Uma Investigação da Aplicação de Aprendizado de Máquina para Detecção de Smells Arquiteturais

Warteruzannan Soyer Cunha[1], **Valter Vieira de Camargo**[1]

[1]Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
Rodovia Washington Luis, km 235 – 13565-905 - São Carlos – SP – Brazil

{warteruzannan.cunha,valter}@ufscar.br

Resumo. *Uma investigação da aplicação de aprendizado de máquina para detectar os smells arquiteturais God Component (GC) e Unstable Dependency (UD) é apresentada neste trabalho. Dois datasets foram criados com exemplos coletados de sistemas reais. A acurácia, precisão e recall foram avaliadas com um conjunto de 10 algoritmos preditivos. Uma seleção de atributos foi realizada a fim de encontrar os mais relevantes para essa detecção. Os algoritmos AdaBoost e SVM (Support Vector Machine) com kernel linear alcançaram os melhores resultados para o GC e UD, respectivamente. Além disso, observou-se que alguns atributos que a princípio não seriam considerados, contribuíram para a precisão da detecção.*

1. Introdução

Embora o conceito de *smells* tenha surgido originalmente no contexto de elementos de código-fonte com baixo nível de abstração (classes, métodos, entre outros), eles também ocorrem em níveis mais altos, como na arquitetura de sistemas. O termo *smell* arquitetural foi cunhado por [Lippert and Roock 2006] para descrever a combinação de decisões equivocadas em nível arquitetural que resultam na deterioração do software, reduzindo sua qualidade. [Garcia et al. 2009] relatam que *smells* desse tipo demandam mais esforço para serem identificados e corrigidos quando comparados a outros de nível mais baixo de abstração. Exemplos de dois *smells* reportados na literatura são o *Unstable Dependency* e o *God Component*. Considerando uma definição direta e simples, pode-se dizer que o UD ocorre quando um componente A depende de um outro B que é menos estável que o A; e o GC ocorre quando um componente tem um elevado número de classes, pacotes ou número de linhas e/ou implementa mais de uma funcionalidade.

Assim, a detecção de *smells* arquiteturais, principalmente com apoio automatizado, é uma é uma temática de significativa relevância, tendo em vista que problemas existentes em nível arquitetural podem ter impactos severos na qualidade do sistema e elevar os custos de manutenção. Portanto, pesquisas têm sido conduzidas a fim de identificar esse tipo de *smell*, apoiando-se em métricas [Fontana et al. 2017], regras definidas manualmente [Velasco-Elizondo et al. 2018] e histórico de versões [Palomba et al. 2013].

Neste trabalho, defende-se a ideia de que a caracterização de *smells* possui um nível de subjetividade envolvido e deve ser realizada a partir de diferentes pontos de vista, seja pessoas ou abordagens. Isto é, caracterizar algo como *smell* depende do contexto e experiência dos engenheiros de software que conhecem o sistema. Alguns autores relatam que trabalhos que fazem o uso de métricas e/ou regras que estabelecem thresholds para a detecção de *smells* (em qualquer nível de abstração) são propensos a erros e podem

apresentar uma quantidade elevada de falsos positivos/negativos, além de não levar em conta a subjetividade dessa tarefa [Wang et al. 2018].

Visto isso, a utilização de aprendizado de máquina (AM), uma subárea da Inteligência Artificial (IA), tem sido explorada por alguns pesquisadores a fim de reduzir a quantidade de erros, melhorar a acurácia na identificação de *code smells* e incluir, de alguma forma, uma análise subjetiva sobre essa decisão. Com isso, o emprego de AM consiste em coletar exemplos de diferentes pontos de vista, generalizar esse conhecimento e classificar entradas desconhecidas [Wang et al. 2018, Moha et al. 2010]. Contudo, poucos pesquisadores têm explorado a utilização dessa técnica na detecção de *smells* arquiteturais. Até o presente momento, trabalhos ou ferramentas que identificam o UD (*Unstable Dependency*) e GC (*God Component*) utilizando AM não foram encontrados, bem como estudos que relatam quais os melhores algoritmos nesse contexto. Além disso, não foram encontrados relatos de quais seriam os atributos mais relevantes para a detecção desses *smells* utilizando AM.

