



UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

**UMA ABORDAGEM BASEADA EM INCORPORAÇÃO DE
PALAVRAS PARA A REMODULARIZAÇÃO DE SOFTWARE
DESENVOLVIDO EM JAVA**

ROBERTO JOSÉ RODRIGUES

Orientador
Márcio de Oliveira Barros

RIO DE JANEIRO, RJ-BRASIL
FEVEREIRO DE 2025

UMA ABORDAGEM BASEADA EM INCORPORAÇÃO DE PALAVRAS PARA
A REMODULARIZAÇÃO DE SOFTWARE DESENVOLVIDO EM JAVA

ROBERTO JOSÉ RODRIGUES

TESE APRESENTADA COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE DOUTOR PELO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO (UNIRIO). APROVADA PELA COMISSÃO EXAMINADORA ABAIXO ASSINADA.

Aprovada por:

Documento assinado digitalmente
 **MARCIO DE OLIVEIRA BARROS**
Data: 07/03/2025 08:12:09-0300
Verifique em <https://validar.iti.gov.br>

Márcio de Oliveira Barros, D.Sc. - UNIRIO

Documento assinado digitalmente
 **GUILHERME HORTA TRAVASSOS**
Data: 01/03/2025 12:37:53-0300
Verifique em <https://validar.iti.gov.br>

Guilherme Horta Travassos, D.Sc. - COPPE/UFRJ

Documento assinado digitalmente
 **REGINA MARIA MACIEL BRAGA VILLELA**
Data: 06/03/2025 09:18:37-0300
Verifique em <https://validar.iti.gov.br>

Regina Maria Maciel Braga Villela, D.Sc. - UFJF

Documento assinado digitalmente
 **Adriana Cesario de Faria Alvim**
Data: 06/03/2025 14:56:48-0300
Verifique em <https://validar.iti.gov.br>

Adriana Cesario de Faria Alvim, D.Sc. - UNIRIO

Documento assinado digitalmente
 **CARLOS EDUARDO RIBEIRO DE MELLO**
Data: 14/03/2025 10:05:00-0300
Verifique em <https://validar.iti.gov.br>

Carlos Eduardo Ribeiro de Mello, D.Sc. - UNIRIO

RIO DE JANEIRO, RJ - BRASIL

FEVEREIRO de 2025

Catálogo informatizada pelo(a) autor(a)

R696 Rodrigues, Roberto José
 Uma Abordagem Baseada em Incorporação de Palavras para
a Remodularização de Software Desenvolvido em Java /
Roberto José Rodrigues. -- Rio de Janeiro : UNIRIO, 2025.
 136 f

 Orientador: Márcio de Oliveira Barros.
 Tese (Doutorado) - Universidade Federal do Estado do
Rio de Janeiro, Programa de Pós-Graduação em Informática,
2025.

 1. Processamento de linguagem natural. 2. Modelos de
incorporação de palavras . 3. Remodularização de software.
I. Barros, Márcio de Oliveira , orient. II. Título.

RESUMO

Rodrigues, Roberto José. **Uma Abordagem Baseada em Incorporação de Palavras para a Remodularização de Software Desenvolvido em Java**. FEVEREIRO DE 2025. 136 f. Pesquisa para qualificação ao curso de Doutorado em Informática - PPGI, Universidade Federal do Estado do Rio de Janeiro (UNIRIO), RIO DE JANEIRO, RJ-BRASIL, FEVEREIRO DE 2025.

Sistemas de software são compostos por módulos interligados, cuja organização afeta diretamente a sua manutenção. Estudos apontam que distribuições inadequadas de módulos aumentam a ocorrência de falhas. A literatura apresenta duas abordagens principais para a remodularização de software: a estrutural, baseada nas dependências entre módulos, e a léxica, que utiliza informações textuais para identificar subsistemas.

Esta pesquisa busca avaliar se técnicas de processamento de linguagem natural, especialmente modelos de incorporação de palavras, podem gerar distribuições de classes em pacotes semelhantes às organizadas pelos desenvolvedores de software e às obtidas por métodos estruturais. Experimentos iniciais com sistemas de código aberto mostraram que os modelos testados não foram eficazes para essa tarefa.

A solução proposta envolve o desenvolvimento de um *corpus* específico para treinar um modelo de incorporação de palavras voltado para a remodularização, utilizando aprendizado por transferência a partir de modelos já existentes. Em seguida, projetos de grandes organizações, como a *Apache Foundation*, serão analisados para identificar padrões nos termos utilizados em diferentes partes do código e aprimorar o *corpus*.

Foram realizados experimentos para comparar algoritmos baseados na estrutura do código e outros baseados em informações léxicas com diferentes configurações, considerando variações nos nomes de classes e pacotes. O algoritmo de agrupamento aglomerativo foi o mais adequado. A melhor configuração envolveu a análise por parágrafos, remoção de *stop-words* e eliminação de palavras repetidas. O modelo *Word2Vec* apresentou melhor desempenho em projetos grandes.

O modelo pré-treinado foi, no entanto, superado por um modelo híbrido, sugerindo que a incorporação de palavras pode ajudar a aproximar a distribuição de classes em pacotes proposta pelos desenvolvedores. A pesquisa destaca que bons identificadores no código melhoram a qualidade dos agrupamentos e que algoritmos de remodularização podem ser aprimorados com informações léxicas, fornecendo um *corpus* específico e análises estratégicas para a organização do software.

Palavras-chave: Processamento de linguagem natural, Modelos de incorporação de palavras, Remodularização de software.

ABSTRACT

A Word Embedding-Based Approach for Remodularizing Java Software

Rodrigues, Roberto José. **Uma Abordagem Baseada em Incorporação de Palavras para a Remodularização de Software Desenvolvido em Java**. FEVEREIRO DE 2025. 136 f. Dissertação (Mestrado em Informática) - PPGI, Universidade Federal do Estado do Rio de Janeiro (UNIRIO), RIO DE JANEIRO, RJ-BRASIL, FEVEREIRO DE 2025.

Software systems consist of interrelated modules and, as such systems grow, their design affects maintenance. Studies show a strong correlation between poor module distribution and software failures. The literature highlights two main approaches to software remodularization: structural, which accounts for the dependencies between modules, and lexical, which analyzes textual information to identify subsystems.

This research aims to determine whether natural language processing (NLP) techniques, particularly word embedding models, can generate class distributions that align with those defined by developers and structural remodularization. Initial experiments using open-source systems revealed that the former models did not distribute classes into packages.

The proposed solution involves developing a specific corpus to train a word embedding model for lexical-based software remodularization. We apply transfer learning from existing models and later analyze additional software projects to refine the corpus by considering terms from different areas of the source code. Large software projects from well-known software development organizations, such as Apache Foundation, are used to explore different word embedding models.

Experiments compared structural and lexical algorithms, evaluating variations in class distributions across different modalities. Results indicated that the agglomerative algorithm was the most effective for clustering. The most suitable configuration involved analyzing paragraphs, removing stop-words, and discarding repeated words. The Word2Vec algorithm showed superior performance for large projects. Additionally, a pre-trained model demonstrated better efficiency in most experiments, though a hybrid model outperformed it in some cases, suggesting that word embedding can improve class distribution in packages.

The study concludes that good code identifiers enhance clustering quality, reinforcing the importance of lexical-based remodularization strategies. The research contributes by providing a specific corpus, improved clustering algorithms, and new insights into remodularization techniques.

Palavras-chave: Natural language processing, Word embedding models, Software remodularization..

LISTA DE FIGURAS

2.1	Arquitetura <i>CBOW/SKIP-GRAM</i> . Imagem retirada de [84]	16
3.1	Dendrograma do agrupamento hierárquico para o software <i>joda-money</i> , mostrando um corte em dois grupos.	50
3.2	Identificação dos termos léxicos para classes convencionais nas modalidades “ <i>single</i> ”, “ <i>complete</i> ” e “ <i>full</i> ”.	54
3.3	Identificação dos termos léxicos para classes internas nas modalidades “ <i>single</i> ”, “ <i>complete</i> ” e “ <i>full</i> ”.	54
3.4	Exemplos de palavras malformadas gerando palavras ignoradas no processo.	63
4.1	Um exemplo de um arquivo CSV para o projeto <code>org.apache.anakia</code>	84
4.2	Texto exemplo para cálculo de similaridades	87
4.3	Um <i>boxplot</i> dos valores de MoJoFM produzidos pelos três algoritmos para a configuração PAR-REMV-UNIQ.	90
4.4	Média, mínimo e máximo do <i>MoJoFM</i> de cada projeto para o algoritmo <i>Word2Vec</i> de acordo com o número de classes.	93
4.5	Um <i>boxplot</i> dos valores de MoJoFM produzidos pelos três algoritmos e pelo modelo <i>Wiki-en</i> para a configuração DOC-REMV-UNIQ.	101
4.6	Projeção t-SNE das proximidades da palavra <i>file</i> para um modelo de incorporação de palavras gerado pelo algoritmo <i>Word2Vec</i> e o modelo <i>Wiki-en</i>	102
4.7	Projeção t-SNE das proximidades da palavra <i>name</i> para um modelo de incorporação de palavras gerado pelo algoritmo <i>Word2Vec</i> e o modelo <i>Wiki-en</i>	102

