no sistema C, no valor de 0.16. Portanto, o *recall* foi melhor nos sistemas em JavaScript.

Apesar da acurácia na identificação de funções utilitárias em módulos de propósito específico ser inferior àquela que inclui os módulos utilitários, ambos os resultados são superiores aos obtidos usando aprendizado de máquina. A Figura 5.1 apresenta uma comparação entre os resultados obtidos nos sistemas proprietários usando as heurísticas propostas e usando um classificador *Random Forests* (validação *cross-project*). Esses valores se referem à identificação de funções utilitárias em todos os módulos dos sistemas. Observa-se que as heurísticas propostas apresentam melhores resultados em todos os quatro sistemas analisados.

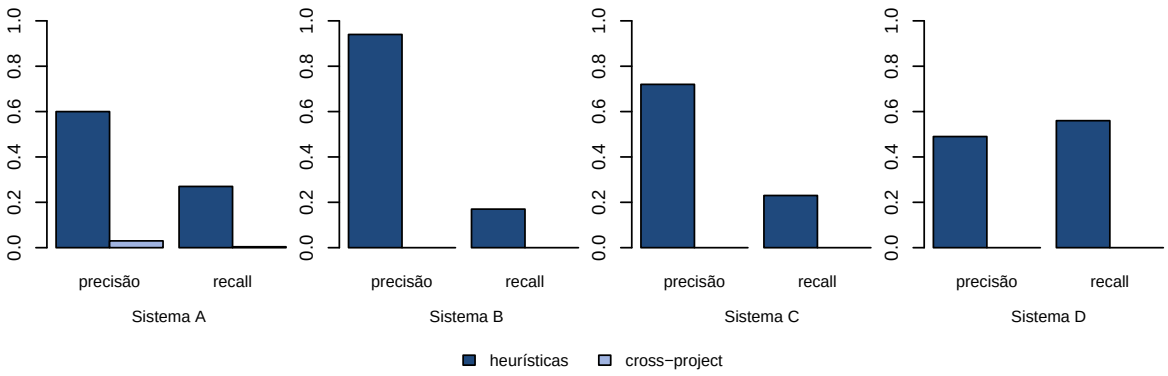


Figura 5.1: Resultados da avaliação das heurísticas comparados à validação *cross-project* nos sistemas proprietários.

Resumo: os resultados obtidos com as heurísticas propostas são superiores aos da abordagem de aprendizado de máquina discutidos no Capítulo 4 (Subseções 4.3.2.1 e 4.4.2). O *recall* é baixo, mas é melhor na identificação de funções fora dos módulos utilitários, e mostra que o conjunto de heurísticas é capaz de identificar funções utilitárias implementadas em módulos inadequados com uma precisão mínima de 49% e máxima de 94% (quando consideradas todas funções de um sistema).

5.2.1 Usando o *Random Forests* para Configurar um Recomendador de Funções Utilitárias

As características que distinguem funções utilitárias podem mudar de um projeto para outro. Desta forma, determinadas regras que um recomendador utiliza para definir uma função utilitária podem afetar o desempenho do mesmo, dependendo do sistema.

O ideal é que, antes de executar o recomendador, ele seja configurado para ter conformidade com o sistema. Um exemplo ocorre no sistema D do *dataset*. Nele, as funções são definidas dentro de outras funções que simulam módulos. Logo, a regra que desconsidera funções aninhadas a funções não anônimas restringe a identificação, pois a grande maioria das funções é aninhada.

Essa configuração deve ser feita pelo especialista do sistema, já que ele conhece bem as características do mesmo. No entanto, acredita-se que algumas configurações possam ser recomendadas usando o *ranking* de preditores gerado com o *Random Forests*. Para verificar essa suposição, foi feito um experimento no qual os cinco preditores menos relevantes de cada sistema proprietário foram usados para desabilitar as respectivas regras no protótipo do recomendador. Como esses preditores tiveram baixa relevância na classificação, isso pode significar que as características não são muito importantes na distinção entre funções utilitárias e não utilitárias. Como não são todos preditores que possuem uma heurística associada, somente os que são relacionados foram usados. A Tabela 5.3 compara os resultados desse experimento aos obtidos com a configuração padrão, ou seja, usando todas as heurísticas propostas nesse trabalho.

Tabela 5.3: Comparação dos resultados da identificação de funções utilitárias entre a configuração padrão e uma configuração personalizada usando o *Random Forests*.

Configuração	Padrão		Personalizada	
Sistema	Precisão	Recall	Precisão	Recall
Sistema A (JavaScript)	0.60	0.27	0.57	0.27
Sistema B (Java)	0.94	0.17	0.90	0.17
Sistema C (Java)	0.66	0.11	0.38	0.16
Sistema D (JavaScript)	0	0	0.41	0.46

Para o sistema A, os cinco piores preditores são *arguments*, *localCall*, *empty*, *returnBoolean* e *nativeCall*. Destes, somente *empty* (funções vazias) e *returnBoolean* (funções que só retornam literais booleanos) possui uma correspondência direta para uma heurística proposta e assim foram removidos. Com essa configuração, a precisão cai de 0.60 para 0.57 e o *recall* não se altera.

No sistema B, os cinco piores preditores são *isFinal*, *isAbstract*, *innerClass*, *exception* e *isGetSet*. Destes, somente *isAbstract* (método abstrato), *exception* (método lança exceção) e *isGetSet* (é um método `get`/`set`) são relacionados a regras, que foram removidas. A precisão cai de 0.94 para 0.90 e o *recall* não se altera.

Para o sistema C, os cinco piores preditores são *innerclass*, *isFinal*, *isAbstract*, *overwrite* e *modifier*. Destes, *isAbstract* (método abstrato), *overwrite* (sobrescreve um

método) e *modifier* (modificador) possuem regras correspondentes e foram removidas. A precisão cai de 0.66 para 0.38 e o *recall* sobe de 0.11 para 0.16.

Finalmente no sistema D, os cinco piores preditores são *arguments*, *prototype*, *returnBoolean*, *empty* e *nested*. Destes, *returnBoolean* (funções que só retornam literais booleanos), *empty* (funções vazias) e *nested* (funções aninhadas) são relacionados a regras, que foram removidas. A precisão sobe de 0 para 0.41 e o *recall* sobe de 0 para 0.46. Como discutido na Seção 5.2, removendo apenas a restrição de funções aninhadas os resultados são melhores (precisão de 0.49 e *recall* igual a 0.56).