Portanto, o presente trabalho busca investigar o emprego de AM na identificação de dois *smells* (GC e/ou UD). As questões de pesquisa que norteiam este trabalho são:

#QP1: Há indícios de que o aprendizado de máquina pode melhorar a acurácia da identificação dos *smells* abordados?

#QP2: Quais são os atributos mais relevantes para identificação dos *smells* abordados neste trabalho?

Como resultados, observou-se que a utilização de AM ofereceu bons resultados. O *AdaBoost* foi o melhor algoritmo para GC e, paralelamente, o SVM (*Support Vector Machine*) com kernel linear para o segundo para o UD, com acurácia de 99,17% e 99,41%, respectivamente. Além disso, foi observado que atributos (Figura 4) que não foram estudados anteriormente contribuem para essa identificação.

2. Fundamentação Teórica

Unstable Dependency: Esse *smell* ocorre quando um elemento arquitetural depende de outro menos estável que ele mesmo [Fontana et al. 2017]. Isso prejudica a manutenção do sistema, uma vez que componentes instáveis são mais suscetíveis a manutenções e erros. Geralmente, a identificação por métricas compara os valores da instabilidade de dois componentes. Entretanto, os resultados podem variar, uma vez que os engenheiros de software podem ter opiniões divergentes sobre instabilidade. Portanto, é necessário detectar esse *smell* sobre diferentes pontos de vista. O conceito de instabilidade é melhor discutido por [Martin 1994].

Too Large Packages/Subsystems (God Component): Analogamente ao *smell God Class*, *God Component* surge quando um elemento arquitetural é muito grande, tem muitas classes, interfaces, arquivos, linhas de código e/ou sub-elementos (pacotes) e/ou duas ou mais funcionalidade estão sendo implementadas, o que prejudica a coesão, manutenção, testabilidade e evolução do software [Lippert and Roock 2006].

Aprendizado de Máquina: O Aprendizado de Máquina (AM) é definido como uma subárea da Inteligência Artificial (IA), cujo o objetivo de criar e melhorar técnicas para a construção de sistemas capazes de tomar decisões baseadas em experiências acumuladas. Assim, os algoritmos de AM conseguem aprender com exemplos e deduzir uma

função (também conhecida como modelo, função objetivo, hipótese, classificador, entre outras) capaz de analisar entradas desconhecidas e rotulá-las de acordo com o conjunto esperado. No geral, o rótulo é atribuído com base nas características de cada instância [Mitchell 1997]. O conjunto de exemplos é conhecido como base de dados ou *dataset* e o rótulo como classe. Neste trabalho as funções deduzidas são chamadas de classificadores.

Validação cruzada: A validação cruzada é uma técnica de AM que consiste em dividir o *dataset* em *k* subconjuntos com a mesma quantidade de exemplos, de forma que um desses é utilizado para teste e o restante para treinamento. Assim, o subconjunto *k-1* é utilizado para teste, depois o *k-2*, e assim sucessivamente, até que todos os exemplos sejam testados [Kohavi et al. 1995].

A Equação 1 ilustra o cálculo da acurácia, na qual *tp* (*true positive*) são os verdadeiros positivos, *tn* (*true negative*) os verdadeiros negativos, *fn* (*false negative*) falso negativos e *fp* (*false positive*) os falsos positivos. A acurácia oferece, no geral, o quanto o classificador está classificando corretemente entradas desconhecidas. Na Equação 2 é apresentado o cálculo da precisão, que permite conhecer o quanto, dos exemplos classificados como verdadeiros, realmente são. Na Equação 3 o cálculo do recall, que permite saber qual a frequência com que exemplos de uma determinada classe são classificados corretamente. A médias dessas métricas podem ser mensuradas cada vez que o classificador é treinado e testado

$$(1)\ Accuracy = \frac{tp + tn}{tp + tn + fn + fp} \qquad (2)\ Precision = \frac{tp}{tp + fp} \qquad (3)\ Recall = \frac{tp}{tp + fn}$$

3. Metodologia

A metodologia para desenvolvimento deste trabalho consistiu nas seis atividades descritas abaixo.