LISTA DE TABELAS

2.1		11
2.2	Pontuações do TF-IDF para todas as palavras [50].	13
3.1	Sistemas de software selecionados para o estudo. A coluna “Pacotes” apresenta o número de pacotes da implementação original dos softwares e a coluna “Classes” mostra o número de classes nesta implementação.	50
3.2	Valores MoJoFM para o agrupamento HAC aplicado às séries s-DEV, c-DEV e f-DEV, utilizando o modelo de incorporação de palavras <i>Google News</i>	57
3.3	Valores MoJoFM para o agrupamento HAC aplicado às séries s-ILS, c-ILS e f-ILS, utilizando o modelo de incorporação de palavras <i>Google News</i>	57
3.4	Cálculo do MoJoFM para o agrupamento f-DEV utilizando os modelos de incorporação de palavras <i>Word2Vec Google News (GNews)</i> , <i>Glove.6B (Glove)</i> , <i>FastText-wiki-news (Wiki-News)</i> , <i>FastText-wiki-en (Wiki-en)</i> e <i>SO_Word2Vec (SO)</i>	59
3.5	Proporção de palavras ignoradas (Ign.) para os sistemas de software, calculada para os diferentes modelos de incorporação de palavras (<i>GNews</i> , <i>Glove</i> , <i>SO</i> , <i>Wiki News</i> e <i>Wiki-en</i>).	60
3.6	Exemplos de palavras encontradas nos modelos de incorporação de palavras <i>Google News</i> e <i>Wiki-en</i> e palavras similares para variações do termo “date”.	61
3.7	Percentual de palavras ignoradas, removendo e considerando as <i>stop words</i> , e usando o modelo de incorporação de palavras <i>Wiki-en</i> nas modalidades M1 a M4.	64
3.8	Valores de MoJoFM obtidos nos experimentos M1 a M4 com o modelo de incorporação de palavras <i>Wiki-en</i>	65
3.9	Percentual de palavras ignoradas usando o modelo de incorporação <i>Wiki-en</i> para as modalidades M5 a M12 com a remoção de <i>stop words</i>	66
3.10	Valores de MoJoFM para os experimentos M5 a M12 utilizando o modelo de incorporação <i>Wiki-en</i> sem <i>stop words</i>	67
3.11	Valores de MoJoFM para as modalidades M1 a M12 considerando somente os termos linguísticos da classe gramatical “substantivo”.	68

3.12	Resultados do MoJoFM para a modalidade M4 usando o agrupamento aglomerativo, o modelo de incorporação de palavras <i>Wiki-en</i> , diferentes medidas de distância e critérios de ligação.	69
3.13	Resultados do MoJoFM para a modalidade M4 usando os algoritmos <i>HAC</i> com a configuração EW, <i>K-Means</i> (KM), <i>Birch</i> (BI) e <i>Spectral Clustering</i> (SC) e o modelo de incorporação <i>Wiki-en</i>	70
4.1	Número de projetos por categoria na Fundação Apache.	80
4.2	Número de projetos por linguagem de programação na Fundação Apache.	81
4.3	Número de linhas de código (LOC) por linguagem de programação nos projetos da Fundação Apache.	81
4.4	Distâncias entre as palavras “ <i>pie</i> ” e “ <i>dessert</i> ” em diferentes configurações de treinamento de modelos de incorporação de palavras.	88
4.5	Alguns parâmetros usados na vetorização	88
4.6	Eficácia dos algoritmos de treinamento de modelos de incorporação de palavras na validação cruzada com a modalidade M4.	91
4.7	Relação de dominância entre as configurações PAR-KEEP-REPT e PAR-KEEP-UNIQ em três projetos.	96
4.8	Resultado para o número de dominâncias dos pares de parágrafo e documento ($D = \text{número de dominâncias (min = 0, máx = 78)}$; $\% = \text{percentual de dominância, entre 0 e 100\%}$).	97
4.9	Número de vitórias para cada algoritmo (<i>Word2Vec</i> , <i>Doc2Vec</i> e <i>FastText</i>) para cada projeto. A coluna “Palavras” indica o número de palavras únicas no projeto.	97
4.10	Número de vitórias dos modelos de incorporação de palavras gerados por cada algoritmo (<i>Word2Vec</i> , <i>Doc2Vec</i> e <i>FastText</i>) para cada projeto e do modelo de incorporação de palavras <i>Wiki-en</i> . O número de palavras de cada projeto indica o número de palavras na representação do seu documento, sem repetição. O número de palavras do <i>Wiki-en</i> indica o número de palavras do vocabulário do projeto que também são encontradas no modelo <i>Wiki-en</i>	103
4.11	Número e tamanho dos projetos das organizações selecionadas para a criação do primeiro modelo, totalizando 1.864.580 arquivos Java.	107
4.12	Média dos valores <i>MoJoFM</i> para a seleção dos 78 projetos da categoria “library” e os 183 projetos da <i>Apache Software Foundation</i> selecionados, calculados usando os diversos modelos estudados.	108
4.13	Média dos valores <i>MoJoFM</i> para a seleção dos 78 projetos da categoria “library” e os 183 projetos selecionados calculados usando os diversos modelos estudados, incluindo o modelo de ajuste-fino.	110

5.1	Dois projetos de software com melhores e piores valores do MoJoFM.	116
A.1	Lista dos 183 projetos selecionados na Seção 4.3, com suas respectivas categorias, número de classes escritas na linguagem de programação Java e número de termos linguísticos identificados em cada projeto.	118

SUMÁRIO

1	INTRODUÇÃO	1
1.1	Contexto	1
1.2	Objetivo	3
1.3	Questão de Pesquisa	4
1.4	Contribuições Esperadas	5
1.5	Metodologia de Pesquisa	6
1.6	Organização do Trabalho	7
2	FUNDAMENTAÇÃO TEÓRICA E TRABALHOS RELACIONADOS	8
2.1	Processamento de Linguagem Natural	8
2.1.1	Bag-of-Words	10
2.1.2	TF-IDF	11
2.1.3	Modelos de Incorporação de Palavras	13
2.1.4	Técnicas de Agrupamento	21
2.1.5	Remodularização de Software	23
2.2	Trabalhos Relacionados	24
2.2.1	Remodularização baseada na Estrutura do Software	24
2.2.2	Remodularização baseada em Análise Léxica	30
2.2.3	Remodularização baseada em Informações Estruturais e Léxicas	34
2.3	Considerações Finais	37
3	REMODULARIZAÇÃO DE SOFTWARE E MODELOS DE INCORPORAÇÃO DE PALAVRAS PRÉ-TREINADOS	40
3.1	Introdução	40
3.2	Algoritmos de Agrupamento	41
3.2.1	Iterated Local Search	41
3.2.2	K-Means	43
3.2.3	Spectral Clustering	44
3.2.4	BIRCH Clustering	45
3.2.5	Agrupamento Hierárquico	47
3.3	Seleção dos Sistemas de Software	49
3.4	Seleção de Modelos de Incorporação de Palavras Pré-treinados	51
3.5	Experimentos baseados nos Nomes de Classes	52
3.5.1	Experimentos com Variações de Modalidade	53
3.5.2	Experimentos com Variações de Modelos de Incorporação	58
3.6	Experimentos baseados em outras Informações da Classe	62

3.6.1	Experimentos com Palavras de Outras Áreas do Código	62
3.6.2	Experimentos com Recombinações de Áreas do Código	64
3.6.3	Filtro por Classes Gramaticais	67
3.7	Experimentos com outros Algoritmos	68
3.8	Ameaças à Validade	70
3.9	Considerações Finais	72
4	REMODULARIZAÇÃO DE SOFTWARE E MODELOS DE INCORPORAÇÃO DE PALAVRAS ESPECÍFICOS	74
4.1	Introdução	74
4.2	Geração de Modelos de Incorporação de Palavras	75
4.3	Sistemas da <i>Apache Foundation</i>	77
4.4	Desenvolvimento dos modelos de incorporação de palavras	83
4.4.1	Validação cruzada	88
4.4.2	Análise de dominância das configurações	92
4.4.3	Análise dos algoritmos	97
4.4.4	Comparação com o modelo Wiki-en	100
4.4.5	Comparação com um modelo reduzido	105
4.5	Um modelo de incorporação de palavras de larga escala	105
4.6	Um modelo de incorporação de palavras gerado por ajuste fino	108
4.7	Considerações finais	110
5	CONCLUSÕES	112
5.1	Introdução	112
5.2	Resultados encontrados	112
5.3	Trabalhos futuros	115
APÊNDICE A	PROJETOS SELECIONADOS PARA GERAÇÃO DE MODELOS DE INCORPORAÇÃO DE PALAVRAS	118
REFERÊNCIAS		123

1 INTRODUÇÃO

Este capítulo apresenta o contexto do trabalho de pesquisa, bem como o seu objetivo, as questões de pesquisa que orienta o estudo, contribuições esperadas e metodologia de pesquisa.

1.1 Contexto

A qualidade de programas e aplicações está diretamente ligada a um *design* eficiente e ao uso otimizado de dispositivos de armazenamento e processamento [121]. Um *design* bem elaborado e cuidadoso requer grande investimento de tempo e recursos, mas ainda pode ser considerado mais barato do que ter um sistema de software de difícil evolução e não confiável, quando o custo de manutenção é levado em conta [121]. Por esta razão, grandes empresas e desenvolvedores de software se preocupam com a manutenção e o gerenciamento das versões dos seus sistemas [130].

A evolução de um sistema de software ao longo do tempo pode resultar em uma estrutura de classes muito entrelaçada, com um excesso de dependências entre as classes. Uma dependência $A \rightarrow B$ ocorre quando uma classe A depende de um método ou atributo da classe B para atingir os seus objetivos. Quando o número de dependências em um software se torna muito grande, qualquer alteração em alguma parte do sistema pode provocar efeitos imprevistos em outras partes, que podem deixar de funcionar corretamente conforme o esperado pelos desenvolvedores.