Resumo: com esses resultados, conclui-se que a recomendação da configuração baseada nos piores preditores do *Random Forests* pode ser boa dependendo do sistema. Na maioria dos casos, a configuração padrão produz melhores resultados. No entanto, a configuração feita com base no conhecimento das características específicas do sistema, ou seja, feita por um especialista do sistema, ainda é a melhor abordagem.

5.3 Survey com Desenvolvedores

As funções encontradas por meio das heurísticas em módulos não utilitários são aquelas cuja movimentação deve ser sugerida para módulos utilitários. Para contabilizar o total de recomendações corretas, seria necessária uma análise manual para verificar se cada uma é ou não utilitária. Devido à falta de acesso a um especialista dos sistemas de código aberto considerados e ao grande número de sistemas e funções, essa análise não é trivial de ser realizada. Assim, para se obter uma estimativa da precisão das recomendações nos sistemas de código aberto do *dataset* considerado, foi realizado um *survey*. Nele, parte das funções identificadas em módulos não utilitários foi julgada por desenvolvedores de software como utilitárias ou não. A seguir, são mostrados alguns exemplos de funções identificadas pelas heurísticas propostas, usando o protótipo implementado.

```
1 /* ace/lib/ace/lib/lang.js */
2 exports.copyObject = function(obj) {
3     var copy = {};
4     for (var key in obj) {
5         copy[key] = obj[key];
6     }
7
8     return copy;
9 };
```

Código 5.9: Exemplo de função utilitária em JavaScript, do sistema *Ace*, implementado fora de módulo utilitário.

```
1 /* myfaces/org/apache/myfaces/view/facelets/tag/jstl/fn/JstlFunction.java*/
2 public static boolean containsIgnoreCase(String name, String searchString){
3     if (name == null || searchString == null) {
4         return false;
5     }
6     return -1 != name.toUpperCase().indexOf(searchString.toUpperCase());
7 }
```

Código 5.10: Exemplo de método utilitário em Java, do sistema *MyFaces*, implementado fora de módulo utilitário.

```
1 /* phaser/src/physics/arcade/World.js */
2 distanceBetween: function (source, target) {
3     var dx = source.x - target.x;
4     var dy = source.y - target.y;
5     return Math.sqrt(dx * dx + dy * dy);
6 }
```

Código 5.11: Exemplo de função utilitária em JavaScript, do sistema *Phaser*, implementada fora de módulo utilitário.

```
1 /* node-inspector/front-end/plataform/DOMExtension.js */
2 function createElement(tagName) {
3     return document.createElement(tagName);
4 }
```

Código 5.12: Exemplo de função utilitária em JavaScript, do sistema *Node Inspector*, implementada fora de módulo utilitário.

O Código 5.9 é uma função do *Ace* que faz a cópia de um objeto. O Código 5.10 mostra um método do *MyFaces* que verifica se uma *string* é encontrada em outra, sem diferenciar letras maiúsculas de minúsculas. O Código 5.11 apresenta uma função do *Phaser* que calcula a distância euclidiana entre dois pontos. Finalmente, o Código 5.12 mostra uma função do *Node Inspector* que encapsula a criação de um elemento do DOM. Todos esses exemplos são funções utilitárias encontradas fora de módulos utilitários por meio das heurísticas propostas.

O *survey* foi realizado por meio de formulários do *Google Forms* enviados por e-mail. Em cada formulário, foram inseridas dez funções para que o participante as julgasse como utilitárias ou não usando uma escala de zero a três, na qual zero significa que a função certamente não é utilitária, um significa que possivelmente não é utilitária, dois significa que possivelmente é utilitária e três significa que certamente é utilitária. O número par de respostas foi usado para forçar o participante a definir o tipo da função. Assim, respostas iguais a zero ou um, foram contabilizadas como funções não utilitárias, e respostas iguais a dois ou três foram consideradas como utilitárias. As escalas intermediárias foram usadas porque as funções não são avaliadas

pelos especialistas dos sistemas e há certa subjetividade ao determinar se uma função é utilitária. Cada pergunta foi acompanhada do código fonte da função, o nome do arquivo, a linha em a mesma foi implementada e um link para tal arquivo. Um exemplo dos formulários usados no *survey* é mostrado no Apêndice B.

O formulário também possui perguntas para que o participante informe seu nível de experiência em desenvolvimento de software e nas linguagens Java ou JavaScript. Em seu cabeçalho foi dada uma breve definição do que são funções utilitárias. Informou-se também o tempo estimado para respondê-lo (de 10 a 15 minutos). Ressaltou-se ainda, a importância de se consultar o arquivo no qual a função foi implementada para entender o contexto em que a mesma se encontra.

Das dez funções de cada questionário, duas são *pontos de controle* para validar a resposta do participante. Elas foram escolhidas manualmente do *dataset* e são claramente de domínio específico, ou seja, dependem de entidades relacionadas ao domínio do sistema e exibem regras de negócio específicas. O Código 5.13 mostra um exemplo desse tipo de método usado nos formulários. Se o participante classificar uma dessas duas funções como utilitária, sua resposta é descartada e o formulário é reenviado a outro desenvolvedor. As demais oito funções foram escolhidas aleatoriamente, sem repetição, dentre as funções identificadas por meio das heurísticas. Uma dada função foi enviada para um único participante, de forma que um número maior de funções fosse avaliado no *survey*. Um formulário foi reenviado a outra pessoa apenas em dois casos: (a) se o participante não respondeu dentro do prazo de aproximadamente uma semana, ou (b) se errou qualquer uma das duas funções de controle.

```
1  /* compiere/base/src/org/compiere/model/MBankStatement.java */
2  @UICallout public void setC_BankAccount_ID(String oldC_BankAccount_ID,
       String newC_BankAccount_ID, int windowNo) throws Exception
3  {
4    if ((newC_BankAccount_ID == null) || (newC_BankAccount_ID.length() == 0))
5      return;
6    int C_BankAccount_ID = Integer.parseInt(newC_BankAccount_ID);
7    if (C_BankAccount_ID == 0)
8      return;
9    setC_BankAccount_ID(C_BankAccount_ID);
10   MBankAccount ba = getBankAccount();
11   setBeginningBalance(ba.getCurrentBalance());
12 }
```

Código 5.13: Exemplo de método claramente não utilitário usado nos formulários como ponto de controle.

A Tabela 5.4 apresenta o número de funções identificadas por meio das heurísticas e o número de funções usadas nos formulários. Dentre as recomendações, 144 métodos Java escolhidos aleatoriamente foram distribuídos em 18 formulários, assim como 120

funções JavaScript foram divididas em outros 15 formulários. Isso correspondeu a 3.85% das recomendações para sistemas Java e 8.52% para JavaScript. Os formulários foram enviados para desenvolvedores com conhecimento nas respectivas linguagens.