#1 Selecionar Sistemas: Foram selecionados sistemas que atendiam aos seguintes critérios: i) ser desenvolvido em Java [1]; e ii) ser de código aberto; e iii) pertencer a uma das seguintes categorias: mobile, ferramentas, plugins, bibliotecas/utilitários, frameworks, *API* e *databases*. Esse último critério foi aplicado a fim de coletar exemplos de *smells* em tipos variados de sistemas. Foram selecionados 5 sistemas mobile, 9 ferramentas, 4 plugins, 23 bibliotecas/utilitários, 6 frameworks, 3 *apis*, 3 *databases* e 1 *middleware*. A busca foi realiza em repositórios do github e/ou repositórios próprios. Ao final, 52 sistemas foram selecionados. A lista completa pode ser encontrada no link https://gist.github.com/papersfiles/a9a4035fb4acdf1ef0b80459941a2fd.

#2 Definir atributos dos Datasets: Consistiu na escolha de um conjunto de métricas que foram utilizadas como os atributos dos *datasets*. A escolha seguiu os seguintes critérios: C1) Métricas utilizadas por outros autores na detecção de *smells* arquiteturais [Fontana et al. 2017, Martin 1994, Kumar and Sureka 2018, Sharma 2016]; e C2) Métricas que caracterizam complexidade de código, como coesão, acoplamento e tamanho. As métricas do segundo critério foram adicionadas justamente para averiguar se elas também possuem alguma influência na decisão. O conjunto escolhido pode ser visualizado no link https://github.com/papersfiles/artigo-vem/blob/master/metrics.

[1] https://www.java.com/en/download/

#3 Extrair valores das métricas escolhidas dos componentes de sistemas selecionados: Consistiu em analisar todos componentes dos sistemas selecionados e extrair os valores das métricas escolhidas anteriormente. Assim, os valores extraídos foram colocados em colunas em dois *datasets*. Para o GC, cada linha (exemplo) do *dataset* representa um componente ao passo que, para o UD, cada linda representa uma dependência entre dois componente. Portanto, a quantidade de atributos previsores do UD é duas vezes maior que a do GC, já que dois componentes são representados em cada linha.

#4 Rotular exemplos dos *datasets*: Consistiu em atribuir rótulos 0 (não é smell) ou 1 (é smell) para cada componente (no caso do GC) ou dependência (no caso do UD). Nesta etapa, a atribuição dos rótulos foi feita usando abordagens existentes. Cada entrada do *dataset* foi avaliada por mais de uma abordagem. Logo, o rótulo foi atribuído de acordo com a maioria das abordagens. Por exemplo, se três relataram a ocorrência do *smell* e duas que não, o rótulo foi 1.

#5 Balancear classes e normalizar os dados: Consistiu em balancear a quantidade de exemplos positivos e negativos para cada *smell* e normalizar[2] a escala dos valores extraídos. Isso evita que os classificadores considerem atributos com valores elevados mais importantes, o que prejudica o aprendizado.

#6 Treinar e avaliar classificadores: Concentrou-se em: i) Escolher um conjunto de classificadores; ii) Separar cada *dataset* em duas partes, uma para treinamento e outra para teste; iii) Treinar os classificadores com os dados de treino e avaliá-los de acordo com a acurácia, precisão e recall com os dados de teste. Além disso, avaliar estatisticamente quais alcançaram os melhores resultados. Utilizou-se os seguintes classificadores: *Naïve Bayes, Decision Tree, Random Forest, Knn, AdaBoost, Support Vector Machine, Redes Neurais Artificiais, XGBoost, Bagging e Dummy Classifier*.

