Trabalho de Conclusão de Curso

Uma aplicação do aprendizado por transferência na detecção de Code Smells

André Moabson da Silva Ramos
amsr@ic.ufal.br

Orientadores:

Prof. Dr. Baldoino Fonseca dos Santos Neto
MSc. Ana Carla Gomes Bibiano

Maceió, Fevereiro de 2021

André Moabson da Silva Ramos

Uma aplicação do aprendizado por transferência na detecção de Code Smells

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientadores:

Prof. Dr. Baldoino Fonseca dos Santos Neto

MSc. Ana Carla Gomes Bibiano

Maceió, Fevereiro de 2021

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecário: Marcelino de Carvalho Freitas Neto – CRB-4 – 1767

R175a Ramos, André Moabson da Silva.
Uma aplicação do aprendizado por transferência na detecção de *Code Smells* / André Moabson da Silva Ramos. – 2021.
24 f. : il., figs. e tabs. color.

Orientador: Baldoino Fonseca dos Santos Neto.
Orientadora: Ana Carla Gomes Bibiano.
Monografia (Trabalho de Conclusão de Curso em Ciência da Computação) – Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2021.

Bibliografia: f. 23-24.

1. *Code smells* - Detecção. 2. Aprendizagem de máquina. 3. Aprendizagem por transferência. I. Título.

CDU: 004.81:159.953.5

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina.

Prof. Dr. Baldoino Fonseca dos Santos Neto - Orientador
Instituto de Computação
Universidade Federal de Alagoas

MSc. Ana Carla Gomes Bibiano - Coorientadora
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro

Prof. Dr. Ícaro Bezerra Queiroz de Araújo - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Prof. Dr. Rafael Maiani de Mello - Examinador
CEFET/RJ

Maceió, Fevereiro de 2021

Agradecimentos

Agradeço primeiramente a Deus pela minha saúde, por ter me permitindo chegar até o momento da apresentação do meu trabalho de conclusão de curso, que ocorre em meio a uma pandemia vivenciada pela humanidade.

Aos meus pais, André e Cleide, bem como todos os meus familiares que sempre me apoiaram em todas as minhas escolhas durante essa longa jornada.

Ao meu orientador, Professor Baldoino Fonseca, por seu apoio dado no desenvolvimento do presente trabalho, fornecendo sempre um direcionamento nos momentos de incerteza e pelas oportunidades que tem me proporcionado.

A minha coorientadora, Ana Carla, pelas dicas valiosas e fundamentais para escrita do trabalho, bem como as correções sugeridas.

Aos professores, Ícaro e Rafael, por aceitarem o convite para participar da banca examinadora.

A minha namorada, Bel, pelos puxões de orelha e incentivos dados nos momentos de desânimo, foram importantes para que eu concluísse esse trabalho.

Ao meu Professor e grande mentor, Ginaldo Júnior (*in memorian*), cuja sua história de vida me motivou a chegar até aqui hoje e continuará servindo como motivação para vencer todos os desafios enfrentados.

Por fim, a todos os meus colegas acadêmicos e profissionais que de alguma forma compartilharam comigo momentos e experiências durante os últimos anos.

"Success consists of going from failure to failure without loss of enthusiasm."
– Winston Churchill

Resumo

Durante o desenvolvimento de um *software*, a presença de *code smells* tem sido relacionada com a degradação na qualidade do software. Diversos estudos mostram a importância de detectar os *smells* no código fonte e aplicar refatoração. No entanto, as abordagens existentes para a detecção de *code smells* são limitadas para determinadas linguagens de programação. Nesse contexto, este trabalho visa ampliar os métodos para detecção de *code smells* utilizando o aprendizado por transferência para construir um grande conjunto de dados para treinamento e validação dos modelos de aprendizagem de máquina, utilizando as regras catalogadas e *thresholds* extraídos da ferramenta *Designite*. Coletando, assim, um total 22.687, 8.501 e 5.953 *smells* detectados em projetos das respectivas linguagens de programação, C++, Java e C#. Em seguida, nós obtivemos 72 modelos pré-treinados, e realizamos o aprendizado por transferência, que consistiu em avaliar o modelo pré-treinado para *smells* no conjunto de dados entre linguagens de programação. Nossos resultados revelaram que se escolhermos o *smell* de *design*, *Unecessary Abstraction*, e a linguagem alvo for C#, então o modelo mais apropriado para detectar esse *code smell* é o baseado no *RandomForest*, pois foi melhor dentre os outros modelos treinados no conjunto de dados da linguagem C++ para o mesmo *smell*. Esses resultados podem ajudar a desenvolvedores e pesquisadores a aplicar as mesmas estratégias de detecção de *code smells* em diferentes linguagens de programação, e utilizar modelos de treinamentos que sejam mais apropriados para cada tipo de *code smell* e linguagens de programação.

Palavras-chave: code smells, detecção, aprendizado por transferência, aprendizado de máquina.

Abstract

During the software development, the presence of code smells has been related to the degradation of the software quality. Several studies present the relevance to detect smells in the source code and to apply refactoring. However, the existing approaches to detect code smells are limited to specific programming languages. In this context, this work aims to extend the techniques of code smell detection using learning transfer to build a large dataset for training and validation of machine learning models, using cataloged rules and extracted thresholds of the Designite tool. Collecting, then, a total of total 22,687, 8,501 e 5,953 detected smells in software projects of the respective programming languages, C++, Java e C#. In a sequence, we obtained 72 pre-trained models and performed the transfer learning to evaluate the pre-trained model for smells in the dataset between programming languages. Our results revealed that the model RandomForest is the most appropriate to detect code smells like the *Unecessary Abstraction* design smell for the C# programming language. These results can help developers and researchers to apply the same code smell detection strategies for different programming languages and to apply the most appropriate training models for each code smell type and programming language.

Key-words: code smells, detection, transfer learning, machine learning.

Conteúdo

Lista de Figuras

Lista de Tabelas

1

Introdução

Neste Capítulo, será feito uma breve introdução mostrado a motivação e os objetivos (geral e específicos) do presente trabalho.

1.1 Motivação

O termo *code smells* foi definido por Fowler et al. (1999) como certas estruturas no código que sugerem (ou às vezes "gritam") uma refatoração, na razão direta que indicam escolhas ruins de *design* e implementação, ocasionando uma degradação na qualidade do software (Di Nucci et al., 2018).

A presença de *code smells* muitas vezes se dá pela adoção de uma solução que seja tecnicamente inferior em detrimento dos prazos estabelecidos no projeto e/ou necessidades de implementar novas funcionalidades dentro de um curto espaço de tempo, esse fator é conhecido como *technical debt*, essa relação é estudada por Zazworka et al. (2011), que também investiga os impactos da presença de *smells* na manutenção e qualidade do *software*.