O excesso de dependências entre classes reduz a capacidade de reutilização, a compreensibilidade e dificulta a manutenção do projeto [12]. Uma distribuição considerada adequada de classes em pacotes auxilia na identificação das classes responsáveis por uma determinada funcionalidade, facilita a navegação entre as partes do software, aumenta a compreensão do código-fonte e facilita a manutenção de um sistema

de software. Experimentos mostraram uma forte correlação entre distribuições de pacotes ruins e a ocorrência de falhas em sistemas de software [38].

Software Module Clustering (SMC) é o problema de encontrar uma distribuição adequada para as classes que representam os conceitos de domínio e as construções computacionais que compõem um programa em estruturas maiores, semelhantes a contêineres. Este problema está fortemente associado à definição de componentes e suas conexões em arquiteturas de software [12], abordando o agrupamento de classes (ou módulos) em pacotes (ou *clusters*) que sejam relevantes para os desenvolvedores. Uma solução para o problema representa uma reorganização da estrutura de um sistema, tentando maximizar a coesão e minimizar o acoplamento [12]. Por isso, este problema também é conhecido como “remodularização de sistemas de software”.

Várias técnicas foram criadas para apoiar a remodularização de sistemas de software [5, 86]. A maioria aplica algoritmos de agrupamento aos sistemas para particioná-los em subsistemas [78]. Devido ao grande número de distribuições alternativas de classes em pacotes para programas não triviais, esse problema é frequentemente abordado sob a perspectiva da Engenharia de Software baseada em Buscas (*Search-based Software Engineering*, ou SBSE) [12]. A SBSE reformula problemas de Engenharia de Software como problemas de otimização e sugere o uso de algoritmos de busca meta-heurísticos para encontrar boas soluções para eles [47].

A maioria das abordagens de remodularização de sistemas de software, chamadas *abordagens estruturais*, emprega engenharia reversa e algoritmos de agrupamento para particionar os sistemas, analisando as dependências estruturais entre os artefatos de software componentes do sistema [6, 11, 35, 38, 78, 120]. Para avaliar a qualidade da distribuição das classes em pacotes, um sistema de software é representado como um Grafo de Dependência de Módulos (*Module Dependency Graph – MDG*). O MDG é um grafo direcionado no qual as classes são apresentadas como *nós*, as dependências são as *arestas* entre os nós e os pacotes (ou *clusters*) são partições do grafo.

Mancoridis et al. [35] propuseram utilizar uma busca local, baseada no algoritmo de *Hill Climbing*, para tentar buscar uma partição de qualidade de classes em pacotes

para um sistema. A busca é guiada por uma função objetivo, chamada *Modularization Quality* (MQ), que busca um equilíbrio entre o acoplamento e a coesão. Pinto [38] e Monçores [87] realizaram a partição de programas escritos em Java usando as dependências entre classes, mas os resultados de buscas baseadas nos algoritmos *Iterated Local Search* (ILS) e *Large Neighborhood Search* (LNS) guiados pela função objetivo MQ ficaram muito diferentes do agrupamento proposto pelos desenvolvedores.

Outras abordagens de remodularização de sistemas de software exploram informações léxicas extraídas do código-fonte para identificar os subsistemas de software [27, 28, 100]. Estas abordagens utilizam técnicas de processamento de linguagem natural. A ampla adoção destas técnicas nas pesquisas de Engenharia de Software se alinha com os avanços recentes no campo de processamento de texto não-estruturado, oferecendo oportunidades para o processamento eficiente de grandes quantidades de informação. Artefatos de software que alimentam técnicas de processamento de linguagem natural para fins de remodularização do software incluem relatórios de problemas, revisões de código e documentação de código-fonte [37]. Existem ainda abordagens mistas para a remodularização de sistemas de software, que utilizam informações léxicas conjugadas com informações estruturais [3, 72].

1.2 Objetivo

O objetivo deste estudo é avaliar como os termos linguísticos podem ser utilizados pelos desenvolvedores com o propósito de orientar a distribuição de classes em pacotes. Para isso, pretende-se avaliar se o uso de técnicas baseadas em informações léxicas, aplicando modelos de incorporação de palavras (*word embeddings*) para chegar a um agrupamento compatível com aquele que os desenvolvedores propuseram.

A intenção não é propor ao desenvolvedor uma solução diferente, pois os desenvolvedores poderiam estranhar e não aceitar uma nova versão, mas estudar o espectro de soluções para ver se dentro de uma mesma categoria ou dentro de um mesmo grupo de softwares existe algum padrão de uso desta informação e mostrar possíveis soluções para ajudar os desenvolvedores a enxergar possíveis abordagens que possam

melhorar a arquitetura do software no que tange a acoplamento e coesão. Assim, o objetivo desta pesquisa é verificar se, por meio do uso de termos linguísticos, é possível explicar a divisão que foi proposta pelos desenvolvedores.

Nesta direção, realizamos experimentos iniciais utilizando técnicas e ferramentas de processamento de linguagem natural e algoritmos de busca reportados na literatura de Engenharia de Software. Além disso, selecionamos sistemas de software de código aberto (*open-source*) com diferentes finalidades e tamanhos, coletando dados sobre os termos léxicos utilizados em diferentes áreas do código-fonte destes sistemas. Os resultados destes experimentos mostraram que os modelos de incorporação de palavras de uso geral não são muito eficientes do ponto de vista da distribuição de classes em pacotes.

Para verificar se é possível melhorar os resultados destes experimentos, desenvolvemos um *corpus* específico para a distribuição de classes em pacotes e um modelo de incorporação de palavras foi treinado para esta tarefa. Em seguida, analisamos diversos projetos de software para que os termos utilizados em diferentes áreas do seu código-fonte fossem considerados no *corpus* e, conseqüentemente, no treinamento do novo modelo. Para isto, utilizamos os projetos de software da *Apache Foundation* e, depois, um amplo conjunto de sistemas de software produzidos por grandes empresas de tecnologia. Os dois conjuntos foram considerados usando diferentes técnicas de treinamento de modelos de incorporação de palavras.

1.3 Questão de Pesquisa

As seguintes questões de pesquisa foram levantadas a partir da definição do problema e do objetivo desta pesquisa, estabelecendo o escopo para investigação.

QP1: Uma abordagem de agrupamento baseada em termos léxicos pode levar a resultados melhores do que os resultados obtidos usando abordagens estruturais?

Para avaliar esta questão de pesquisa, comparamos a distribuição de classes em pacotes gerada por diferentes abordagens com a distribuição de classes em pacotes proposta pelos desenvolvedores de sistemas de software selecionados para o estudo. A intenção é verificar se as abordagens léxicas geram distribuições mais próximas àquela proposta pelos desenvolvedores quando comparadas com distribuições estruturais.

QP2: É possível melhorar os resultados das abordagens de agrupamento baseadas em termos léxicos construindo um corpus específico para o problema de agrupamento?

Para avaliar esta questão de pesquisa, construímos um *corpus* específico, assim como um modelo vetorial para orientar o problema de distribuição de classes em pacotes, utilizando diferentes estratégias para a coleta dos termos léxicos utilizados na construção do *corpus*.

1.4 Contribuições Esperadas

Durante o desenvolvimento deste trabalho, foram produzidos softwares e resultados de análises. Consideramos que as seguintes contribuições foram atingidas:

1. A divulgação dos resultados da análise de algoritmos de agrupamento aplicados a abordagens de distribuição de classes em pacotes baseados em informações léxicas coletadas do código-fonte de programas escritos usando a linguagem Java;
2. O desenvolvimento de um *corpus* específico para o problema da remodularização de software, capturando conceitos do domínio da Engenharia de Software;
3. Uma estratégia para remodularização de software utilizando o algoritmo mais adequado de agrupamento identificado e um modelo de incorporação de palavras desenvolvido com base no *corpus* citado acima.

1.5 Metodologia de Pesquisa

A metodologia utilizada nesta pesquisa compreende um processo de cinco etapas. Este processo começa com a **definição do tema da pesquisa**, do problema em análise, do objetivo da pesquisa, das questões que serão analisadas e da metodologia que será aplicada nesta análise. Estes itens são relatados nesta introdução e orientam o desenvolvimento do restante deste documento.

A partir das definições iniciais, iniciamos uma **revisão da literatura** com estudos de abordagens estruturais e léxicas disponíveis para identificar o conhecimento necessário para prosseguir com a pesquisa. Esta busca foi realizada de modo *ad-hoc*, através de referências identificadas a partir de um *snowballing* baseado em um conjunto de artigos de referência. Nesta revisão, priorizamos a busca por publicações sobre abordagens léxicas para a distribuição de classes em pacotes com o propósito de reconhecer os estudos realizados e identificar tópicos ainda não explorados.

Com as informações obtidas da literatura, planejamos a abordagem, que envolve a seleção dos softwares a serem usados nos experimentos, a definição das linguagens e das bibliotecas necessárias e a construção ou seleção das ferramentas que forem necessárias para o desenvolvimento do projeto, incluindo aqui a estrutura necessária para testar os diferentes algoritmos de agrupamento, a coleta de dados para construção do *corpus* e o treinamento do modelo de incorporação de palavras para a modularização.