4. Resultados e Discussões

Os parágrafos abaixo são voltados a responder as questões de pesquisa elencadas para este trabalho. Para facilitar o entendimento, segue uma explicação sobre o processo de teste simples: Uma vez que todos os exemplos dos *datasets* já foram rotulados com abordagens existentes, isto é, sob diferentes pontos de vista, parte dos dados é separada para treinamento dos classificadores e outra para teste. Logo, os classificadores são treinados somente com os dados destinados a treino, assim, não conhecem o rótulo de exemplos presentes na porção de teste. Em seguida, cada exemplo em teste é submetido ao classificador já treinado e um rótulo é retornado por esse. Com isso, é possível verificar se o classificador acertou ou não, já que rótulo dessa entrada já é conhecido. Esse processo possibilita calcular métricas como acurácia, precisão e recall. Vale ressaltar que os testes foram realizados utilizando uma técnica de validação cruzada com k=10 repetida 30 vezes. Essa técnica é amplamente aceita pela comunidade de AM e é melhor explicada na seção 2.

#QP1: Há indícios de que o aprendizado de máquina pode melhorar a acurácia quanto à identificação dos *smells* abordados? Na Tabela 1 (a) podem ser visualizadas a média da acurácia (coluna 2), precisão (coluna 3) e recall (coluna 4) dos testes realizados com o *dataset* do UD, de acordo com cada classificador (coluna 1). O

[2]https://scikit-learn.org/stable/modules/preprocessing.html

XGBoost alcançou 98,97% de precisão, o *SVM com kernel linear* alcançou 99,88% de recall e 99,41% de acurácia. Visto que o *dataset* foi oi rotulado com base em diferentes abordagens existentes, os classificadores conseguiram aprender com esses exemplos e fornecer bons resultados, alguns com acurácia acima de 99%. Entretanto, a métrica (precisão, recall, etc) que mede a eficiência do classificador deve ser escolhida com cuidado. Por exemplo, o classificador *KNN* obteve 92,23% de acurácia, muito próximo ao SVM - *Polynomial*. Entretanto, a diferença da precisão é significativa, portanto, o classificador escolhido deve ter um equilíbrio entre essas métricas.

De forma similar, os resultados para o *dataset* do GC podem ser vistos na na Tabela 1 (b). O AdaBoost alcançou a melhor acurácia e precisão, respectivamente, 99,17% e 98,84%. Por outro lado, SVM com kernel linear apresentou recall de 100%. Notou-se que alguns têm boa acurácia e precisão, porém baixo recall, a exemplo, SVM - Polynomial e Sigmoid. Isso indica que mesmo que a acurácia seja boa, a taxa de erro na identificação das classes positivas e negativa é alta. O *dataset* construído para o GC permite identificar, com alta precisão, componentes com alto custo de manutenção e direcionar os esforços de manutenção. Além disso, esses dados podem ser estendidos para atribuir a grau de severidade do *smell* encontrado; um algoritmo de regressão pode ser utilizado.

Tabela 1 (a) - Acurácia, precisão e revocação do *unstable dependency*

Algoritmo	Acurácia	Precisão	Revocação
Knn	0.9223	0.8685	0.9954
Random Forest - entropy	0.9507	0.9243	0.9819
Random Forest - gini	0.9517	0.9279	0.9795
Redes Neurais Artificiais	0.9908	0.9849	0.9969
Naive Bayes	0.6947	0.6277	0.9569
SVM - Linear	0.9941	0.9896	0.9988
SVM - Polynomial	0.9399	0.9189	0.9651
SVM - Sigmoid	0.8671	0.8543	0.8853
SVM - RBF	0.9785	0.9623	0.9959
AdaBoost	0.9904	0.9854	0.9955
Decision tree - entropy	0.9867	0.9782	0.9955
Decision Tree - gini	0.9824	0.9769	0.9882
XGBoost	0.9938	0.9897	0.9980
Bagging	0.9912	0.9896	0.9929
Dummy - stratified	0.4972	0.4973	0.4978