Como exemplo de um impacto causado pela presença de *code smells*, Palomba et al. (2019) mostra como certos *smells* podem afetar o consumo de energia por dispositivos móveis, com métodos onde foram detectados certos tipos de *smells* tendo um consumo de até 87 vezes maior em relação a outros.

Como visto, detectar *code smells* é uma tarefa importante pois a sua detecção e em seguida refatoração está diretamente ligada a melhorias na qualidade do software, o desenvolvimento de ferramentas para detecção de *code smells* envolve diversos métodos e o aprimoramento deles é de grande importância.

Estudos anteriores apresentam estratégias de detecção de *code smells* através de análises estáticas (Fontana et al., 2015; Liu and Zhang, 2017; Pecorelli et al., 2019). No entanto, as ferramentas existentes que utilizam essas estratégias são limitadas para detectar *code smells* somente para determinadas linguagens de programação (Fontana et al., 2015; Liu and Zhang, 2017; Pecorelli et al., 2019). Dessa forma, a literatura existente para a detecção de *code smell* é muito limitada para determinadas linguagens de programação, dificultando o reaproveitamento dessas estratégias de detecção para outras linguagens de programação.

Visto estas limitações, uma técnica que pode ser adotada para contorná-la é a técnica de aprendizado por transferência que consiste em utilizar um modelo pré-treinado em um determinado conjunto de dados para resolver uma tarefa, para construir um outro modelo que visa resolver outra tarefa, aproveitando o conhecimento obtido no primeiro e treinando o segundo com poucos exemplos, como forma de reduzir os esforços necessários para resolver a segunda tarefa (Pan and Yang, 2010; Liang et al., 2019).

Dessa forma, um modelo pré-treinado para um determinado *smell* em um conjunto de dados construído a partir de projetos de uma linguagem de programação pode servir como base para construir um outro modelo que seja capaz de detectar o mesmo *smell* em uma outra linguagem de programação. Sendo assim, seria possível abranger mais linguagens de programação com o menor esforço possível.

1.2 Objetivo Geral

O objetivo deste trabalho é avaliar a aplicação do aprendizado por transferência na detecção de *code smells* para diferentes linguagens de programação.

1.3 Objetivos Específicos

- Desenvolver uma ferramenta para detectar *code smells* através de métricas e construir um conjunto de dados para cada *smell*;
- Investigar o uso do aprendizado por transferência no contexto de detecção de *code smells* para diferentes linguagens de programação;
- Avaliar os possíveis benefícios do aprendizado por transferência em comparação com o aprendizado de máquina tradicional.

1.4 Estrutura do Trabalho

O trabalho foi organizado em cinco capítulos, composto do atual e os seguintes:

- Capítulo 2, onde será descrito a fundamentação teórica do trabalho.
- Capítulo 3, onde será descrito a metodologia utilizada para execução do experimento;
- Capítulo 4, onde será mostrado os resultados do experimento bem como as discussões;
- Capítulo 5, onde será feito as considerações finais.

2

Fundamentação teórica

Neste capítulo, será descrito a fundamentação teórica sobre os principais assuntos deste trabalho, apresentando o aprendizado por transferência e mostrando suas diferenças com o aprendizado de máquina tradicional, bem como métodos e ferramentas que são comumente utilizadas para detecção de *code smells*.

2.1 Aprendizado por transferência

O aprendizado por transferência consiste em utilizar um modelo pré-treinado em um determinado conjunto de dados para resolver uma tarefa, para construir um outro modelo que visa resolver outra tarefa, aproveitando o conhecimento obtido no primeiro e treinando o segundo com poucos exemplos, como forma de reduzir os esforços necessários para resolver a segunda tarefa.

No aprendizado de máquina tradicional, geralmente, para cada tarefa utiliza-se o mesmo conjunto de dados para treinamento e testes (Pan and Yang, 2010). A Figura 2.1 mostra a diferença entre os dois processos, onde no processo de aprendizado por transferência, mais de um modelo pré-treinado para diferentes tarefas podem ser aproveitados para obter um modelo que resolve a tarefa fim.

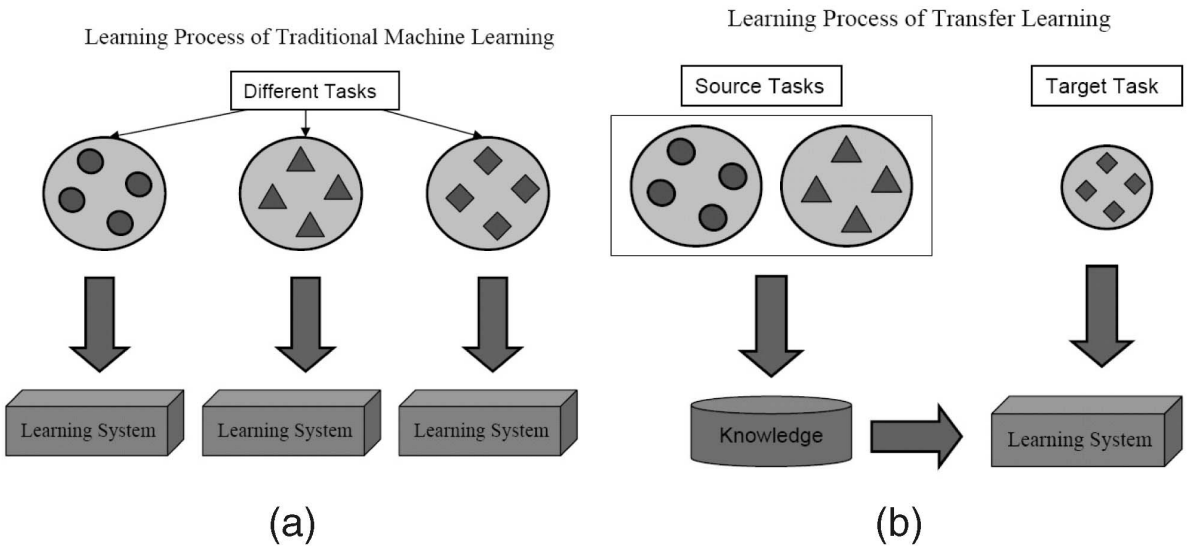

Figura 2.1: Diferença entre os processos de aprendizado, (a) aprendizado de máquina tradicional e (b) aprendizado de máquina por transferência. Fonte: Pan and Yang (2010).