Tabela 5.4: Número de funções identificadas por meio das heurísticas e de funções avaliadas nos formulários.

Linguagem	F. identificadas	Formulários	F. avaliadas	% F. avaliadas
Java	3733	18	144	3.85%
JavaScript	1407	15	120	8.52%

5.3.1 Resultados

A Tabela 5.5 exibe o número total de formulários, o número de participantes para quem eles foram enviados, quantas respostas foram obtidas e quantas respostas tiveram erro nas funções de controle, ou seja, pelo menos uma das funções claramente não utilitárias recebeu resposta dois (possivelmente utilitária) ou três (certamente utilitária). Para Java, 20 dos 25 desenvolvedores para quem os formulários foram enviados responderam, o que corresponde a uma taxa de resposta de 80%. Dois participantes erraram um dos métodos de controle. Em três formulários não houve resposta no primeiro envio. Em dois formulários os participantes erraram algum ponto de controle no primeiro envio e não houve resposta no segundo envio. Os formulários em JavaScript foram enviados para 21 desenvolvedores; destes, 18 responderam (taxa de resposta de 85%) e três erraram algum ponto de controle. Para dois formulários, os participantes erraram algum ponto de controle no primeiro envio. Em um formulário, não houve resposta no primeiro envio. Em um formulário o participante errou algum ponto de controle no primeiro envio e não houve resposta no segundo e terceiro envios.

Tabela 5.5: Número de formulários enviados, respondidos e de respostas com erro nas funções de controle.

Linguagem	Formulários	Envios	Respostas	Erros
Java	18	25	20	2
JavaScript	15	21	18	3

As Figuras 5.2 e 5.3 apresentam a experiência dos participantes do *survey*. Neles foram contabilizados apenas os participantes com respostas válidas, ou seja, excluem-se aqueles que erraram as funções de controle.

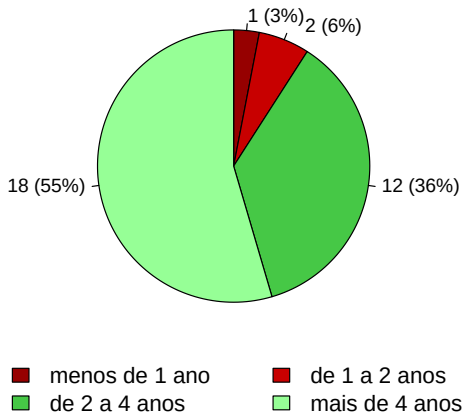


Figura 5.2: Experiência dos participantes em desenvolvimento de software.

Experiência dos participantes em desenvolvimento de software: o nível de experiência dos participantes em desenvolvimento de software, informado pelos mesmos, é exibido na Figura 5.2. Verifica-se que, 18 participantes possuem mais de quatro anos de experiência, correspondendo a 55% deles. Doze desenvolvedores possuem entre dois e quatro anos de experiência (36%), dois participantes possuem de um a dois anos e apenas um possui menos de um ano de experiência. Ou seja, 90% dos participantes possuem mais de dois anos de experiência em desenvolvimento de software.

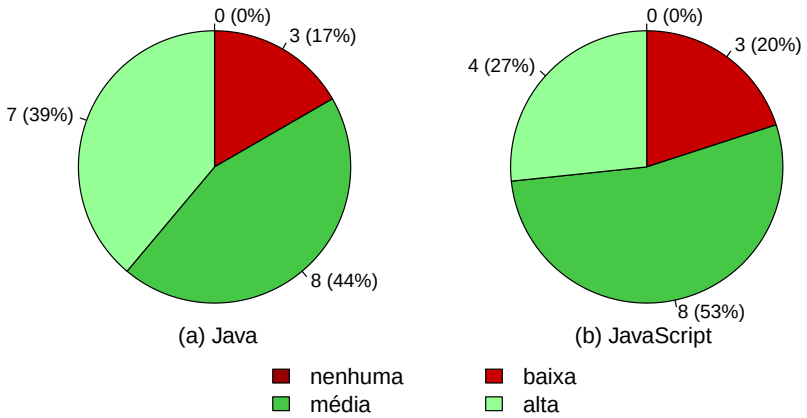


Figura 5.3: Experiência dos participantes nas linguagens Java e JavaScript.

Experiência dos participantes em Java e JavaScript: a Figura 5.3 exibe o nível de experiência dos participantes respectivamente em Java e JavaScript. Para Java, sete participantes informaram ter alto domínio da linguagem, o que corresponde a 39% dos participantes. Outros oito participantes (44%) declararam ter experiência média e três possuem baixa experiência. Nos formulários para JavaScript, oito participantes

declararam ter média experiência, correspondendo a 53% dos participantes. Outros quatro participantes têm experiência alta e três possuem baixo domínio. Esses resultados mostram que a grande maioria dos participantes possui boa experiência em desenvolvimento de software e em ambas as linguagens, o que é importante para obter respostas mais confiáveis.

Funções de controle: a Tabela 5.6 apresenta as respostas obtidas nas funções de controle, aquelas claramente de domínio específico, utilizadas para validar as respostas. Para Java, no método de controle I, 16 participantes marcaram a opção zero (certamente não utilitário), correspondendo a 80% das respostas, dois participantes marcaram a opção 1 (possivelmente não utilitário) e dois erraram, respondendo que possivelmente é utilitário. Já o método de controle II foi indicado como não utilitário por 17 participantes, sendo 85% das respostas. Foi marcado como possivelmente não utilitário por dois participantes e apenas um errou, marcando a opção 2 (possivelmente utilitário). Para JavaScript, na primeira função, dez participantes marcaram a opção zero (certamente não utilitária), o que corresponde a 56% das respostas. Seis desenvolvedores marcaram a opção 1 (possivelmente não utilitária), sendo 33% das respostas, e dois erraram, indicando-a como possivelmente utilitária. Na segunda função, 14 participantes, ou seja, 78% deles, a definiram como certamente não utilitária, três desenvolvedores a marcaram como possivelmente não utilitária e um errou, escolhendo a opção 2 (possivelmente utilitária). Esses resultados mostraram que a maioria dos participantes acertaram os pontos de controle. No entanto, a primeira função de controle para os formulários JavaScript pode não ter sido uma boa escolha, dado que para uma parcela considerável dos participantes ela não é claramente de propósito específico (44% marcaram as opções 1 ou 2).