Tabela 1 (b) - Acurácia, precisão e revocação do *god component*

Algoritmo	Acurácia	Precisão	Revocação
Knn	0.9512	0.9118	0.9991
Random Forest - entropy	0.9795	0.9640	0.9962
Random Forest - gini	0.9772	0.9597	0.9961
Redes Neurais Artificiais	0.9879	0.9767	0.9996
Naive Bayes	0.8416	0.9282	0.7408
SVM - Linear	0.9826	0.9664	1.0000
SVM - Polynomial	0.8973	0.9750	0.8157
SVM - Sigmoid	0.8438	0.8321	0.8615
SVM - RBF	0.9675	0.9458	0.9918
AdaBoost	0.9917	0.9884	0.9951
Decision tree - entropy	0.9878	0.9868	0.9888
Decision Tree - gini	0.9707	0.9586	0.9841
XGBoost	0.9899	0.9849	0.9952
Bagging	0.9836	0.9751	0.9927
Dummy - stratified	0.5007	0.5009	0.4987

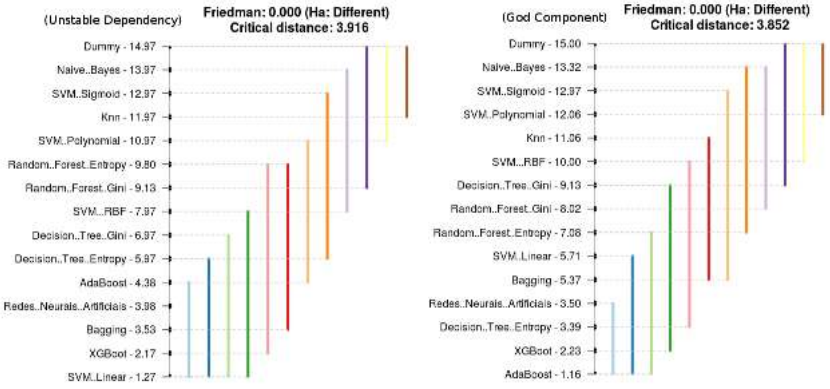
Para complementar a análise, o teste de Friedman foi aplicado a fim de verificar se os classificadores apresentam resultados estatisticamente iguais. O resultado foi igual a 0.00, confirmando que há diferença significativa. Logo, o teste Nemeneiy foi realizado pois permite verificar qual a diferença significativa entre esses resultado e, por conseguinte, descartar a utilização de classificadores que apresentam boa acurácia mas são, estatisticamente, inferiores a outros.

Como pode ser visualizado na Figura 1 (unstable dependency), o valor critico é de 3.916, isto é, caso a diferença entre classificadores fosse menor que esse valor, seriam considerados equivalentes. Por exemplo, a distância entre o SMV - Linear e XGBoost é de 0.9 (1.17 - 1.27), menor que 3.916. Logo, o SVM com kernel linear e o XGBoost podem ser utilizados de forma equivalentes neste contexto.

Para o GC, o SVM com kernel linear alcançou, em média, a acurácia de 98.26%, o que da a falsa sensação de que ele é um dos mais indicados nesse contexto. Entretanto, com o teste de Nemeneiy pode ser verificado que outros como o AdaBoost, XGBoost, *Decicion Tree - Entropy* e Redes Neurais Artificiais apresentam melhores resultados.

#RQP1 Portanto, há indícios de que o aprendizado de máquina pode melhorar a acurácia quanto à identificação dos *smells* abordados no presente trabalho. Uma vez os *datasets* foram construídos com base em outros trabalhos e ferramentas, ou seja, com

Figura 1. Teste de Nemeneiy para com o *Unstable dependency* e *God Component*



base em diferentes visões, os algoritmos de AM conseguiriam generalizar o conhecimento extraído e alcançar bons resultados quanto à acurácia, precisão e recall.