No presente contexto, o estudo de Sharma et al. (2019) investigou a viabilidade de utilizar aprendizado por transferência com aprendizado profundo no contexto de detecção de *code smells*. Para a realização do experimento, o autor selecionou projetos do GitHub nas linguagens C# e Java, também foram escolhidos quatro *code smells*, sendo eles, *Complex Method*, *Empty Catch Block*, *Magic Number* e *Multifaceted Abstraction*. Este estudo foi dividido em duas partes: (i) métodos de aprendizado profundo para detectar *code smells*; e (ii) transferência aprendizado utilizando aprendizado profundo. Duas arquiteturas de redes neurais foram selecionadas, *Convolutional Neural Network* (CNN) e *Recurrent Neural Network* (RNN). O experimento consiste em treinar os modelos com amostras de código de projetos na linguagem C# previamente anotados utilizando a ferramenta de detecção de *code smells* chamada *Designite* (Sharma et al., 2016), em seguida avaliar o modelo treinado em amostradas de testes obtidas de

uma outra linguagem, no caso, a linguagem escolhida para avaliar o model treinado foi Java, os resultados indicam que é possível aplicar aprendizado profundo e aprendizado por transferência na detecção de *code smells*, com exceção do *Multifaceted Abstraction* onde ambos os modelos treinados apresentaram um baixo desempenho.

2.2 Métodos e ferramentas para detecção de Code Smells

Diversos métodos têm sido utilizadas na detecção de *code smells*, de acordo com Sharma and Spinellis (2018), esses métodos são baseados em: (i) métricas; (ii) regras / heurísticas; (iii) história; (iv) otimização; e, mais recentemente, (v) aprendizado de máquina. Nos métodos baseados em métricas, a árvore de sintaxe abstrata do código-fonte é comumente usada para calcular um conjunto de métricas, uma vez que os limiares são definidos é possível detectar smells. No entanto, alguns *code smells* não podem ser detectados apenas utilizando métricas, como *Rebellious Hierarchy*, *Missing Abstraction*, *Cyclic Hierarchy* e *Empty Catch Block*. Os métodos baseados em heurísticas são mais adequados para estes smells, neste método é definido uma regra / heurística de acordo com o *smell*. Métodos baseados em história analisam a evolução do código-fonte. Mais recentemente, métodos baseados em aprendizado de máquina são comumente utilizados para detecção de *code smells*, alguns algoritmos geralmente utilizados são: *Support Vector Machine*, *Bayesian Belief Networks* e *Logistic Regression*.

Liu and Zhang (2017) faz uma comparação de diversas ferramentas utilizadas para detecção de *code smells* mostrando para quais linguagens elas funcionam, os *smells* suportados, vantagens e desvantagens, por exemplo, a ferramenta *InFunsion* possui suporte para 7 *smells* e projetos nas linguagens Java, C e C++, porém não é gratuita, já o *JDeodorant* é um *plugin* para o *Eclipse*[1], suporta 4 *smells*, é gratuito, porém só suporta projetos na linguagem Java.

[1] www.eclipse.org

3

Metodologia

Neste capítulo, será descrito a metodologia utilizada para a execução do experimento, as linguagens de programação utilizadas, os projetos, algoritmos de aprendizado de máquina utilizados, *code smells* selecionados e construção do conjunto de dados para treinamento e métricas utilizadas para avaliação dos modelos.

3.1 Configurações do Estudo

Nosso estudo consiste em obter modelos treinados para detectar os *smells* em um determinada linguagem de programação e utilizá-los para detectar os mesmos *smells* em outras linguagens. Nós utilizamos 4 algoritmos de aprendizado de máquina supervisionado, 6 *code smells* e 3 linguagens de programação, conforme segue abaixo.

- **Algoritmos:** *Support Vector Machine*, *Decision Tree*, *Random Forest* e *Logistic Regression*;
- **Code Smells:** *Multifaceted Abstraction*, *Unnecessary Abstraction*, *Insufficient Modularization*, *Wide Hierarchy*, *Long Method* e *Complex Method*;
- **Linguagens de programação:** Java, C# e C++.

Com base nisso, as seguintes questões de pesquisa foram elaboradas:

Questão de pesquisa 1: Como construir um conjunto de dados para treinar os modelos de detecção de *code smells* para diferentes linguagens de programação?

Questão de pesquisa 2: Qual modelo pré-treinado escolher para realizar o aprendizado por transferência a depender do *smell* e linguagem alvo?

Para obter os dados necessários para responder as questões de pesquisa e alcançar o objetivo, nós seguimos os seguintes passos ilustrados na Figura 3.1 e as etapas descritas passo a passo em 3.1.1.

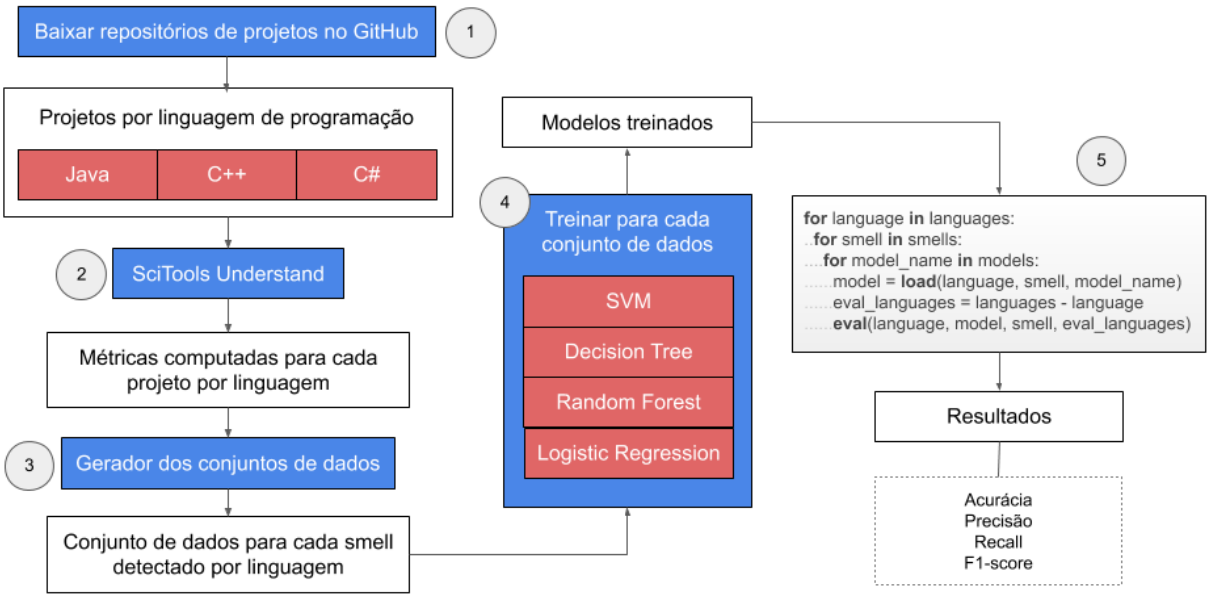

Figura 3.1: Processo de construção dos conjuntos de dados e avaliação do aprendizado por transferência.

3.1.1 Etapas do estudo

1. Selecionar e baixar 20 repositórios de projetos para cada linguagem definida, os repositórios foram selecionados manualmente do GitHub[1] observando critérios de quantidade de estrelas, *forks*, *commits* e data da última atualização.