Tabela 5.6: Número de respostas para os pontos de controle.

		É utilitária?				
Linguagem	**Função**	**(não) 0**	**1**	**2**	**3 (sim)**
Java	PC I	16	2	2	0
	PC II	17	2	1	0
JavaScript	PC I	10	6	2	0
	PC II	14	3	1	0

Precisão: os resultados da classificação das funções, segundo os participantes do *survey*, são mostrados na Figura 5.4. Respectivamente para Java e para JavaScript, 43% e 42% das funções foram classificadas como certamente utilitárias. Aquelas indicadas

como possivelmente utilitárias são, respectivamente para Java e JavaScript, 24% e 26%, as possivelmente não utilitárias correspondem a 17% e 11%, e 17% e 22% das funções não são utilitárias segundo os participantes. Assim, conclui-se que, nesta amostra aleatória das funções detectadas pelas heurísticas propostas neste artigo, 66% dos métodos Java e 67% das funções em JavaScript podem ser considerados utilitários (respostas nos níveis 2 e 3 da escala adotada).

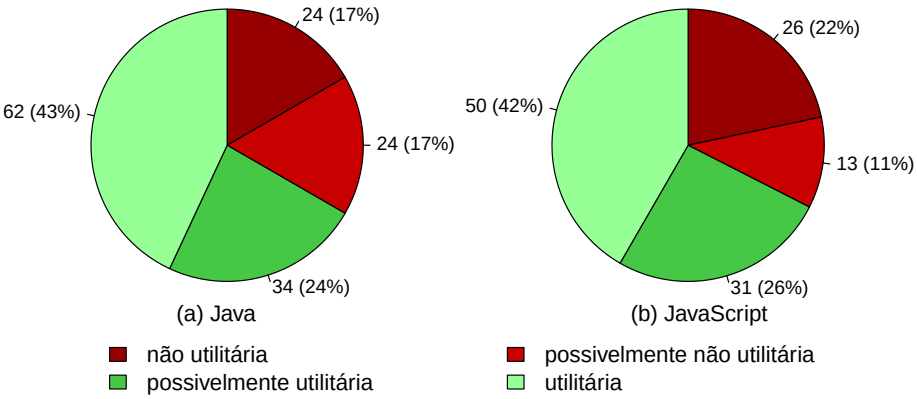


Figura 5.4: Classificação das funções de acordo com as respostas dos formulários.

Resumo: o *survey* realizado com 33 desenvolvedores, avaliando uma amostra aleatória de 5.13% do total de recomendações mostrou que 66% dos métodos em Java e 67% das funções em JavaScript podem ser considerados utilitários. Esta é uma precisão aproximada da identificação de funções utilitárias em módulos não utilitários obtida por meio das heurísticas propostas. A maioria dos participantes do *survey* possui mais de quatro anos de experiência em desenvolvimento de software, correspondendo a 54% do total de participantes, e a maioria dos desenvolvedores possuem alta ou média experiência nas linguagens, equivalendo a 81% do total de participantes.

5.4 Ameaças à Validade dos Resultados

Uma ameaça à validade dos resultados é que a análise das funções identificadas no *survey* é subjetiva, pois envolve a opinião de pessoas. No entanto, a experiência da maioria dos participantes aumenta a chance de se obter uma resposta tecnicamente mais precisa. Além disso, uma definição de funções utilitárias (conforme consideradas neste trabalho) foi inserida nos formulários para que os participantes baseassem suas respostas.

Devido ao grande número de funções identificadas, foi inviável avaliar cada uma delas, sendo os resultados do *survey* correspondentes a 3,8% e 8,5% das funções re-

comendadas para Java e JavaScript, respectivamente. Contudo, esta é uma amostra aleatória das funções e considera-se um resultado representativo do total.

5.5 Considerações Finais

Neste capítulo foi apresentado um conjunto de heurísticas para identificar funções utilitárias baseadas em propriedades de código obtidas por meio de análise estática. Os resultados mostraram que essas heurísticas são capazes de identificar funções utilitárias com boa precisão, mas com *recall* baixo. Os resultados são melhores que a abordagem de aprendizado de máquina, levando em consideração uma abordagem mais próxima de um cenário de aplicação prática. O *ranking* de preditores do *Random Forests* não se mostrou efetivo para recomendar a configuração das heurísticas, cabendo ao especialista do sistema desabilitar as regras que julgar adequadas ao mesmo. Os resultados obtidos no *survey* também mostram uma precisão de aproximadamente 66% na identificação de funções utilitárias implementadas em módulos de propósito específico.

Capítulo 6

Conclusão

Funções utilitárias, geralmente, são definidas em bibliotecas próprias, para facilitar o reuso. No entanto, é comum encontrá-las implementadas junto a funções de propósito específico, aumentando as chances de gerar retrabalho e duplicação de código. Através de um estudo exploratório realizado, mostrou-se que esse problema ocorre na prática, ou seja, os desenvolvedores implementam funções utilitárias fora de módulos utilitários. Uma das formas de resolver esse problema é mover a função utilitária para um módulo adequado. Porém, com a falta de ferramentas de suporte para identificar tais funções, esse tipo de refatoração se torna custosa para os desenvolvedores em grandes sistemas. Além disso, o ferramental voltado para refatoração e modularização ainda é pobre no caso de linguagens fracamente tipadas e dinâmicas como JavaScript. Para suprir essa deficiência, nesta dissertação de mestrado foi proposto um conjunto de heurísticas baseadas em métricas de código estáticas para identificar funções utilitárias.

Foi investigado, inicialmente, o uso de aprendizado de máquina para resolver o problema de pesquisa. Por meio do estudo exploratório concluiu-se que a maioria das funções implementadas em módulos utilitários é, de fato, utilitária. Logo, um classificador *Random Forests* foi treinado, com base nessa suposição, para identificar funções utilitárias. Foram realizados treinamentos e testes *intra-project* e *cross-project* em sistemas proprietários e de código aberto em Java e JavaScript. Os resultados mostraram que, apesar de o classificador apresentar um bom desempenho na avaliação *intra-project*, ele não foi capaz de generalizar a identificação de funções utilitárias na avaliação *cross-project*, que é mais próxima de um cenário de aplicação prática. Nesse caso, os resultados obtidos foram muito inferiores. Por exemplo, nos sistemas de código aberto, a precisão caiu pela metade com a segunda abordagem. Para os sistemas proprietários, obteve-se precisão média de 84% na avaliação *intra-project* e 0.75% na avaliação *cross-project*.