#QP2: Quais são os atributos mais relevantes para identificação dos *smells* abordados neste trabalho?

A seleção de atributos foi realizada utilizando o modelo *Extra Trees Classifier* [3], implementado na biblioteca SckitLearn [4]. Esse modelo permite atribuir a cada atributo um *score* de importância. Vale ressaltar que o nome de cada métrica pode ser visto em https://github.com/papersfiles/artigo-vem/blob/master/metrics juntamente com seu acrônimo, bem como os sufixos *max, min, std, sum, q1* e *q2* significam, respectivamente, máximo, mínimo, desvio padrão, soma, primeiro e segundo quartil. O prefixo *pck2* indica que o valor desse atributo foi extraído do componente aferente.

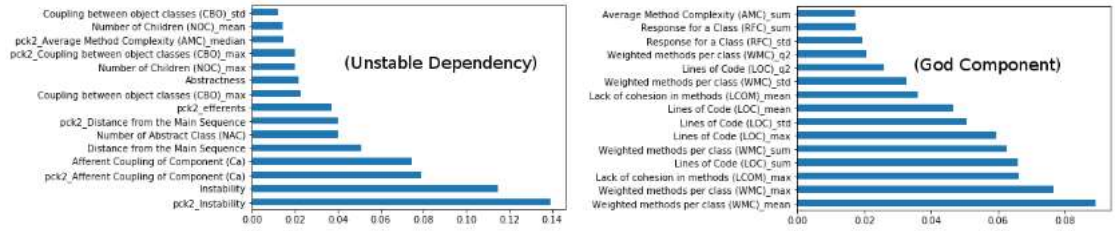
#RQP2: Para facilitar a visualização, apenas os 15 atributos [5] mais importantes para a detecção do *unstable dependency* são apresentados na Figura 2 do lado esquerdo. É importante notar que, atributos que não foram utilizados no processo de rotulação aparecem entre aqueles mais relevantes para a identificação desse *smell*, por exemplo CBO (*Coupling between object classes*) e AMC (*Average Method Complexity*).

Os atributos mais importantes [6] para a detecção do *god component* podem ser visualizados na Figura 2 do lado direito. Destaca-se a importância de alguns que não foram utilizados para a rotular exemplos do *dataset*, por exemplo, RFC (*Response for a Class*). Os três atributos mais relevantes foram *WMC_mean*, *WMC_max* e *LCOM_max*, estão relacionados à complexidade e falta de coesão, assim, é importante que o engenheiro de software fique atento à essas métricas.

Como ameaça à validade, é importante ressaltar que, a fim de remover o viés inserido pela forma de rotulação, os classificadores também foram testados sem a presença do conjunto de atributos utilizados na rotulação, assim, pôde ser verificado se o aprendizado extraído é consequência da forma de rotulação ou os classificadores realmente estavam aprendendo.

[3]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html
[4]https://scikit-learn.org/stable/index.html
[5]https://github.com/papersfiles/artigo-vem/blob/master/features_unstable_dependency.csv
[6]https://github.com/papersfiles/artigo-vem/blob/master/features_god_component.csv

Figura 2. Importância dos Atributos na Detecção dos Smells



5. Trabalhos correlatos

[Wang et al. 2018] apresentam uma estratégia para rotular os exemplos de *code smells*. A abordagem é restrita a *code smells* e não explora a identificação em nível arquitetural. [Garcia et al. 2009] realizaram um estudo para definir a taxonomia e identificação de 04 (quatro) *smells* arquiteturais. Porém, é uma abordagem totalmente manual, que prejudica a acurácia identificação. [Fontana et al. 2016] avaliam um conjunto de algoritmos para a detecção de *code smells*, a saber: *J48, JRip, Random Forest, Naive Bayes, SMO* e *LibSVM*. Entretanto concentra-se em *code smells* e não exploram a detecção em nível arquitetural. [Fontana et al. 2017] apresentam uma ferramenta nomeada *Arcan* com o objetivo de detectar três *smells* arquiteturais, a saber: i) *Cyclic Dependency, Unstable Dependency* e *Hub-Like Dependency (HL)*. A ferramenta não explora o uso de AM para essa tarefa. [Maiga et al. 2012] apresentam uma abordagem que busca identificar *code smells* utilizando aprendizado de máquina Entretanto concentra-se em *code smells* e não explora a identificação em nível arquitetural.