2. Executar a ferramenta *SciTools Understand*[2] para coletar as métricas dos projetos, dentre diversas funcionalidades a ferramenta permite computar métricas para diversas linguagens incluindo as escolhidas para o experimento.

3. Gerar os conjuntos de dados que vão ser utilizados para treinar os modelos, onde, dado um conjunto de *smells*, suas respectivas regras e as métricas computadas na etapa anterior o gerador irá criar, para cada linguagem, um arquivo com uma coluna que vai assumir dois valores, "SIM" caso a regra do *smell* seja satisfeita ou "NÃO" caso contrário.

4. Treinar os modelos com os conjuntos de dados gerados na etapa anterior e salvá-los para uso posterior.

5. Executar o aprendizado por transferência, conforme o pseudocódigo na Figura 3.1. O resultado desta etapa será um arquivo com as métricas necessárias para avaliação dos modelos, sendo elas, acurácia, precisão, *recall* e *F1-score*.

3.2 Regras para detecção dos *Code Smells*

As regras para detecção dos *code smells* selecionados foram extraídas da ferramenta *DesigniteJava*[3], bem como os respectivos valores de *thresholds*. A ferramenta possui uma versão de código aberto, disponível no GitHub. As regras são importantes para construção dos conjuntos de dados que servirão para treinar e avaliar os modelos.

Para detectar os *smells* com as regras extraídas da ferramenta *Designite* foi necessário relacionar a nomenclatura das métricas utilizadas com a da ferramenta *SciTools Understand*, a tabela 3.2 mostra essa correspondência.

Nome	**Abrev.**	***SciTools Understand***	**Granularidade**
Lack of Cohesion in Methods	LCOM	PercentLackOfCohesion	Classe
Number of Fields	NOF	CountDeclClassVariable + CountDeclInstanceVariable	Classe
Number of Methods	NOM	CountDeclMethod	Classe
Number of Public Methods	NOPM	CountDeclMethodPublic	Classe

[1] https://github.com
[2] https://www.scitools.com
[3] https://github.com/tushartushar/DesigniteJava

Weighted Methods per Class	WMC	SumCyclomaticModified	Classe
Number of Children	NC	CountClassDerived	Classe
Lines Of Code	LOC	CountLine	Classe/Método
Cyclomatic Complexity	CC	Cyclomatic	Método

Tabela 3.1: Métricas utilizadas nas regras e nomenclatura utilizada na ferramenta *SciTools Understand*.

A tabela 3.1 mostra as regras catalogadas, as métricas utilizadas para cada *smell* e seus respectivos *thresholds*, valores que foram utilizados no experimento realizado por Sharma et al. (2019).

Nome	**Granularidade**	**Tipo**	**Métrica**	**Op. Lógico**
Multifaceted Abstraction	Classe	Design	LCOM => 0.8 NOF >= 7 NOM >= 7	E
Unnecessary Abstraction	Classe	Design	NOF >= 5 NOM == 0	E
Insufficient Modularization	Classe	Design	NOPM >= 20 NOM >= 30 WMC >= 100	OU
Wide Hierarchy	Classe	Design	NC >= 10	N/A
Long Method	Método	Impl.	LOC >= 100	N/A
Complex Method	Método	Impl.	CC >= 8	N/A

Tabela 3.2: Regras extraídas da ferramenta *Designite*.

3.3 Projetos selecionados

A tabela 3.3 mostra os projetos selecionados manualmente, 20 de cada linguagem, somando um total de 60 projetos, que foram selecionados observando as atividades recentes no repositório, número de *commits*, estrelas e *forks*.

Nome	**Linguagem**	***Commits***	**Estrelas**	***Forks***
terminal	cpp	2035	72168	6506
bitcoin	cpp	27843	50012	27804
okhttp	java	4815	39471	8416
imgui	cpp	6311	27888	4617
gson	java	1485	19223	3715

osquery	cpp	5831	17674	2109
PhotoView	java	462	17525	3806
ExoPlayer	java	10688	17322	5117
eShopOnContainers	csharp	3831	16790	7130
folly	cpp	9372	16359	3835
tdesktop	cpp	9148	14959	3217
faiss	cpp	472	12572	2146
mockito	java	5488	11540	2052
lombok	java	3136	10038	1913
flameshot	cpp	1260	10019	640
efcore	csharp	11098	9957	2491
ImHex	cpp	496	9830	435
keepassxc	cpp	3982	9731	788
jellyfin	csharp	20037	9724	997
Mindustry	java	11176	9333	1405
QuickLook	csharp	775	8085	650
AgentWeb	java	687	7878	1445
Files	csharp	1947	7363	439
termux-app	java	796	7261	1092
libzmq	cpp	8199	6793	1894
Signal-Server	java	673	5766	1406
Humanizer	csharp	2030	5357	749
monero	cpp	10184	5256	2431
tomcat	java	22962	5215	3554
Hazel	cpp	211	5025	762
junit5	java	6625	4455	1001
PX4-Autopilot	cpp	34735	4228	10792
ethminer	cpp	14334	4023	1400
xmrig	cpp	2751	3944	1825
StockSharp	csharp	7515	3777	1198
qTox	cpp	7800	3469	846
bisq	java	14106	3430	1106
Ryujinx	csharp	1536	3197	502
neo	csharp	1226	3147	923
reverse-proxy	csharp	359	3093	233
baritone	java	2675	2674	697
SharpZipLib	csharp	929	2554	772
gdal	cpp	43675	2355	1292
TelegramBots	java	849	2259	689

TwitchLeecher	csharp	311	1983	250
server	cpp	1904	1867	423
hudi	java	1375	1759	740
runner	csharp	325	1494	253
pinvoke	csharp	1621	1260	136
gitahead	cpp	351	1176	115
poi	java	10712	1127	499
copybara	java	2342	1067	170
omnisharp-roslyn	csharp	5310	1050	317
ModAssistant	csharp	594	1031	231
privatezilla	csharp	296	1006	56
msgpack-cli	csharp	3302	758	164
archiva	java	8659	275	111
catalyst	csharp	275	208	30
EnhancePoEApp	csharp	310	132	24
L2jOrg	java	985	37	35

Tabela 3.3: Projetos selecionados.

3.4 Métricas utilizadas para avaliação dos modelos

O problema de detecção de *code smells* pode ser tratado como um problema de classificação, onde o modelo treinado para um determinado *smell* deve classificar como "SIM" caso identifique o mesmo ou "NÃO" caso contrário.

Para computar as métricas é necessário construir a matriz de confusão onde os significados das siglas TP, TN, FP e FN, são:

- **TP:** *True Positive*, quando o modelo acerta a classe alvo "SIM".
- **TN:** *True Negative*: quando o modelo acerta a classe alvo "NÃO".
- **FP:** *False Positive*: quando o modelo erra a classe alvo, classificando como "SIM" mas na verdade era "NÃO".
- **FN:** *False Negative*: quando o modelo erra a classe alvo, classificando como "NÃO" mas na verdade era "SIM".