A solução proposta será avaliada com a execução de estudos experimentais e análises estatísticas sobre os resultados. Nesta etapa, usaremos os conjuntos de dados selecionados e as ferramentas desenvolvidas para a realização dos experimentos. Esta etapa estabelece:

- Experimentos iniciais, baseados nos nomes de classes, utilizando poucos softwares e modelos pré-treinados de incorporação de palavras;
- Experimentos baseados em outras informações da classe, analisando o conteúdo dos arquivos de código-fonte Java;

- Experimentos com outros algoritmos de agrupamento e refinamento das estratégias do agrupamento aglomerativo;
- Experimentos utilizando uma quantidade maior de softwares para geração dos *corpora* para o desenvolvimento de modelos de incorporação de palavras e ajuste fino destes modelos para a modularização de software.

Estes experimentos serão realizados a partir das especificações definidas na etapa de planejamento para a **avaliação da solução**.

Finalmente, os dados gerados nesta etapa servirão para as análises estatísticas definidas na etapa anterior, onde a **conclusão** resume os achados, possíveis problemas e resultados finais.

1.6 Organização do Trabalho

Este trabalho está organizado em cinco capítulos. O Capítulo 1 apresenta esta introdução com a definição do problema, objetivos, a questão de pesquisa, contribuições esperadas e a metodologia da pesquisa.

O Capítulo 2 apresenta uma fundamentação teórica sobre processamento de linguagem natural e uma revisão da literatura sobre trabalhos relacionados a abordagens de modularização estrutural, léxica e mista.

O Capítulo 3 apresenta a seleção dos softwares, a definição das estratégias para a realização dos experimentos utilizando modelos de incorporação de palavras e o agrupamento de classes em pacotes e os resultados desses estudos.

O Capítulo 4 apresenta a proposta de pesquisa, que consiste no treinamento de modelos de incorporação de palavras a partir de grupos de projetos de software desenvolvidos por organizações não-governamentais (como a *Apache Software Foundation*), ou grandes empresas do ramo.

Por fim, Capítulo 5 apresenta as conclusões do trabalho de pesquisa, as limitações do trabalho e suas perspectivas de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA E TRABALHOS RELACIONADOS

Este capítulo apresenta a fundamentação teórica necessária para o entendimento da Tese. Essa fundamentação teórica envolve conceitos de Processamento de Linguagem Natural que serão utilizados de formas diversas para a remodelarização de software. Em seguida, analisamos os trabalhos relacionados sobre remodelarização de software, a começar por aqueles que utilizam informações sobre a estrutura do projeto de software, seguindo para os que utilizam informações léxicas e as abordagens mistas.

2.1 Processamento de Linguagem Natural

O *Processamento de Linguagem Natural* (*Natural Language Processing* – NLP) é um ramo da Inteligência Artificial (IA) que trata do processamento de linguagens humanas por meio de máquinas. Atualmente, vemos implementações de técnicas de NLP em ferramentas como os assistentes de voz, mecanismos de busca e serviços de tradução automática, que têm por objetivo simplificar tarefas da vida diária [21].

O NLP combina linguística computacional – modelagem baseada em regras da linguagem humana – com modelos estatísticos, de aprendizado de máquina e aprendizado profundo, permitindo que os computadores processem a linguagem humana na forma de texto ou dados de voz e “compreendam” seu significado. Samuel [102] definiu *machine learning* como o campo de estudo que dá aos computadores a capacidade de aprender sem serem explicitamente programados. Mais tarde, foi fornecida uma definição de *machine learning* mais orientada para a engenharia: diz-se que um problema de computador aprende com a experiência E em relação a alguma tarefa T e alguma medida de desempenho P , se seu desempenho em T , medido por P , melhora com a experiência E [93].

A forma mais comum de categorizar tarefas de *machine learning* leva a paradigmas de aprendizado supervisionado e não supervisionado. O aprendizado

supervisionado implica na presença da variável que se espera prever para que o modelo possa receber *feedback* do “professor”, indicando se a previsão está próxima da verdade. No aprendizado semi-supervisionado, os valores-alvo estão parcialmente ausentes. Quando a variável de destino não está presente, o objetivo normalmente se torna encontrar padrões nos dados ou agrupamentos de pontos de dados que parecem se encaixar em um mesmo padrão [93].

Alguns autores utilizam a denominação “tradicional” para definir os algoritmos que não exigem intervenção humana [104, 93]. Recentemente, de acordo com Câmara Pereira e Borysov [93], quatro tendências em *machine learning* foram formadas:

- Sistemas baseados em regras, que utilizam regras escritas manualmente, como sistemas especialistas, árvores de decisão, tabelas de decisão e programação lógica. São considerados sistemas de aprendizado de máquina interpretáveis;
- Algoritmos baseados em *kernel*, focam no conceito de vizinhança, incluindo algoritmos como K-NN [60], SVM [66] e seus derivados. Exigem uma definição cuidadosa de similaridade (ou *kernel*) e não demandam a definição da forma da função modelo;
- Estatística Bayesiana, que se baseia na teoria da probabilidade Bayesiana para fazer inferências nos dados, ou seja, prever a distribuição de uma variável de interesse e os parâmetros do modelo a partir de um conjunto de observações;
- Redes neurais (*Deep Neural Networks*), que refletem o renascimento da visão conexionista para aprendizado de máquina. Atingiram padrões fantásticos nas áreas que demandam muitos dados de alta dimensão, como visão computacional, tradução automática, robótica, jogos de computador e reconhecimento de fala.

O NLP melhorou o processamento de linguagem natural utilizando redes neurais com os modelos de incorporação de palavras (*word embeddings*), que são representações vetoriais (numéricas) para palavras, de modo que o vetor que representa uma palavra preserva seu significado: palavras semanticamente semelhantes são mapeadas para

vetores semelhantes [84]. As técnicas de incorporação de palavras usam redes neurais, modelos probabilísticos ou redução de dimensão em matrizes de coocorrência de palavras para construir os vetores numéricos que representam as palavras [8].

Os modelos de incorporação de palavras podem ser treinados para capturar a semântica do código-fonte e usados para uma série de tarefas *down-stream*, como identificação do autor do código e classificação de *malware*, aplicando ajustes mínimos e menos dados de treinamento do que necessário para treinar um modelo para essas tarefas a partir do zero [25]. As formas mais populares para incorporação de palavras são BoW (*Bag-of-words*) e TF-IDF (*Term Frequency-Inverse Document Frequency*), que são técnicas de pré-processamento que podem gerar uma forma numérica a partir de um texto de entrada.

2.1.1 Bag-of-Words

Bag-of-Words (BOW) é uma abordagem simples para representação vetorial de documentos. Um texto é representado como um saco (*bag*) contendo muitas palavras. A gramática e a ordem das palavras não são tratadas, enquanto a frequência das palavras é mantida [8]. Um exemplo didático pode ser construído a partir de resenhas de filmes. Considere as seguintes resenhas [50]:

- Resenha R1: Este filme é muito assustador e longo;
- Resenha R2: Este filme não é assustador e é lento;
- Resenha R3: Este filme é assustador e bom.

A partir destas resenhas, é possível construir um vocabulário com todas as palavras (únicas) nas três resenhas. O vocabulário consistirá de dez palavras: “e”, “assustador”, “bom”, “este”, “filme”, “lento”, “longo”, “muito”, “não” e “é”. A seguir, tomamos cada uma dessas palavras marcando sua ocorrência nas três resenhas com 1’s e 0’s. Este processo produzirá três vetores, sendo um para cada resenha (Tabela 2.1).

Aplicações do algoritmo *Bag-of-Words* podem ser encontradas em vários artigos [41, 81, 106, 119]. Em Verma et al. [119], os autores implementaram um *chatbot*, um

Tabela 2.1

	e	assustador	bom	este	filme	lento	longo	muito	não	é	Tamanho
R1	1	1	0	1	1	0	1	1	0	1	7
R2	1	1	0	1	1	1	0	0	1	2	8
R3	1	1	1	1	1	0	0	0	0	1	6

recurso de Inteligência Artificial que pode ser incorporado e usado em qualquer aplicação de mensagens, usando BoW e o modelo *seq2seq* [111]. Os autores investigaram os modelos utilizados e sua complexidade na aprendizagem e para implementação. Após a construção dos modelos, concluíram que o modelo BoW não aprende com a quantidade de amostras, mas sempre escolhe o resultado de um conjunto específico de palavras. Como o modelo BoW não é um modelo baseado em redes neurais, ele não requer semanas de treinamento para obter os seus resultados. Assim, um algoritmo de aprendizado de máquina baseado em aprendizado estatístico tradicional seria suficiente.

No trabalho de Gabryel et al. [41], o algoritmo BoW foi modificado para fins de classificação de dados sobre vendas coletados no *site* de um grande banco polonês. Para fins de comparação, os mesmos experimentos foram conduzidos usando um classificador Bayesiano *Naive*. Os resultados mostraram que o algoritmo apresentado funcionou melhor do que o classificador Bayesiano. Os autores também ressaltaram que a classificação provavelmente poderia ser feita de forma mais eficaz por outros métodos de inteligência computacional, como redes neurais, sistemas *fuzzy* e outros. No entanto, o custo desses métodos é muito maior do que os cálculos simples realizados pelo algoritmo BoW. Sua grande vantagem também inclui a possibilidade de extensão direta da base de conhecimento com novos *leads* de vendas, enquanto o sistema está em operação.

2.1.2 TF-IDF

O *Term Frequency-Inverse Document Frequency* (também conhecido como TF-IDF) é outra maneira de representar um documento de forma vetorial com base em suas palavras. O TF-IDF é um esquema de pontuação que representa uma medida de quão

importante uma palavra é para um documento [2]. Quanto maior o valor do TF-IDF para um termo em um documento, mais relevante ele é no contexto do *corpus*.