Portanto, para identificar funções utilitárias implementadas em módulos não utilitários foram propostas heurísticas que podem ser computadas via análise estática do código fonte de sistemas em Java e JavaScript. Logo, essas heurísticas podem ser integradas a ferramentas dos ambientes de desenvolvimento de software atuais. Os resultados obtidos com as heurísticas propostas são superiores aos obtidos com a abordagem de aprendizado de máquina. Avaliando essa nova abordagem na identificação de funções utilitárias de quatro sistemas proprietários de uma empresa de desenvolvimento de software brasileira, obteve-se uma precisão média de 68%. Ademais, obteve-se uma precisão de 66% e 67% respectivamente para Java e JavaScript, na identificação de funções utilitárias implementadas em módulos não utilitários, considerando uma amostra das funções de sistemas de código aberto em um *survey* realizado com 33 desenvolvedores.

6.1 Contribuições

Esta pesquisa forneceu as seguintes contribuições:

- Uma classificação manual das funções de quatro sistemas proprietários de uma empresa de desenvolvimento de software brasileira como utilitárias ou não (Capítulo 3);
- Uma investigação da técnica de aprendizado de máquina para identificação de funções utilitárias usando o classificador *Random Forests* e preditores baseados em métricas estáticas de código (Capítulo 4);
- Um conjunto de heurísticas baseadas em métricas obtidas por meio de análise estática para identificar funções utilitárias em Java e JavaScript (Capítulo 5);
- Avaliação e comparação de duas abordagens para identificar funções utilitárias em quatro sistemas da indústria e 106 sistemas de código aberto em Java e JavaScript (Capítulos 4 e 5);
- Um *survey* realizado com 33 desenvolvedores que avaliaram 264 funções de sistemas de código aberto como utilitárias ou não (Capítulo 5).

6.2 Limitações

O trabalho possui as seguintes limitações:

- A classificação manual das funções nos sistemas proprietários foi feita com base na opinião de apenas um especialista;
- A definição de módulo utilitário usada na pesquisa não é totalmente precisa, pois assumiu-se que módulos utilitários têm a palavra *"util"* em seu diretório;
- Os desenvolvedores que participaram do *survey* realizado não são especialistas nos respectivos sistemas nos quais as funções avaliadas foram implementadas;
- Apenas uma parcela das funções utilitárias identificadas por meio das heurísticas foi avaliada no *survey* realizado.

6.3 Trabalhos Futuros

Esta pesquisa pode ser complementada com os seguintes trabalhos futuros:

- *Abordagem proposta:* (i) implementar as heurísticas propostas em ferramentas que podem ser integradas aos ambientes de desenvolvimento de software em Java e em JavaScript para recomendar que as funções utilitárias implementadas em módulos inadequados sejam movidas para bibliotecas utilitárias; (ii) verificar pré-condições para certificar-se que as funções podem de fato ser movidas para outros módulos preservando o comportamento dos programas; (iii) investigar métricas de código obtidas dinamicamente para melhorar as heurísticas usadas nos sistemas em JavaScript.
- *Abordagem via aprendizado de máquina:* (i) investigar mais detalhadamente os motivos da diferença entre os resultados obtidos com as abordagens *intra-project* e *cross-project* usando o classificador *Random Forests*; (ii) avaliar o uso de outros algoritmos de aprendizado de máquina na identificação de funções utilitárias.
- *Avaliação:* (i) avaliar um maior número de funções identificadas por meio das heurísticas propostas envolvendo a opinião de desenvolvedores de software experientes; (ii) realizar uma avaliação das funções identificadas por meio das heurísticas considerando a opinião dos especialistas dos sistemas de código aberto; (iii) comparar os resultados obtidos através das heurísticas com outras soluções que recomendam refatorações para mover funções (como exemplo, com as recomendações produzidas pelos sistemas *JDeodorant* [Fokaefs et al., 2007] e *JMove* [Sales et al., 2013]).

Apêndice A

Dataset

Neste apêndice, são listados os sistemas de código aberto do *dataset*. Uma breve descrição é fornecida, assim com o número de linhas de código de cada sistema. A Tabela A.1 lista os sistemas em Java e a Tabela A.2 mostra os sistemas em JavaScript.

Tabela A.1: 84 sistemas de código aberto em Java do *Qualitas Corpus*.

Sistema	Descrição	KLOC
Ant	Ferramenta de compilação	123
AOI	Ferramenta de modelagem e renderização 3D	103
ArgoUML	Ferramenta de modelagem UML	179
AspectJ	Linguagem de programação orientada a aspectos	377
Axion	*Engine* de banco de dados relacional	23
Azureus	Aplicação para compartilhamento de arquivos	518
Batik	Biblioteca para manipulação e renderização de SVG	174
C-JDBC	*Middleware* de *cluster* de banco de dados	88
Castor	*Framework* para mapeamento objeto relacional	130
Cayenne	*Framework* para mapeamento objeto relacional	120
Checkstyle	Ferramenta de análise estática de código	22
Cobertura	Ferramenta para calcular cobertura de teste	43
Collections	Coleção de componentes reusáveis	32
Columba	Cliente de e-mail	61
Compiere	Sistema de ERP e CRM	430
Derby	Banco de dados relacional	495
DisplayTag	Biblioteca de customização de *tags* HTML	13
DrawSWF	Aplicação para gerar animações em *Flash*	23

DrJava	IDE para Java	56
Emma	Ferramenta para calcular cobertura de teste	20
Exoportal	Plataforma de colaboração corporativa	45
Findbugs	Ferramenta para detecção de *bugs*	93
Fitlibrary	Biblioteca de teste	20
Freecol	Jogo de estratégia de colonização	99
Freecs	Servidor de chat	19
Freemind	Programa para mapeamento mental	39
Galleon	Servidor de mídia para gravador de vídeo digital	82
GanttProject	Gerenciamento e agendamento de projetos	30
Hadoop	Plataforma de computação distribuída	168
Heritrix	*Crawler* para criação de histórico de páginas web	57
Hibernate	*Framework* de persistência	154
HSQLDB	Gerenciador de banco de dados	159
HtmlUnit	*Framework* de teste para aplicações web	46
Informa	Biblioteca de RSS	9
iReport	*Framework* para geração de relatórios	192
iText	Biblioteca para geração de PDF	75
JAG	Ferramenta para criação de aplicações J2EE	15
James	Servidor de e-mail	31
JasperReports	*Framework* para geração de relatórios	155
JChempaint	Editor gráfico para estruturas químicas	126
JEdit	Editor de código	108
Jena	*Framework* para Web Semântica	65
Jext	Editor de código	55
JFreechart	Biblioteca para renderização de gráficos	132
JGraph	Biblioteca para renderização gráfica	28
JGraphpad	Biblioteca para renderização de gráficos	23
JGrapht	Biblioteca de grafos	12
JGroups	Toolkit para comunicação em grupo	52
JHotDraw	*Framework* para editores gráficos	79
JMeter	Ferramenta para realizar testes de carga	65
JoggPlayer	Aplicação para reprodução de áudio	31
JPF	*Framework* para sistemas extensíveis	12
JRat	Analisador de desempenho	12
JRefactory	Ferramenta de refatoração	114