As métricas utilizadas para avaliar os modelos são:

- **Acurácia:** Taxa de classificação correta do modelo (como sendo um *smell* ou não);

$$A = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

- **Precisão:** Dos que o modelo classificou como sendo um *smell*, quantos realmente eram.

$$P = \frac{TP}{TP + FP} \tag{3.2}$$

- **Recall:** Quantia de classificados como *smell* do total de amostras que realmente eram *smells*.

$$R = \frac{TP}{TP + FN} \tag{3.3}$$

- **F1-score:** Média harmônica entre precisão e *recall*.

$$F = 2 \cdot \frac{P \cdot R}{P + R} \tag{3.4}$$

4

Resultados e Discussões

Neste capítulo, será apresentado os resultados obtidos através do experimento descrito no Capítulo 3 e discussões.

4.1 Conjuntos de dados do Modelo de Treinamento

Essa seção apresenta os dados da primeira questão de pesquisa **QP1. Como construir um conjunto de dados para treinar os modelos de detecção de *code smells* para diferentes linguagens de programação?**. Para responder essa questão nós construímos um conjunto de dados com *smells* detectados, esse conjunto de dados servirá para treinar os modelos de detecção de *code smells*. Dessa forma, nós computamos as mesmas métricas para classes e métodos em todos os projetos das linguagens selecionadas e construímos um conjunto de dados para cada *smell*, rotulando os fragmentos (classe ou método) com as regras catalogadas em 3.2 e seus respectivos *thresholds*. Nós obtivemos um conjuntos de dados com 22.687, 8.501 e 5.953 *smells* detectados em projetos das respectivas linguagens de programação, C++, Java e C#.

A Figura 4.1 mostra o número total de *smells* que foram detectados através das regras e *thresholds* definidos em 3.2 para projetos das linguagens de programação C++, Java e C#.

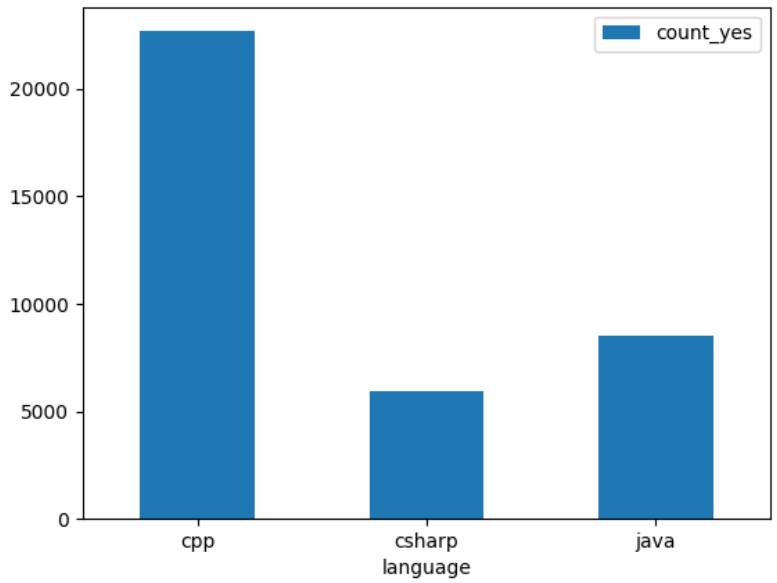

Figura 4.1: Número total de *code smells* por linguagem de programação

Como podemos observar, projetos na linguagem C++ possuem um número maior de *smells* detectados através das regras, este fato pode estar relacionado com as características dos projetos que são desenvolvidos utilizando a linguagem. A Figura 4.2 traz um detalhamento maior, mostrando quais tipos de *smells* tiveram uma maior ocorrência.

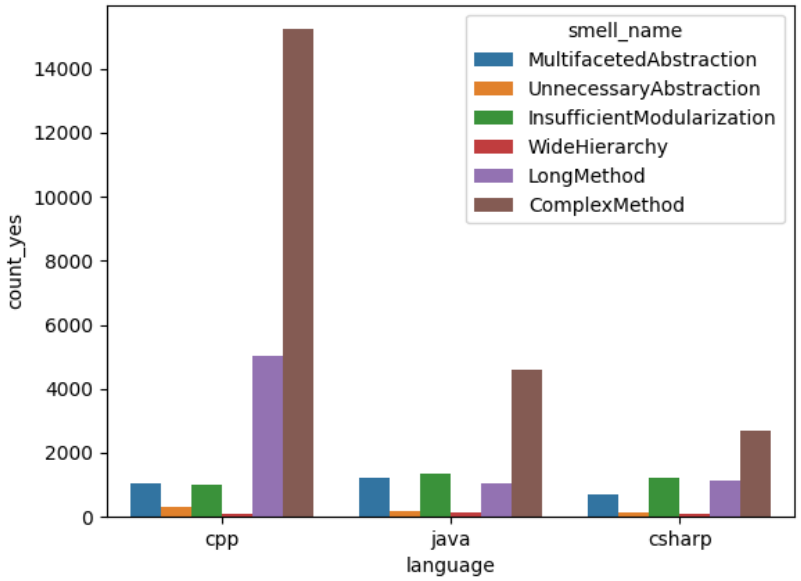

Figura 4.2: Número de cada *code smell* por linguagem de programação

Como observado na figura 4.2 foi possível construir um grande conjunto de dados contendo instâncias de *smells* de *design* e implementação, tendo sido detectados utilizando as mesmas regras para todos os projetos das linguagens C++, Java e C#. A linguagem C++ possui mais instâncias de *smells* relacionados a implementação, sendo eles *Complex Method* e *Long Method*.

Sumário da Questão de Pesquisa 1: Para treinar os modelos de detecção é necessário construir um conjunto de dados para treinamento. Para isso, é necessário os seguintes passos: (i) computar as mesmas métricas para classes e métodos em todos os projetos das linguagens selecionadas, (ii) coletar um conjunto de dados para cada *smell*, rotulando os fragmentos (classe ou método) com as regras catalogadas em 3.2 e seus respectivos *thresholds*.

4.2 Modelos de Treinamento usando Aprendizado por transferência

Essa seção apresenta os dados da segunda questão de pesquisa **QP2. Qual modelo pré-treinado escolher para realizar o aprendizado por transferência a depender do *smell* e linguagem alvo?**. Para responder essa pergunta, nós utilizamos 72 modelos que foram treinados utilizando os conjuntos de dados obtidos e descritos na seção 4.1, onde para cada *smell* em cada linguagem de programação foram treinados 4 modelos, um para cada algoritmo selecionado. O processo de aprendizagem por transferência consistiu em avaliar (através das métricas

definidas em 3.4) o modelo treinado para o *smell* na linguagem A no conjunto de dados da linguagem B.