A criação de representações vetoriais de palavras com TF-IDF é composta por dois valores: o primeiro calcula a Frequência de Termo normalizada (*Term Frequency* - TF), ou seja, o número de ocorrências de uma palavra em um documento dividido pelo número total de palavras naquele documento; o segundo representa o *Inverse Document Frequency* (IDF), que calcula o logaritmo do número de documentos no *corpus* dividido pelo número de documentos onde o termo sob análise aparece [8]. Em linguística, um *corpus* (latim para *corpo*) refere-se a uma coleção de textos. Tais coleções podem ser formadas por um único idioma ou podem abranger vários idiomas – existem inúmeras razões pelas quais um *corpora* multilíngue ¹ pode ser útil [82]. A pontuação de cada termo em um documento é calculada da seguinte forma:

$$TF\text{-}IDF(t, d) = TF(t, d) \times IDF(t) \quad (2.1)$$

$$IDF(t) = \log \frac{N}{1 + n_t} \quad (2.2)$$

Nestas equações, t representa a palavra ou termo sob análise, d representa um documento do *corpus*, N representa o número total de documentos no *corpus* e n_t representa o número de documentos em que o termo está presente. A adição da constante 1 no denominador do IDF evita o erro de divisão por zero [9].

Com base no exemplo anterior, as pontuações do TF-IDF para as palavras em relação às resenhas são apresentadas na Tabela 2.2. O TF-IDF produz valores maiores para palavras menos frequentes e é particularmente alto quando tanto os valores de IDF quanto de TF são altos, ou seja, a palavra é rara em todos os documentos combinados, mas frequente em um único documento.

TF-IDF tem sido explorada em aplicações de recuperação de informação e extração de palavras-chave. Para estudar a existência de plágio entre dois códigos-fonte

¹ Um *corpora* é o plural de *corpus*.

Tabela 2.2: Pontuações do TF-IDF para todas as palavras [50].

Termo	R1	R2	R3	TD-IDF (R1)	TD-IDF (R2)	TD-IDF (R3)
e	1	1	1	0,269	0,244	0,326
assustador	1	1	1	0,269	0,244	0,326
bom	0	0	1	0,000	0,000	0,684
este	1	1	1	0,269	0,244	0,326
filme	1	1	1	0,269	0,244	0,326
lento	0	1	0	0,000	0,512	0,000
longo	1	0	0	0,565	0,000	0,000
muito	1	0	0	0,565	0,000	0,000
não	0	1	0	0,000	0,512	0,000
é	1	2	1	0,269	0,488	0,326

escritos em linguagens diferentes, Karnalim [56] utilizou uma técnica inspirada em TF-IDF, analisando a importância dos termos no contexto de resposta individual e em conjunto. Segundo Souza [109], o aprendizado de programação de computadores engloba o desenvolvimento de um conjunto de habilidades como lógica, matemática, assim como outras não diretamente relacionadas com a Ciência da Computação. Com base nestas habilidades, Souza apresentou uma proposta usando a TF-IDF para estudar a importância de uma habilidade para a construção de uma solução a partir de um código-fonte. A avaliação utilizando TF-IDF individualmente para as submissões demonstrou-se eficiente ao determinar a importância de cada habilidade para a solução. No entanto, não determina um nível de habilidade utilizado na construção da resposta.

2.1.3 Modelos de Incorporação de Palavras

Embora tanto o BoW quanto o TF-IDF tenham sido populares na comunidade de NLP, ainda havia um vazio no que se refere à compreensão do contexto das palavras [50]. Novas técnicas utilizando redes neurais foram desenvolvidas para levar em consideração a compreensão do contexto das palavras nas sentenças e *corpus*. Estas técnicas são conhecidas como modelos de incorporação de palavras e fornecem uma representação vetorial para palavras que captura algo sobre o seu significado [2].

Os modelos de incorporação de palavras se baseiam na hipótese de que as palavras vizinhas em um texto têm semelhança semântica entre si. Estes modelos auxiliam no mapeamento de palavras semanticamente semelhantes para vetores geometricamente próximos. Eles utilizam uma métrica de similaridade para medir a similaridade semântica entre os termos. Por exemplo, a métrica de similaridade de cosseno é calculada como $\cos(\phi)$, onde o ϕ é o ângulo entre a representação vetorial de duas palavras ou documentos. Portanto, se o ângulo for *zero*, as palavras estão sobrepostas. Por outro lado, se o ângulo for 90° , as palavras não possuem similaridade contextual e são independentes entre si.

Os modelos de incorporação de palavras *Word2Vec*², *Glove*³, *FastText*⁴ e *BERT*⁵ são independentes de contexto e produzem apenas um vetor para cada palavra, combinando todos os diferentes sentidos da palavra neste vetor [2].

2.1.3.1 *Word2Vec*

O *Word2Vec*, idealizado por Tomas Mikolov, atualmente pesquisador do CIIRC⁶ (*Czech Institute of Informatics, Robotics and Cybernetics*) [118], é uma rede neural rasa de três camadas que processa texto “vetorizando” palavras. Sua entrada é um *corpus* de texto e sua saída é um conjunto de vetores que representam as palavras nesse *corpus* [89]. O *Word2vec* codifica cada palavra em um vetor, mas em vez de treinar contra as palavras de entrada por meio de reconstrução, como uma máquina de Boltzmann restrita, ele treina palavras contra palavras vizinhas no *corpus* [89].

A arquitetura do *Word2vec* é semelhante à de um *autoencoder*, que recebe um vetor de entrada grande e compacta-o em um vetor denso menor. Em vez de descompactá-lo de volta ao tamanho do vetor de entrada original, produzindo

² Google Word2Vec - <https://code.google.com/archive/p/word2vec/>

³ GloVe: Global Vectors for Word Representation, <https://nlp.stanford.edu/projects/glove/>

⁴ FastText, Library for efficient text classification and representation learning, <https://fasttext.cc/>

⁵ BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, <https://arxiv.org/abs/1810.04805>

⁶ Czech Institute of Informatics, Robotics and Cybernetics https://en.wikipedia.org/wiki/Czech_Institute_of_Informatics,_Robotics_and_Cybernetics

probabilidades de palavras-alvo [43], a saída da rede neural do *Word2vec* é um vocabulário em que cada item está associado a um vetor. Este vetor pode ser alimentado em uma rede de aprendizado profundo para ser utilizado em outras tarefas de NLP ou pode ser consultado para detectar relações entre as palavras [89].

Uma máquina de Boltzmann restrita é uma rede estocástica com duas camadas: uma visível e outra oculta [108, 48]. O *Word2Vec* tem um modelo em três camadas. A camada de entrada tem tantos neurônios quantas palavras no vocabulário para treinamento. O número de neurônios na camada oculta é igual à dimensão dos vetores de palavras resultantes. A terceira e última camada incluída no *Word2Vec* possui o mesmo número de neurônios que a camada de entrada [14].

A eficácia do *Word2Vec* vem de sua capacidade de agrupar vetores de palavras semelhantes. Dado um conjunto de dados grande o suficiente, o *Word2Vec* pode fazer estimativas sobre o significado de uma palavra com base em suas ocorrências no texto. A camada visível da rede representa os dados observados e está conectada à camada oculta, que deverá aprender a extrair características desses dados. Os vetores de palavras são posicionados no espaço vetorial de modo que palavras que compartilham contextos comuns no *corpus* estejam localizadas próximas umas das outras no espaço [43].

O *Word2Vec* é um modelo preditivo eficiente em termos de computação para aprender incorporações de palavras a partir de texto bruto. Do ponto de vista de uso prático, o *Word2Vec* produz um vetor por palavra, enquanto o TF-IDF produz uma pontuação por palavra. O *Word2Vec* plota as palavras em um espaço vetorial multidimensional de forma que palavras semelhantes tendem a estar próximas umas das outras. As palavras ao redor de uma palavra fornecem o contexto para essa palavra [8]. O objetivo do *Word2vec* é agrupar os vetores de palavras semelhantes em um espaço vetorial, ou seja, ele detecta semelhanças matematicamente criando vetores que são representações numéricas distribuídas de recursos de palavras, como o contexto de palavras individuais, sem intervenção humana [89].

Existem duas arquiteturas principais que produzem modelos de incorporação de palavras com *Word2Vec*: as arquiteturas *skip-gram* e *Continuous Bag-of-Words* (CBOW).

A arquitetura CBOW, semelhante a uma rede neural *feed-forward*, prevê palavras-alvo a partir das palavras de contexto circundantes. Ou seja, dada uma determinada sentença, por exemplo, “o gato senta no tapete”, seleciona-se uma palavra-alvo, como “tapete”, e as palavras de contexto como [“o”, “gato”, “senta”, “no”]. Com base nestas escolhas, o modelo pega as representações distribuídas das palavras de contexto para tentar prever a palavra-alvo [43]. O modelo *Skip-Gram* é uma rede neural com uma camada oculta treinada para prever a probabilidade de uma determinada palavra estar presente dada uma palavra de entrada. O modelo *Skip-Gram* é o oposto do modelo CBOW: ele toma uma palavra como uma entrada e tenta prever as palavras antes e depois desta, ou seja, o modelo tenta aprender e prever as palavras de contexto em torno da palavra de entrada especificada [91]. As arquiteturas BOW e SKIP-GRAM podem ser vistas na Figura 2.1.