JSPWiki	*Engine* para *Wiki*	51
jsXe	Editor de XML	9
JTOpen	Biblioteca de suporte a modelos de sistemas *IBM i*	377
Jung	*Framework* para manipulação de grafos e redes	28
Log4j	Biblioteca de *logging*	19
Lucene	Sistema de busca e indexação de documentos	184
MegaMek	Jogo de estratégia de guerra	205
MvnForum	Plataforma de fórum	98
MyFaces	*Framework* para desenvolvimento web	126
NakedObjects	*Framework* para sistemas orientados a domínio	60
OpenJms	API *Java Message Service*	42
PMD	Analisador de código fonte	37
POI	API para manipular documentos do *Microsoft Office*	131
Pooka	Cliente de e-mail	50
ProGuard	Ferramenta para otimizar e ofuscar código Java	49
Quartz	Biblioteca para agendar tarefas	32
QuickServer	Biblioteca para aplicações com protocolo TCP	13
Roller	Plataforma para blog	52
RSSOwl	Leitor de RSS	79
SandMark	Ferramenta para estudo de proteção de software	81
Spring	*Framework* para desenvolvimento de aplicações Java	154
Struts	*Framework* para aplicações MVC	82
SunFlow	Sistema de renderização de imagens foto-realistas	17
Tapestry	*Framework* para aplicações web	45
Tomcat	Servidor web	179
Velocity	*Engine* de *templates* para referenciar objetos Java	32
WCT	Ferramenta para gerenciamento de fluxo de trabalho	36
Weka	Coleção de algoritmos de aprendizado de máquina	322
Xalan	Biblioteca para conversão de documentos XML	249
Xerces	*Parser* para XML	124

Tabela A.2: 22 sistemas de código aberto em JavaScript do *GitHub*.

Sistema	Descrição	KLOC
Ace	Editor de código	229
Bower	Gerenciador de pacotes	7
Brackets	Editor de código	57
Echarts	Biblioteca de gráficos e visualização	32
Ghost	Plataforma de publicação de blogs	18
Gitbook	Ferramenta para construir livros	5
Ionic	*Framework* para desenvolvimento *mobile* em HTML5	72
Leaflet	Biblioteca para interação com mapas	7
Markdown Here	Extensão para formatar e-mail em HTML	6
Material	*Framework* para interface para *AngularJS*	24
Material-UI	Componentes para interface do *Material*	1
Mocha	*Framework* de teste	8
Mongoose	Ferramenta de modelagem de objetos para *MongoDB*	33
Node Inspector	*Debugger* para o *Node.js*	119
PDF.js	Leitor de PDF em HTML 5	32
Phaser	*Framework* de jogos para HTML5	62
Pixi.js	Renderizador 2D em HTML5	14
React	Biblioteca do *Facebook* para construir interfaces	10
React Native	*Framework* para construir aplicativos com o *React*	8
Three.js	Biblioteca para renderização 3D	35
TimelineJS	*Timeline* para *Storytelling*	16
Typeahead.js	Biblioteca do *Twitter* para busca de *autocomplete*	2

Apêndice B

Formulário Usado no Survey

Neste apêndice, é apresentado um exemplo de formulário, referente à linguagem Java, usado no *survey*. O formato dos formulários aplicados para as linguagens Java e JavaScript foi o mesmo, mudando apenas as terminologias "método" e "função" e as próprias funções de cada questionário.

Identificação

*Obrigatório

1. **Nome**

   .....

2. **Experiência em desenvolvimento de software** *

   Marcar apenas uma oval.

   - ◯ Menos de 1 ano
   - ◯ De 1 a 2 anos
   - ◯ De 2 a 4 anos
   - ◯ Mais de 4 anos

3. **Experiência em Java** *

   Marcar apenas uma oval.

   - ◯ Nenhuma
   - ◯ Baixa (trabalhou poucas vezes, pouco conhecimento)
   - ◯ Média
   - ◯ Alta (amplo domínio)

Classificação de métodos utilitários

Métodos utilitários são métodos de propósito geral, não relacionados a um domínio ou sistema específico. Portanto, eles podem ser reutilizados em outros sistemas. Como exemplo, podemos citar métodos para manipulação de datas, strings, estruturas de dados e métodos para facilitar o uso de bibliotecas externas.

Neste formulário, apresentamos 10 métodos implementados em Java e solicitamos que avalie cada um deles como utilitário ou não, numa escala de 0 a 3. Essa avaliação pode levar de 10 a 15 minutos apenas.

IMPORTANTE: junto do método, incluímos um link para o arquivo onde ele está implementado. A consulta a esse arquivo pode te ajudar a entender melhor o contexto de implementação do método e, portanto, a nos fornecer uma resposta mais precisa.

Método 1

```
199    public static double calculateMedian(List values, boolean copyAndSort) {
200        double result = Double.NaN;
201        if (values != null) {
202            if (copyAndSort) {
203                int itemCount = values.size();
204                List copy = new ArrayList(itemCount);
205                for (int i = 0; i < itemCount; i++) {
206                    copy.add(i, values.get(i));
207                }
208                Collections.sort(copy);
209                values = copy;
210            }
211            int count = values.size();
212            if (count > 0) {
213                if (count % 2 == 1) {
214                    if (count > 1) {
215                        Number value = (Number) values.get((count - 1) / 2);
216                        result = value.doubleValue();
217                    } else {
218                        Number value = (Number) values.get(0);
219                        result = value.doubleValue();
220                    }
221                } else {
222                    Number value1 = (Number) values.get(count / 2 - 1);
223                    Number value2 = (Number) values.get(count / 2);
224                    result = (value1.doubleValue() + value2.doubleValue()) / 2.0;
225                }
226            }
227        }
228        return result;
229    }
```

Localizado na linha 199 da classe Statistics disponível em:
https://www.dropbox.com/s/ybr32pgm0yazk0v/Statistics.java?dl=0

4. **O método 1 é utilitário?** *
   Marcar apenas uma oval.