As figuras 4.3, 4.4, e 4.5 apresentam as avaliações dos modelos pré-treinados para os *smells* de *design* utilizando a métrica f1-*score*.

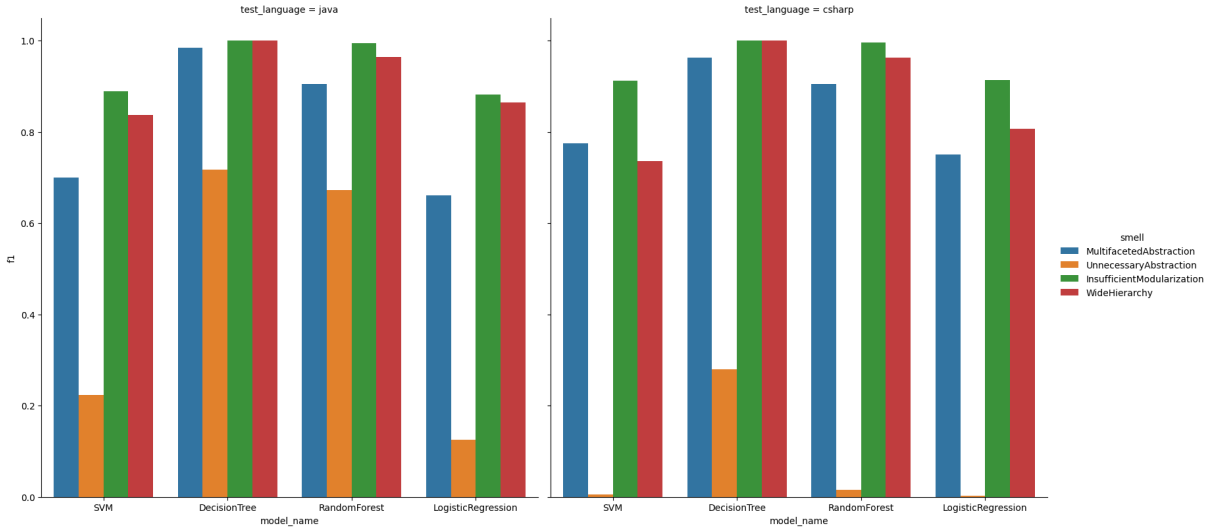

Figura 4.3: *Smells* de *Design*: f1-*score* para cada modelo treinado no conjunto de dados da linguagem C++ e avaliado nos das linguagens Java e C# respectivamente.

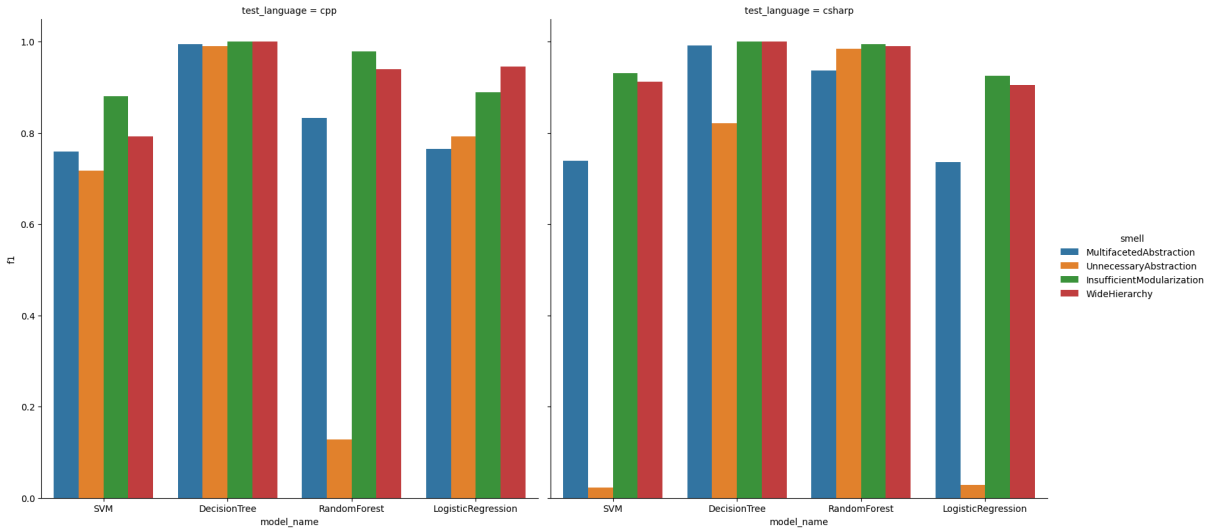

Figura 4.4: *Smells* de *Design*: f1-*score* para cada modelo treinado no conjunto de dados da linguagem Java e avaliado nos das linguagens C++ e C# respectivamente.

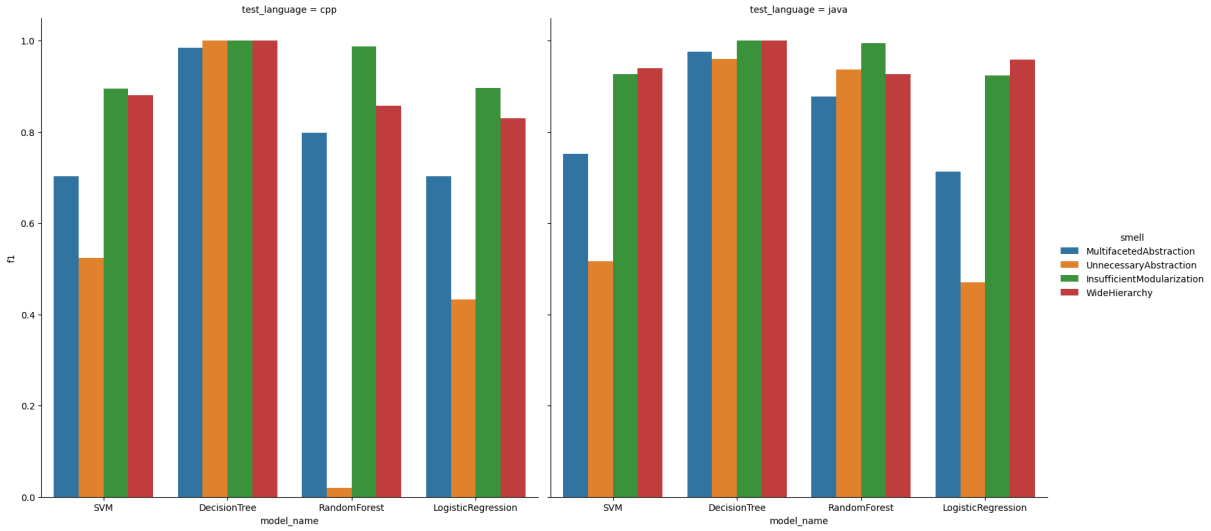

Figura 4.5: *Smells* de *Design*: f1-*score* para cada modelo treinado no conjunto de dados da linguagem C# e avaliado nos das linguagens C++ e Java respectivamente.