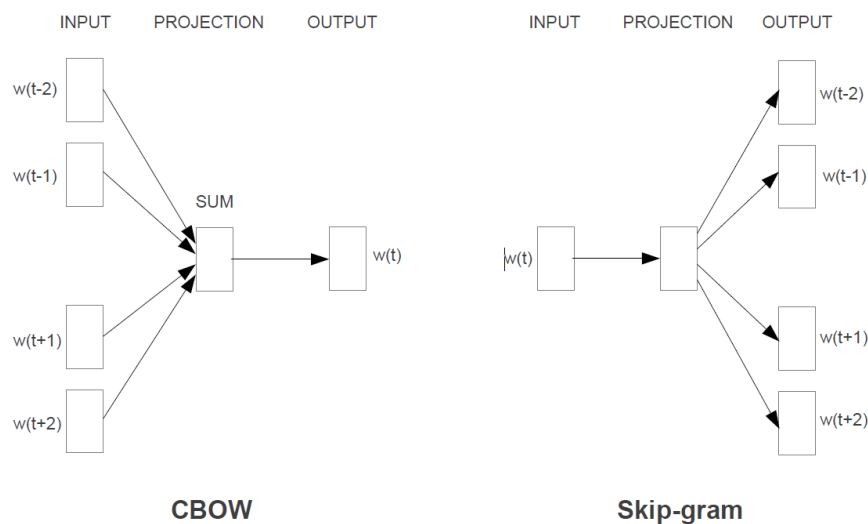


Figura 2.1: Arquitetura *CBOW/SKIP-GRAM*. Imagem retirada de [84]

Modelos de incorporação de palavras são treinados expondo um modelo a uma grande quantidade de dados de texto e ajustando as representações vetoriais com base no contexto em que as palavras aparecem. Um modelo define uma maneira de representar palavras como vetores em um espaço multidimensional, onde a similaridade semântica entre duas palavras é calculada pela distância entre os vetores que as representam. A

ideia principal é aprender representações que codificam o significado semântico e os relacionamentos entre as palavras.

A similaridade é capturada pelas posições relativas de dois vetores no espaço. Considerando uma consulta, teremos um vetor representando a palavra procurada e os vetores de todas as palavras do *corpus*. A similaridade entre o vetor de consulta e cada vetor do *corpus* é calculada usando a métrica do cosseno. Os documentos com as maiores pontuações de similaridade para a consulta retornam como resultados de busca. A métrica de cosseno mede o cosseno do ângulo entre dois vetores. Diferentemente das distâncias euclidiana e Manhattan, a métrica de distância de cosseno foca no ângulo entre os vetores (X, Y) ao invés da sua magnitude. Isso é particularmente útil para dados de alta dimensão, como modelos de incorporação de palavras, onde a magnitude dos vetores pode variar amplamente [29].

$$\text{Similaridade do Cosseno}(\mathbf{X}, \mathbf{Y}) = \frac{\mathbf{X} \cdot \mathbf{Y}}{\|\mathbf{X}\| \times \|\mathbf{Y}\|}$$

Os modelos de incorporação de palavras como *Word2Vec* permitem o uso de modelos prontamente disponíveis, pré-treinados em *corpora* de bilhões de palavras, que podem ser carregados e explorados para capacitar várias tarefas de NLP [37]. Um modelo pré-treinado é um arquivo contendo termos e seus vetores de palavras associados. O modelo do *Google* com base no *Word2Vec*, por exemplo, foi treinado nos dados de notícias do *Google* (cerca de 100 bilhões de palavras), contém cerca de 3 milhões de palavras e sentenças e foi calculado usando vetores de 300 dimensões [1].

Doc2Vec [63] é outra técnica derivada do *Word2Vec* que cria uma incorporação independentemente de seu comprimento. Enquanto *Word2Vec* calcula um vetor para cada palavra no *corpus*, o *Doc2Vec* calcula um vetor de características para cada documento no *corpus*. O modelo *Doc2Vec* é baseado no *Word2Vec*, apenas adicionando outro vetor (ID de parágrafo) à entrada. Enquanto os vetores de palavras do *Word2Vec* representam o conceito de uma palavra, o vetor de documentos pretende representar o conceito de um documento. Existem duas implementações do *Doc2Vec*:

- *Distributed Bag of Words* (DBOW): é o modelo análogo ao modelo *Skip-gram* no *WordVec*. Os vetores de parágrafo são obtidos treinando uma rede neural na tarefa de prever uma distribuição de probabilidade de palavras em um parágrafo dada uma palavra selecionada aleatoriamente do parágrafo;
- *Distributed Memory* (DM): atua como uma “memória” que lembra o que está faltando no contexto atual ou como o “tópico” do parágrafo.

2.1.3.2 GloVe

Pennington et al. [92] argumentam que a abordagem usada pelo *Word2Vec* é sub-ótima, uma vez que não explora totalmente as informações estatísticas globais relativas às coocorrências de palavras. No modelo que eles chamaram de *Global Vectors* (GloVe), os autores relatam que “o modelo produz um espaço vetorial com subestrutura significativa, como evidenciado pelo seu desempenho de 75% em uma recente tarefa de analogia de palavras, também superando modelos relacionados em tarefas de similaridade, e como reconhecimento de entidades”.

Ao invés de extrair as incorporações por uma rede neural que foi projetada para executar uma tarefa diferente, como prever palavras vizinhas (CBOW) ou prever a palavra-foco (*Skip-Gram*), as incorporações do *Glove* são otimizadas diretamente, de modo que o produto escalar de vetores de duas palavras equivale ao registro do número de vezes que as duas palavras ocorreram próximas uma da outra [99].

2.1.3.3 FastText

FastText é uma extensão do modelo *Word2Vec* desenvolvida pelo *Facebook AI Research Lab*. Seu foco principal é obter soluções escaláveis para as tarefas de classificação e representação de texto enquanto processa grandes conjuntos de dados com rapidez e precisão [17]. O *FastText* também suporta os modelos CBOW e *Skip-Gram*. Ao invés de aprender vetores para palavras diretamente, o *FastText* representa cada palavra como um *n-grama* de caracteres. Por exemplo, tomando a palavra “artificial” com $n = 3$, a representação FastText desta palavra será <AR, ART,

RTI, TIF, IFI, FIC, ICI, IAL, AL>, onde os símbolos de maior e menor indicam o início e o fim da palavra [99]. O *FastText* funciona bem com palavras raras. Então, mesmo que uma palavra não tenha sido vista durante o treinamento, ela pode ser dividida em n-gramas para obter suas incorporações. *Word2vec* e *GloVe*, por sua vez, não fornecem qualquer representação vetorial para palavras que não estão no dicionário utilizado no treinamento do modelo. Em suma, o *Word2Vec* trata cada palavra no corpus como uma entidade atômica e gera um vetor para cada palavra. Nesse sentido, *Word2Vec* é similar ao *Glove* — ambos tratam palavras como a menor unidade para treinar. Já o *FastText*, trata cada palavra como sendo composta de n-gramas de caracteres. Assim, o vetor para uma palavra é construído da composição desses n-gramas de caracteres.

2.1.3.4 BERT

BERT (*Bidirectional Encoder Representations from Transformers*), um modelo de aprendizado de máquina baseado em transformadores (*transformers*) [117], foi desenvolvido em 2018 por Jacob Devlin e seus colegas para aplicações de NLP [32]. BERT é um modelo pré-treinado que ajuda as máquinas a aprender representações do contexto de escrita de texto em muitas tarefas de processamento de linguagem natural [10]. A arquitetura BERT é composta por vários codificadores *transformer* empilhados, onde cada codificador é composto por duas subcamadas: uma camada *feed-forward* e uma camada *self-attention*. Com este esquema, o BERT aprende relações contextuais entre palavras em uma sentença [10].

Segundo Devlin et al. [32], ao contrário dos modelos recentes de representação linguística, BERT foi projetado para pré-treinar representações bidirecionais profundas a partir de textos não rotulados, condicionando conjuntamente no contexto dos lados esquerdo e direito em todas as camadas. Como resultado, o modelo BERT pré-treinado pode ser ajustado com apenas uma camada de saída adicional para criar modelos de última geração para uma ampla gama de tarefas, como resposta a perguntas e inferência de linguagem, sem modificações substanciais de arquitetura para cada tarefa [32].

2.1.3.5 *Aprendizado por Transferência*

O aprendizado por transferência (*Transfer Learning*) é um método de aprendizado de máquina onde um modelo desenvolvido para uma tarefa é reutilizado como ponto de partida para um modelo em uma segunda tarefa [19]. É uma abordagem popular no aprendizado profundo, onde modelos pré-treinados são usados como ponto de partida em tarefas de visão computacional e processamento de linguagem natural [19].

Na aprendizagem por transferência, o conhecimento de um modelo de aprendizado de máquina já treinado é aplicado a um problema diferente, onde a ideia geral é usar o conhecimento que um modelo aprendeu a partir de uma tarefa com muitos dados de treinamento rotulados disponíveis em uma nova tarefa que não tem muitos dados. Em vez de iniciar o processo de aprendizagem do zero, começamos com os padrões aprendidos para resolução de uma tarefa relacionada [34].

Um exemplo de uso do aprendizado por transferência é a classificação de imagens. Um modelo de aprendizado de máquina pode ser treinado com dados rotulados para identificar e categorizar o tema de um conjunto de imagens. O modelo pode então ser adaptado e reutilizado para identificar outro assunto específico dentro de um conjunto de imagens por meio do aprendizado por transferência. Os elementos gerais do modelo permanecerão os mesmos, economizando recursos que seriam consumidos em uma nova rodada completa de treinamento [49].