	0	1	2	3	
Certamente NÃO é utilitário	◯	◯	◯	◯	Certamente é utilitário

Método 2

```
154
155    public static boolean iteratorsContentEqual(Iterator iter1, Iterator iter2) {
156        while (iter1.hasNext() && iter2.hasNext()) {
157            if (!iter1.next().equals(iter2.next())) {
158                return false;
159            }
160        }
161
162        //noinspection RedundantIfStatement
163        if (iter1.hasNext() || iter2.hasNext()) {
164            return false;
165        }
166
167        return true;
168    }
169
```

Localizado na linha 155 da classe Tools disponível em:
https://www.dropbox.com/s/r75t07vmzufq7d7/Tools.java?dl=0

5. **O método 2 é utilitário?** *
   Marcar apenas uma oval.

	0	1	2	3	
Certamente NÃO é utilitário	◯	◯	◯	◯	Certamente é utilitário

Método 3

```
54
55      public static String convertSpecialChars(final String s) {
56          char c;
57          final int len = s.length();
58          final StringBuffer sbuf = new StringBuffer(len);
59
60          int i = 0;
61          while (i < len) {
62              c = s.charAt(i++);
63              if (c == '\\') {
64                  c = s.charAt(i++);
65                  if (c == 'n') {
66                      c = '\n';
67                  } else if (c == 'r') {
68                      c = '\r';
69                  } else if (c == 't') {
70                      c = '\t';
71                  } else if (c == 'f') {
72                      c = '\f';
73                  } else if (c == '\b') {
74                      c = '\b';
75                  } else if (c == '\"') {
76                      c = '\"';
77                  } else if (c == '\'') {
78                      c = '\'';
79                  } else if (c == '\\') {
80                      c = '\\';
81                  }
82              }
83              sbuf.append(c);
84          }
85          return sbuf.toString();
86      }
```

Localizado na linha 55 da classe OptionConverter disponível em:
https://www.dropbox.com/s/0la0lqeluuuo7a1/OptionConverter.java?dl=0

6. **O método 3 é utilitário?** *
   Marcar apenas uma oval.

	0	1	2	3	
Certamente NÃO é utilitário	◯	◯	◯	◯	Certamente é utilitário

Método 4

```
224    @UICallout public void setC_BankAccount_ID (String oldC_BankAccount_ID,
225            String newC_BankAccount_ID, int windowNo) throws Exception
226    {
227        if ((newC_BankAccount_ID == null) || (newC_BankAccount_ID.length() == 0))
228            return;
229        int C_BankAccount_ID = Integer.parseInt(newC_BankAccount_ID);
230        if (C_BankAccount_ID == 0)
231            return;
232        setC_BankAccount_ID(C_BankAccount_ID);
233        //
234        MBankAccount ba = getBankAccount();
235        setBeginningBalance(ba.getCurrentBalance());
236    }
```

Localizado na linha 224 da classe MBankStatement disponível em:
https://www.dropbox.com/s/a6az5a5cvy8g47v/MBankStatement.java?dl=0

7. **O método 4 é utilitário?** *
   Marcar apenas uma oval.

	0	1	2	3	
Certamente NÃO é utilitário	◯	◯	◯	◯	Certamente é utilitário

Método 5

```
326
327    public final static boolean isNumberType(final Class<?> type) {
328        return type == Long.TYPE || type == Double.TYPE ||
329            type == Byte.TYPE || type == Short.TYPE ||
330            type == Integer.TYPE || type == Float.TYPE ||
331            Number.class.isAssignableFrom(type);
332    }
333
```

Localizado na linha 327 da classe ELArithmetic disponível em:
https://www.dropbox.com/s/wi3muhv9pyctvty/ELArithmetic.java?dl=0

8. **O método 5 é utilitário?** *
   Marcar apenas uma oval.

	0	1	2	3	
Certamente NÃO é utilitário	◯	◯	◯	◯	Certamente é utilitário

Método 6

```
535
536     public boolean isAInteraction(Object handle) {
537         return handle instanceof Interaction;
538     }
539
```

Localizado na linha 536 da classe FacadeMDRImpl disponível em:
https://www.dropbox.com/s/0ij030pwfkm2ecp/FacadeMDRImpl.java?dl=0

9. **O método 6 é utilitário?** *
   Marcar apenas uma oval.

	0	1	2	3	
Certamente NÃO é utilitário	◯	◯	◯	◯	Certamente é utilitário

Método 7

```
368     public static String toLowerCase(String v)
369     {
370         if (v == null)
371         {
372             return "";
373         }
374         if (v.length() == 0)
375         {
376             return "";
377         }
378
379         return v.toLowerCase();
380     }
381
```

Localizado na linha 368 da classe JstlFunction disponível em:
https://www.dropbox.com/s/m57o7u97r2bbbx0/JstlFunction.java?dl=0

10. **O método 7 é utilitário?** *
    Marcar apenas uma oval.

	0	1	2	3	
Certamente NÃO é utilitário	◯	◯	◯	◯	Certamente é utilitário

Método 8

```
349     public String getUniqueVariableName(String prefix) {
350         if (prefix == null || prefix.equals("")) prefix = "var";
351         Random rnd = new Random();
352         long uid = ((System.currentTimeMillis() >>> 16) << 16) + rnd.nextLong();
353         return prefix + String.valueOf(Math.abs(uid)).trim();
354     }
355
```

Localizado na linha 349 da classe RModel disponível em:
https://www.dropbox.com/s/5s9lmry3jbv313a/RModel.java?dl=0

11. **O método 8 é utilitário?** *
    Marcar apenas uma oval.

	0	1	2	3	
Certamente NÃO é utilitário	◯	◯	◯	◯	Certamente é utilitário

Método 9

```
480     public void setCreditCardType (String CreditCardType)
481     {
482         if (!isCreditCardTypeValid(CreditCardType))
483         throw new IllegalArgumentException ("CreditCardType Invalid value - " + CreditCardType +
484                                             " - Reference_ID=149 - A - C - D - M - N - P - V");
485         set_Value ("CreditCardType", CreditCardType);
486
487     }
```

Localizado na linha 480 da classe X_C_BP_BankAccount disponível em:
https://www.dropbox.com/s/5c85ijodpjfy3zv/X_C_BP_BankAccount.java?dl=0

12. **O método 9 é utilitário?** *
    Marcar apenas uma oval.