Com os resultados das figuras 4.3, 4.4 e 4.5, é possível escolher um modelo a depender do *smell* e linguagem alvo, por exemplo, suponhamos que o *smell* a ser detectado seja o *Multifaceted Abstraction* e a linguagem alvo seja Java, o modelo que obteve o melhor desempenho foi o que utilizou o algoritmo *DecisionTree* treinado com o conjunto de dados da linguagem C#. Já para outro *smell* de *design*, o *Unecessary Abstraction*, se a linguagem alvo for C# podemos escolher o modelo que utilizou o algoritmo *RandomForest* treinado no conjunto de dados da linguagem Java pois foi melhor dentre os outros modelos treinados no conjunto de dados da linguagem C++ para o mesmo *smell*.

As figuras 4.6, 4.7, e 4.8 apresentam as avaliações dos modelos pré-treinados para os *smells* de implementação utilizando a métrica f1-*score*.

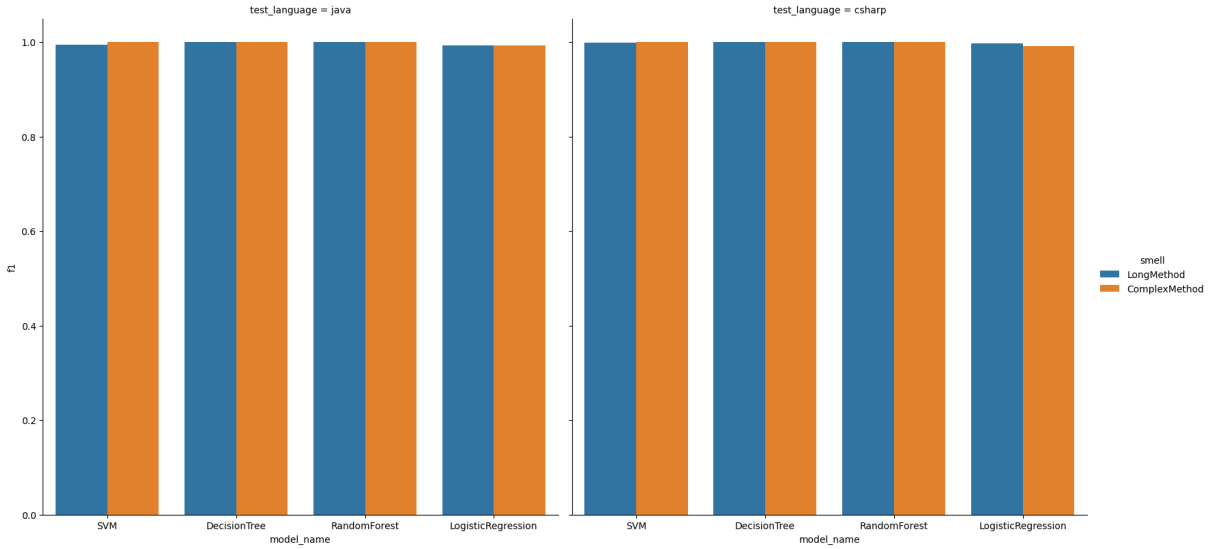

Figura 4.6: *Smells* de implementação: f1-*score* para cada modelo treinado no conjunto de dados da linguagem C++ e avaliado nos das linguagens Java e C# respectivamente.

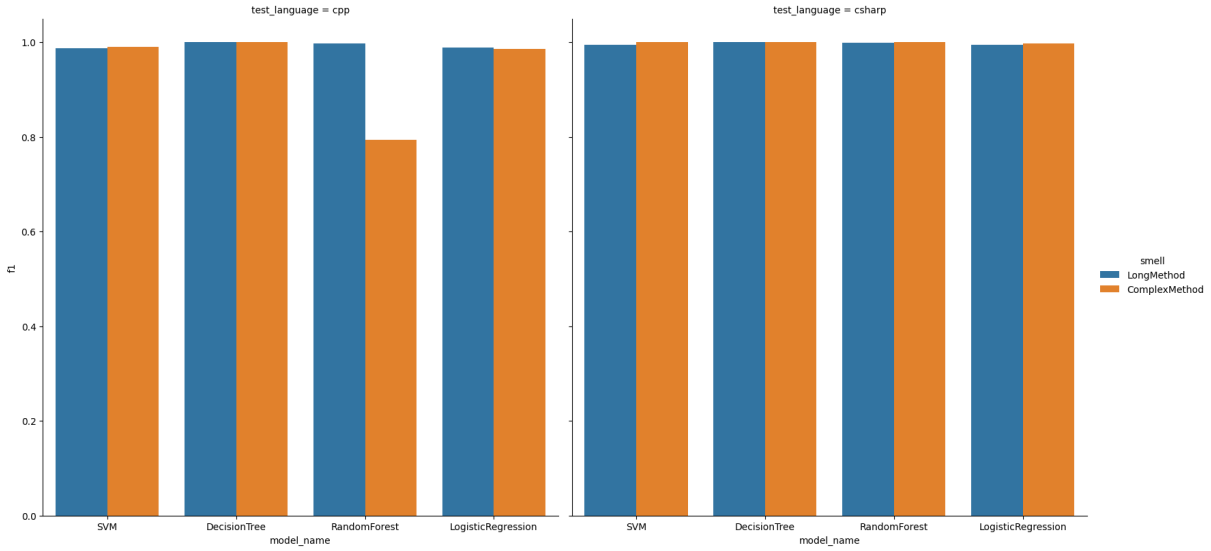

Figura 4.7: *Smells* de implementação: f1-*score* para cada modelo treinado no conjunto de dados da linguagem Java e avaliado nos das linguagens C++ e C# respectivamente.