2.1.3.6 *Part-of-speech Tagging*

Part-of-speech tagging (PoS tagging) é o processo de classificar e rotular palavras, indicando sua classe gramatical em uma frase, sentença ou discurso (*part-of-speech*) [95]. Exemplos de classes gramaticais são substantivo, verbo, adjetivo, advérbio, conjunção, pronome, entre outros. As partes do discurso também são conhecidas como classes de palavras ou categorias léxicas. A coleção de *tags* utilizadas por uma tarefa específica é conhecida como conjunto de *tags* [97].

A entrada do classificador é uma sequência $\{x_1, x_2, \dots, x_n\}$ de palavras (tokenizadas) e um conjunto de *tags* e a saída é uma sequência $\{y_1, y_2, \dots, y_n\}$ de *tags*, em

que cada saída y_i corresponde a uma entrada x_i . A *PoS Tagging* é uma tarefa de desambiguação: as palavras são ambíguas – podem ser associadas a mais de uma classe gramatical — e o objetivo é encontrar a *tag* correta para cada situação [54].

As técnicas de *PoS Tagging* são classificadas em técnicas baseadas em regras e técnicas baseadas em métodos estocásticos. As abordagens baseadas em regras geralmente utilizam o contexto da palavra, como as palavras ao seu redor, para determinar a classe apropriada para ela [95]. Classificadores baseados em regras usam um dicionário para obter possíveis *tags* para marcar cada palavra. Se a palavra tiver mais de uma *tag* possível, os classificadores baseados em regras usam regras escritas à mão para identificar a *tag* correta [55].

Os classificadores baseados em métodos estocásticos geralmente fornecem melhores resultados quando comparados aos classificadores baseados em regras. Qualquer modelo que, de alguma forma, incorpore métodos estatísticos pode ser rotulado como estocástico. Os classificadores estocásticos mais simples resolvem ambiguidades na classe gramatical de uma palavra com base apenas na probabilidade da ocorrência de uma palavra com uma *tag* específica. O problema com essa abordagem é que, embora possa gerar uma *tag* válida para uma determinada palavra, ela também pode gerar sequências de *tags* inapropriadas. Esta abordagem é chamada de “Abordagem de Frequência de Palavras”. Uma alternativa é calcular a probabilidade de ocorrência de uma determinada sequência de *tags*. Essa abordagem é melhor, pois considera as *tags* para palavras individuais com base no contexto. Esta abordagem é chamada de “Probabilidade de sequência de *tags*” [55].

2.1.4 Técnicas de Agrupamento

As técnicas de agrupamento (*clustering*), no contexto de modularização de software, são frequentemente consideradas métodos de aprendizado não supervisionado, cujo propósito é particionar uma população em grupos de elementos com características similares. A população é um conjunto de n elementos que podem ser clientes, produtos, lojas, agências, entre outros, descritos por m atributos [83]. Os algoritmos de

agrupamento dividem os dados em um número de *clusters* (grupos, subconjuntos ou categorias). Não existe uma definição universalmente aceita, mas a maioria dos pesquisadores descreve um *cluster* considerando a homogeneidade interna e a separação externa [127].

Segundo Jain [51], o procedimento de análise de *cluster* tem quatro etapas: seleção e extração de características, projeto ou seleção de algoritmos de agrupamento, validação de *clusters* e interpretação dos resultados. O objetivo final do agrupamento é fornecer aos usuários *insights* significativos sobre os dados, para que eles possam resolver os problemas encontrados [127]. Para Jain, o objetivo do agrupamento de dados é descobrir os agrupamentos naturais de um conjunto de padrões, pontos ou objetos.

Reunindo conceitos encontrados em diversos artigos [51, 13, 103, 127], podemos categorizar os métodos de agrupamento como:

- Métodos hierárquicos, que analisam o conjunto de dados e produzem uma estrutura representada por uma árvore binária, ou dendrograma, na qual a raiz representa todos os elementos em um único grupo e as folhas representam cada um dos objetos como um agrupamento unitário;
- Métodos particionais, que atribuem os objetos de um conjunto de dados a agrupamentos de maneira simultânea e sem uma estrutura hierárquica;
- Métodos baseados em grafos, que abordam cada elemento do conjunto de dados como um vértice e a medida de distância/similaridade para outros elementos é representada nas arestas criadas com outros vértices deste grafo; e
- Métodos baseados em densidade, onde valores podem ser representados de acordo com diversas distribuições de probabilidade. Desta maneira, métodos de agrupamentos baseados em misturas de densidade assumem que objetos podem ser derivados de diferentes distribuições probabilísticas ou de uma mesma distribuição com parâmetros diferentes e seu objetivo é identificar os agrupamentos e suas distribuições.

- Métodos Baseados em Redes Neurais, que utilizam redes neurais artificiais para agrupar um conjunto de dados recebido como entrada. Durante anos, esta categoria foi dominada por métodos baseados em Redes Neurais Competitivas.

2.1.5 Remodularização de Software

Software Module Clustering (SMC) é o problema de organizar módulos de software em estruturas maiores, semelhantes a contêineres, chamadas *clusters*. Os módulos precisam ser distribuídos em *clusters* porque grandes sistemas de software podem envolver centenas ou milhares de módulos e deve haver alguma organização de nível superior para ajudar os desenvolvedores a encontrar os módulos responsáveis por uma determinada tarefa executada pelo sistema [87].

De maneira formal, dados um conjunto $N = 1, \dots, n$ de módulos e custo $c(i, j)$ indicando a relação entre cada par $i, j \in N$, o problema SMC consiste em encontrar uma partição $P = \{C_1, C_2, \dots, C_m\}$ de N em grupos de m , $1 \leq m \leq n$, com $C_i \neq \emptyset, i = 1, \dots, m; C_i \cap C_j = \emptyset, i, j = 1, \dots, m, i \neq j$ e $\bigcup_{i=1}^m C_i = N$, de modo a maximizar ou minimizar algum critério [87]. Mancoridis et al. [35] foram os primeiros a apresentar um estudo baseado em busca para o problema SMC.

Monçores [87] define o problema SMC como um problema de partição de grafos onde as dependências entre os módulos que compõem um sistema de software podem ser modeladas como um grafo denominado *Module Dependency Graph* (MDG) [74]. Este grafo é a entrada para o problema SMC, que pode então ser visto como um problema de partição do grafo. O MDG é um grafo direcionado onde os módulos são representados por nós, as dependências entre os módulos são representadas por arestas entre esses nós e cada *cluster* é representado por uma partição do grafo [87].

Abordagens estruturais utilizam algoritmos de agrupamento baseados nas dependências existentes entre os componentes de um projeto de software. As abordagens baseadas no léxico partem da ideia de que as informações léxicas fornecidas pelos desenvolvedores no código-fonte representam uma fonte importante de informações para a remodularização do software. Abordagens mistas são baseadas na combinação de informações léxicas e estruturais para apoiar tarefas de compreensão dentro da

manutenção e reengenharia de sistemas de software. Aqui, a dimensão lexical considera os domínios do problema e do desenvolvimento, e a dimensão estrutural se refere à estrutura sintática do programa, junto com o controle e fluxo de dados que ele representa.

A qualidade da distribuição dos módulos é frequentemente avaliada por duas métricas: acoplamento e coesão [128]. O acoplamento mede a dependência entre *clusters*, enquanto a coesão mede a força interna de um *cluster*. A literatura técnica sobre projeto de software sugere que uma distribuição de módulos deve ser considerada uma boa decomposição para um sistema de software se apresentar baixo acoplamento e alta coesão [12]. Essas métricas são calculadas a partir das dependências observadas entre os módulos. Diz-se que um módulo *A* depende de um módulo *B* se requer alguma função, procedimento ou definição declarada no módulo *B* para desempenhar suas funções.

2.2 Trabalhos Relacionados

2.2.1 Remodularização baseada na Estrutura do Software

Wiggerts [120] analisou algoritmos comumente utilizados para agrupar artefatos de software, fornecendo uma visão geral dos métodos de agrupamento e uma base teórica para aplicá-los na remodularização de software. Este trabalho fez parte de um projeto maior chamado “*Resolver*”, cujo objetivo era definir e implementar uma espécie de “*fábrica de renovação de sistemas*” baseada em pesquisas científicas. Eles apontaram técnicas de remodularização existentes usando ou reinventando algoritmos de agrupamento e concluíram que a remodularização de software poderia se beneficiar mais da teoria geral sobre a análise de *clusters*.

Mitchell e Mancoridis [73] utilizaram uma técnica automática que trata o agrupamento como um problema de otimização que pode ser resolvido usando algoritmos meta-heurísticos, como *hill-climbing* e algoritmos genéticos. A técnica foi implementada em uma ferramenta chamada *Bunch* [74, 35], criada para decompôr a estrutura de sistemas de software em subsistemas funcionais. Mesmo considerando que o resultado da avaliação inicial do *Bunch* tenha sido relevante, os autores decidiram realizar uma nova rodada de avaliação usando um maior número de sistemas e uma

maior diversidade de tamanhos em termos de número de classes. Esta segunda rodada confirmou que a ferramenta *Bunch* foi capaz de produzir bons resultados com eficiência, mas também expôs algumas áreas de melhoria que não estavam aparentes na primeira avaliação. *Bunch* foi projetada para ser estendida e incluir outros algoritmos de agrupamento e funções de *fitness* chamadas *Modularization Quality* (MQ).