	0	1	2	3	
Certamente NÃO é utilitário	◯	◯	◯	◯	Certamente é utilitário

Método 10

```
10      public boolean allowsCredit(int months, boolean reliable, double balance) {
11          return months > 12 && reliable && balance < 6000;
12      }
```

Localizado na linha 10 da classe Credit disponível em:
https://www.dropbox.com/s/tsf6doc6l454ic8/Credit.java?dl=0

13. **O método 10 é utilitário?** *
Marcar apenas uma oval.

	0	1	2	3	
Certamente NÃO é utilitário	◯	◯	◯	◯	Certamente é utilitário

Powered by
Google Forms

Referências Bibliográficas

Abebe, S. L.; Arnaoudova, V.; Tonella, P.; Antoniol, G. & Guéhéneuc, Y.-G. (2012). Can lexicon bad smells improve fault prediction? Em *19th Working Conference on Reverse Engineering (WCRE)*, pp. 235–244.

Archer, K. J. & Kimes, R. V. (2008). Empirical characterization of Random Forest variable importance measures. *Computational Statistics & Data Analysis*, 52(4):2249–2260.

Avelino, G.; Passos, L.; Hora, A. & Valente, M. T. (2016). A novel approach for estimating truck factors. Em *24th International Conference on Program Comprehension (ICPC)*, pp. 1–10.

Bavota, G.; Oliveto, R.; Gethers, M.; Poshyvanyk, D. & Lucia, A. D. (2014). Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 40(7):671–694.

Bloch, J. (2008). *Effective Java*. Prentice Hall PTR, 2ª edição.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.

Breiman, L. (2001). Random Forests. *Machine Learning*, 45(1):5–32.

Caruana, R.; Karampatziakis, N. & Yessenalina, A. (2008). An empirical evaluation of supervised learning in high dimensions. Em *25th International Conference on Machine Learning (ICML)*, pp. 96–103. ACM.

Caruana, R. & Niculescu-Mizil, A. (2006). An empirical comparison of supervised learning algorithms. Em *23rd International Conference on Machine Learning (ICML)*, pp. 161–168. ACM.

Cortes, C. & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3):273–297.

Costa, D. A.; Abebe, S. L.; McIntosh, S.; Kulesza, U. & Hassan, A. E. (2014). An empirical study of delays in the integration of addressed issues. Em *30th International Conference on Software Maintenance and Evolution (ICSME)*, pp. 281–290.

Dias, M.; Bacchelli, A.; Gousios, G.; Cassou, D. & Ducasse, S. (2015). Untangling fine-grained code changes. Em *22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 341–350.

Dietterich, T. G. (2000). An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine Learning*, 40(2):139–157.

Fokaefs, M.; Tsantalis, N. & Chatzigeorgiou, A. (2007). JDeodorant: Identification and removal of feature envy bad smells. Em *33rd IEEE International Conference on Software Maintenance (ICSM)*, pp. 519–520.

Fowler, M. (1999). *Refactoring - Improving the Design of Existing Code*. Addison-Wesley.

Friedberg, R. M. (1958). A learning machine: Part i. *IBM Journal of Research and Development*, 2(1):2–13.

Guyon, I.; Weston, J.; Barnhill, S. & Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. *Machine Learning*, 46(1):389–422.

Han, J.; Kamber, M. & Pei, J. (2011). *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 3ª edição.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076.

Lessmann, S.; Baesens, B.; Mues, C. & Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496.

Liaw, A. & Wiener, M. (2002). Classification and regression by randomforest. *R News*, 2(3):18–22.

Mann, H. B. & Whitney, D. R. (1947). On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60.

Mendes, T.; Valente, M. T. & Hora, A. (2016a). Identificação de funções utilitárias em Java e Javascript. Em *X Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS)*.

Mendes, T.; Valente, M. T.; Hora, A. & Serebrenik, A. (2016b). Identifying utility functions using Random Forests. Em *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pp. 614–618.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., 1ª edição.

Osman, M.; Chaudron, M. & Putten, P. V. D. (2013). An analysis of machine learning algorithms for condensing reverse engineered class diagrams. Em *29th IEEE International Conference on Software Maintenance (ICSM)*, pp. 140–149.

Peters, F.; Menzies, T. & Marcus, A. (2013). Better cross company defect prediction. Em *10th IEEE Working Conference on Mining Software Repositories (MSR)*, pp. 409–418.

Richards, G.; Lebresne, S.; Burg, B. & Vitek, J. (2010). An analysis of the dynamic behavior of Javascript programs. *SIGPLAN Not.*, 45(6):1–12.

Robillard, M.; Walker, R. & Zimmermann, T. (2010). Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86.

Russell, S. J. & Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc.

Sales, V.; Terra, R.; Miranda, L. F. & Valente, M. T. (2013). Recommending move method refactorings using dependency sets. Em *20th Working Conference on Reverse Engineering (WCRE)*, pp. 232–241.

Sarkar, S.; Ramachandran, S.; Kumar, G. S.; Iyengar, M. K.; Rangarajan, K. & Sivagnanam, S. (2009). Modularization of a large-scale business application: A case study. *IEEE Software*, 26(2):28–35.

Scott, A. J. & Knott, M. (1974). A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, 30(3):507–512.

Silva, D.; Terra, R. & Valente, M. T. (2014). Recommending automated extract method refactorings. Em *22nd IEEE International Conference on Program Comprehension (ICPC)*, pp. 146–156.

Silva, L.; Ramos, M.; Valente, M. T.; Anquetil, N. & Bergel, A. (2015). Does Javascript software embrace classes? Em *22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 73–82.

Terra, R.; Miranda, L. F.; Valente, M. T. & Bigonha, R. S. (2013). Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *ACM SIGSOFT Software Engineering Notes*, 38(5):1–4.

Terra, R.; Valente, M. T.; Czarnecki, K. & Bigonha, R. S. (2015). A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, 45(3):315–342.

Tian, Y.; Nagappan, M.; Lo, D. & Hassan, A. E. (2015). What are the characteristics of high-rated apps? A case study on free android applications. Em *301st International Conference on Software Maintenance and Evolution (ICSME)*, pp. 301–310.

Tiwari, D. & Solihin, Y. (2012). Architectural characterization and similarity analysis of Sunspider and Google's V8 Javascript benchmarks. Em *3rd IEEE International Symposium on Performance Analysis of Systems & Software (ISPAS)*, pp. 221–232.

Tsantalis, N. & Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367.

Vuk, M. & Curk, T. (2006). ROC Curve, Lift Chart and Calibration Plot. *Metodoloski zvezki*, 3(1):89–108.

Zhang, H. (2004). The optimality of Naive Bayes. Em *17th International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pp. 562–567. AAAI Press.