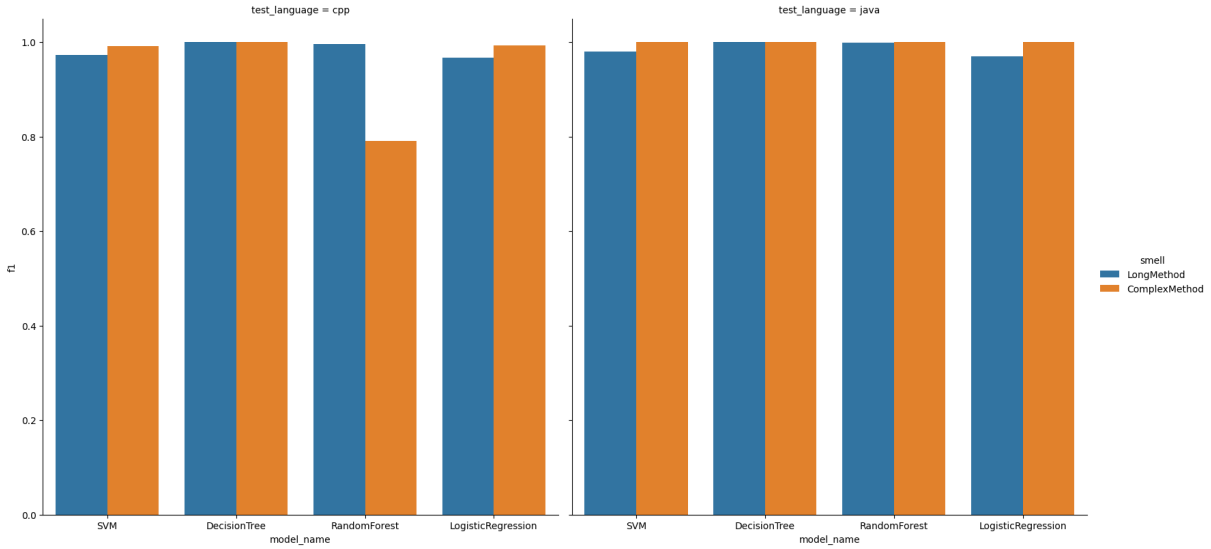

Figura 4.8: *Smells* de implementação f1-*score* para cada modelo treinado no conjunto de dados da linguagem C# e avaliado nos das linguagens C++ e Java respectivamente.

Os resultados obtidos mostram que houve uma variação maior entre os modelos relacionados a *smells* de *design*, o que pode indicar que alguns modelos são melhores para identificar certos tipos de *smells*. Por outro lado, os modelos relacionados a *smells* de implementação, cuja as regras de identificação para os selecionados são mais simples, não obtiveram um bom desempenho, os resultados não mostram uma variação expressiva.

Sumário da Questão de Pesquisa 2: O modelo que utilizou o algoritmo *DecisionTree* geralmente é o mais apropriado para detectar os *smells* de *design*, independente da linguagem de programação. Mas, para os *smells* de implementação, pode ser utilizado qualquer modelo avaliado, independente da linguagem de programação, pois não obtiveram diferenças significativas.

5

Conclusão

A proposta do presente trabalho foi investigar o uso do aprendizado por transferência no contexto de detecção de *code smells*, para isso foi necessário obter os modelos pré-treinados em projetos das linguagens C++, Java e C# para os *smells* de *design* e implementação escolhidos. Houve uma dificuldade para encontrar conjuntos de dados já prontos para a fase de treinamento e validação, a solução adotada foi construir um grande conjunto de dados para treinar e validar os modelos.

Foi possível construir um grande conjunto de dados para treinamento e validação dos modelos, utilizando as regras catalogadas e *thresholds* extraídos da ferramenta *Designite*, obtendo um total 22.687, 8.501 e 5.953 *smells* detectados em projetos das respectivas linguagens de programação, C++, Java e C#, mostrando que é possível detectar os *smells* em projetos das três linguagens utilizando a mesma regra. O número alto de *smells* na linguagem C++ se deu pela grande ocorrência dos *smells* de implementação: *Complex Method* e *Long Method*.

Após a obtenção dos 72 modelos pré-teinados, foi realizado o aprendizado por transferência, que consistiu em avaliar o modelo pré-treinado para o *smell* no conjunto de dados da linguagem A no da linguagem B, por exemplo, considerando o *smell* de *design*, *Unecessary Abstraction*, se a linguagem alvo for C# podemos escolher o modelo que utilizou o algoritmo *RandomForest* treinado no conjunto de dados da linguagem Java pois foi melhor dentre os outros modelos treinados no conjunto de dados da linguagem C++ para o mesmo *smell*.

As contribuições não se limitaram aos resultados do aprendizado por transferência, como consequência, foi desenvolvido uma ferramenta para detecção de *smells* baseado em métricas utilizando a ferramenta *SciTools Understand*, bem como os conjuntos de dados utilizados no experimento e os modelos pré-treinados serão disponibilizados para que outros pesquisadores possam utilizar em estudos futuros.

Todos os dados e códigos fontes utilizados estão disponíveis em `https://github.com/moabson/transfer-learning`.

5.1 Trabalhos futuros

- Realizar o balanceamento dos conjuntos de dados;
- Explorar o uso do aprendizado por transferência com o aprendizado profundo;
- Explorar mais tipos de *Code Smells*, linguagens de programação e projetos;
- Utilizar modelos pré-treinados de outros trabalhos.

Referências bibliográficas

Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: Are we there yet? *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings*, 2018-March:612–621, 2018. **DOI** 10.1109/SANER.2018.8330266.

Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. Automatic metric thresholds derivation for code smell detection. *International Workshop on Emerging Trends in Software Metrics, WETSoM*, 2015-Augus(October):44–53, 2015. ISSN 23270969. **DOI** 10.1109/WETSoM.2015.14.

M. Fowler, D.R.J.B.W.O.K.B. Martin Fowler, K. Beck, J.C. Shanklin, P. Becker, Addison-Wesley, E. Gamma, Safari Tech Books Online (Online service), J. Brant, W. Opdyke, et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley object technology series. Addison-Wesley, 1999. ISBN 9780201485677. URL https://books.google.com.br/books?id=1MsETFPD3I0C.

Huan Liang, Wenlong Fu, and Fengji Yi. A Survey of Recent Advances in Transfer Learning. *International Conference on Communication Technology Proceedings, ICCT*, pages 1516–1523, 2019. **DOI** 10.1109/ICCT46805.2019.8947072.

Xinghua Liu and Cheng Zhang. The detection of code smell on software development: a mapping study. 126(Icmmct):560–575, 2017. **DOI** 10.2991/icmmct-17.2017.120.

Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, 105:43–55, 2019. ISSN 09505849. **DOI** 10.1016/j.infsof.2018.08.004.

Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010. ISSN 10414347. **DOI** 10.1109/TKDE.2009.191.

Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. Comparing heuristic and machine learning approaches for metric-based code smell detection. *IEEE International Conference on Program Comprehension*, 2019-May:93–104, 2019. **DOI** 10.1109/ICPC.2019.00023.

Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158–173, 2018. ISSN 01641212. **DOI** 10.1016/j.jss.2017.12.034. URL https://doi.org/10.1016/j.jss.2017.12.034.

Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. Designite: a software design quality assessment tool. pages 1–4, 05 2016. **DOI** 10.1145/2896935.2896938.

Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. On the feasibility of transfer-learning code smells using deep learning. 04 2019.

Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, MTD '11, page 17–23, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305860. **DOI** 10.1145/1985362.1985366. URL https://doi.org/10.1145/1985362.1985366.