6. Considerações Finais

Este trabalho apresentou uma investigação do emprego de AM para a identificação de dois *smells* arquiteturais: *god component* e *unstable dependency*. Os atributos mais relevantes para a identificação foram relatados, bem como a descrição de classificadores que apresentam bons resultados nesse contexto. Os melhores classificadores para a identificação para o GC e UD foram o AdaBoost e SVM com kernel linear, respectivamente.

A base de dados construída neste trabalho pode servir como apoio para ferramentas que, automaticamente, analisam dependências de terceiros (SDKs, bibliotecas, etc) incorporadas no projeto e relatam ao engenheiro de software sobre uma possível necessidade de refatoração. Além disso, o *dataset* pode ser estendido para identificar outros *smells* que ocorrem na dependência entre elementos arquiteturais, por exemplo o *Hub Like Dependency* [Fontana et al. 2017]. Ademais, este trabalho pode servir de base para estudos que comparem diferentes estratégias de rotulação do UD (ou GC), uma vez que este apoiou-se em abordagens existentes ao invés de desenvolvedores

Como trabalhos futuros, pretende-se avaliar a detecção de outros *smells* arquiteturais como: *ambiguous interfaces, feature concentration* e *unstable interface*. Outra importante contribuição é a utilização de componentes arquiteturais representados como classes, interfaces, entre outros, não restringindo somente a pacotes.

Referências

[Fontana et al. 2016] Fontana, F. A., Mäntylä, M. V., Zanoni, M., and Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection.

Empirical Software Engineering, pages 1143–1191.

[Fontana et al. 2017] Fontana, F. A., Pigazzini, I., Roveda, R., Tamburri, D., Zanoni, M., and Di Nitto, E. (2017). Arcan: A tool for architectural smells detection. In *2017 IEEE International Conf. on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5–7*.

[Garcia et al. 2009] Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. (2009). Toward a catalogue of architectural bad smells. In *International Conference on the Quality of Software Architectures*, pages 146–162. Springer.

[Kohavi et al. 1995] Kohavi, R. et al. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada.

[Kumar and Sureka 2018] Kumar, L. and Sureka, A. (2018). An Empirical Analysis on Web Service Anti-pattern Detection Using a Machine Learning Framework. *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pages 2–11.

[Lippert and Roock 2006] Lippert, M. and Roock, S. (2006). *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons.

[Maiga et al. 2012] Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Gueheneuc, Y.-G., and Aimeur, E. (2012). Smurf: A svm-based incremental anti-pattern detection approach. In *Reverse engineering (WCRE), 2012 19th working conference on*, pages 466–475. IEEE.

[Martin 1994] Martin, R. (1994). Oo design quality metrics. *An analysis of dependencies*, pages 151–170.

[Mitchell 1997] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

[Moha et al. 2010] Moha, N., Gueheneuc, Y. G., Duchien, L., and Meur, A. F. L. (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, pages 20–36.

[Palomba et al. 2013] Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Lucia, A. D., and Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278.

[Sharma 2016] Sharma, T. (2016). Designite – A Software Design Quality Assessment Tool.

[Velasco-Elizondo et al. 2018] Velasco-Elizondo, P., Castañeda-Calvillo, L., García-Fernandez, A., and Vazquez-Reyes, S. (2018). Towards detecting mvc architectural smells. *Advances in Intelligent Systems and Computing*, 688:251–260. cited By 0.

[Wang et al. 2018] Wang, Y., Hu, S., Yin, L., and Zhou, X. (2018). Using code evolution information to improve the quality of labels in code smell datasets. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*.