Mais tarde, os autores apresentaram um estudo de caso para demonstrar como a ferramenta *Bunch* pode ser usada para criar visualizações da estrutura de sistemas de software [86]. Para medir a similaridade dos resultados de agrupamento individuais com a decomposição de referência, usaram duas medidas de similaridade diferentes: *EdgeSim* e *MeCl* [88]. A medida de similaridade *EdgeSim* normaliza o número de arestas *intra* e *intercluster* que concordam entre duas partições. Já a medida de similaridade de *MeCl* determina a distância entre um par de partições, calculando primeiro os maiores subconjuntos de módulos de ambas as partições que concordam em relação à sua colocação em *clusters*. Para ilustrar a abordagem para avaliar os resultados do agrupamento *Bunch*, os autores analisaram a biblioteca de classes Java Swing ⁷, que consiste em 413 classes que possuem 1.513 dependências entre elas.

Tzerpos e Holt [116] estudaram uma série de algoritmos de remodelarização de software e compararam o seu desempenho. O trabalho se concentrou na questão da estabilidade. Os autores definiram a estabilidade de um agrupamento formalmente e explicaram por quê é uma propriedade essencial para um algoritmo de agrupamento, fornecendo resultados experimentais da avaliação da estabilidade que comparou vários algoritmos de agrupamento apresentados na literatura. Os autores definiram um algoritmo de remodelarização de software como estável se os *clusters* obtidos não forem muito afetados por pequenas modificações em sua entrada, ou seja, pequenas alterações no sistema de software em questão. Para incluir todos os tipos de algoritmos, foi adotada a métrica *MoJo* [114], que mede a distância entre dois agrupamentos diferentes de um mesmo conjunto de características. Foram usados dois grandes sistemas de tamanho comparável, mas com filosofias de desenvolvimento diferentes: o *TOBEY*, um sistema

⁷ Javasoft Swing Libraries: Java Foundation Classes - <http://www.javasoft.com/products/jfc>

industrial proprietário e que serve como *back-end* de otimização para vários produtos de compilador da IBM, e a versão 2.0.27a do Linux. Os algoritmos usados para comparação foram cinco versões de algoritmos de agrupamento hierárquico que utilizam o coeficiente de *Jaccard* e o algoritmo desenvolvido. Os resultados indicaram que a métrica utilizada pode ser um bom preditor da estabilidade de um algoritmo de agrupamento.

Lutz [68] descreveu um sistema evolutivo, baseado em um algoritmo genético, que encontra boas decomposições hierárquicas para sistemas complexos. O trabalho envolveu o desenvolvimento de novos operadores de mutação e cruzamento (*crossover*) específicos de domínio. O autor também descreveu uma função para medir a complexidade de uma decomposição modular hierárquica (*hierarchical modular decomposition* - HMD) com base no princípio de comprimento mínimo de descrição. Esta função serve como função de *fitness* e orienta o processo evolutivo e foi implementada em um programa Java.

Lutz buscou responder a uma pergunta de pesquisa fundamental sobre a remodularização de software: o que torna uma HMD de um sistema melhor do que outra? Para a criação da função *fitness*, foi utilizada uma variante de uma métrica da teoria da informação de Wood [123] para definir quando se deve considerar uma HMD melhor que outra. Ao contrário da medida original de Wood, a nova medida é sensível à direcionalidade das dependências entre classes, dando-lhe uma aplicabilidade maior no contexto do problema de remodularização de software. Em alguns experimentos, a HMD encontrada pelo sistema desenvolvido mostrou ser muito melhor do que a HMD encontrada pelo sistema de Wood [123].

Mahdavi et al. [71] abordaram a remodularização de software como um problema de otimização, concentrando os esforços em abordagens baseadas em busca. Os autores identificaram que a abordagem de busca local conhecida como *Hill Climbing* foi a mais bem-sucedida. O algoritmo recebe um MDG como entrada, onde cada MDG contém uma lista de informações “de-para” para os módulos dentro do sistema a serem agrupados. Essas informações são convertidas em um vetor de conexões ponderadas entre nós. O peso é definido como “um” se o valor do peso de uma conexão no MDG não for especificado. O

objetivo é chegar a uma partição de grafos em que cada agrupamento maximize o número de arestas internas e minimize o número de arestas externas. Em termos de Engenharia de Software, isso corresponde a coesão máxima e acoplamento mínimo. Na abordagem, foi utilizada a função de *fitness* “*Basic MQ*” conforme usada pelos autores da *Bunch* [73] que captura a métrica de "acoplamento mínimo/coesão máxima". Os resultados mostraram que a técnica de *Hill Climbing* múltipla produziu melhores resultados em todos os MDG, ponderados e não ponderados.

Wu et al. [125] compararam seis algoritmos de agrupamento, aplicando-os a versões subsequentes de cinco grandes sistemas de código aberto. O objetivo era esclarecer até que ponto os algoritmos podem ser utilizados para apoiar a modularização e a documentação arquitetural durante o ciclo de vida de um grande sistema. Os algoritmos estudados foram o agrupamento aglomerativo com corte em 0,75 e 0,90, com as medidas de dissimilaridade utilizando a distância euclidiana entre os vetores de atributos *single-linkage* e *complete-linkage*, *ACDC* [115] e *Bunch*. O método *single linkage* (ou do vizinho mais próximo) baseia-se na distância entre os pontos de cada cluster mais próximos entre si e o método *complete linkage* baseia-se na distância entre os pontos de cada cluster mais distantes entre si [22].

Os autores definiram três critérios para análise: *stability*, que enfatiza a persistência da estrutura entre agrupamento de versões sucessivas de um sistema em evolução; *authoritativeness*, uma medida de quão próximo a partição resultante se assemelha a uma visão lógica criada por um especialista; e *extremity of cluster distribution*, que parte do princípio que a distribuição de tamanho de um *cluster* não deve exibir valores extremos. Para avaliar quantitativamente a última métrica, os autores definiram uma medida chamada *NED* (*non-extreme distribution*). Neste experimento, assumiram que quaisquer *clusters* de tamanho menor do que cinco ou maior que 100 seriam considerados *clusters* extremos. Os autores concluíram que um algoritmo estável pode não produzir agrupamentos significativos. Para ser significativo, o algoritmo deve produzir agrupamentos que possam ajudar os desenvolvedores a entender o sistema.

Maqbool e Babri [78] realizaram uma revisão das técnicas de agrupamento hierárquico para recuperação de arquitetura e modularização de sistemas de software, por meio de abordagens com avaliação experimental. O foco do trabalho foi a análise de medidas e algoritmos de agrupamento hierárquico no domínio de software e a identificação de seus pontos fortes e fracos neste domínio, para que possam ser utilizados de forma eficaz para a recuperação da arquitetura. Para esta avaliação, analisaram o comportamento de várias medidas de similaridade e distância no contexto do software, identificando famílias de medidas de similaridade e distância, como o coeficiente de Jaccard, coeficiente de Sorenson-Dice, distância de Bray-Curtis, Distância de Chord, distância de Hellinger, Distância Euclideana, coeficiente Simple matching, Rogers-Tanimoto e Gower-Legendre. Nesta análise, mostraram que todos os membros de uma família darão os mesmos resultados de agrupamento. Os seis algoritmos de agrupamento hierárquico foram *Single Linkage* (SLA), *Complete Linkage* (CLA), *Weighted Average Linkage* (WLA) e *Unweighted Average Linkage* (ULA), *Combined Algorithm* (CA) e mais dois algoritmos *Weighted Combined Algorithm* (WCA) [77] e o LIMBO [3] que os autores se referem como uma nova classe de formação de vetores de características, chamando de *New Feature Vector formation* (NFV).

Os experimentos foram realizados em quatro sistemas de software legados, fornecendo uma visão do comportamento dos vários algoritmos de agrupamento. Os resultados experimentais mostraram que a abordagem adotada pelos algoritmos da classe NFV (WCA/LIMBO) reduz substancialmente as decisões arbitrárias de agrupamento, que são comuns durante o processo de agrupamento hierárquico no domínio de software, produzindo melhores resultados. Uma conclusão importante que os autores destacaram é que a qualidade dos resultados depende não apenas do algoritmo e da medida de similaridade, mas também das características do sistema de software ao qual o algoritmo foi aplicado. Por exemplo, o NFV, quando usado com a medida de perda de informação (LIMBO), é capaz de identificar funções utilitárias que não acessam recursos. Os autores definem “utilitárias” como funções que podem não depender umas das outras, mas são

usadas em muitas partes de um sistema. Uma característica importante das utilitárias é que elas possuem conjuntos de recursos semelhantes.

Bittencourt e Guerrero [16] apresentam um estudo para avaliar algoritmos de agrupamento em *design* extraído do código-fonte por meio de análise estática, avaliando quatro algoritmos: *edge betweenness clustering*, *k-means clustering*, *modularization quality clustering* [74] e *design structure matrix clustering*. Para medir a similaridade entre diferentes agrupamentos, utilizaram uma medida de qualidade relativa chamada *MoJoSim*, derivada da medida MoJo apresentada por Tserpos et al. [114]. As dimensões de avaliação utilizadas neste trabalho foram os critérios definidos por Wu e Holt [125]: *authoritativeness*, *stability* e *extremity*. Os resultados mostraram que o algoritmo *k-means* obteve o melhor desempenho em termos de *authoritativeness* e *extremity* e que o algoritmo *modularization quality clustering* [74] produziu *clusters* mais estáveis. Os autores também apontaram que as técnicas de agrupamento automatizadas não podem recuperar sozinhas as arquiteturas de sistemas, mas podem fornecer um primeiro passo para acelerar o processo de recuperação de arquitetura assistido por especialistas.

Praditwong et al. [96] introduziram uma abordagem multi-objetivo para a remodularização de software com duas formulações com múltiplos objetivos. Eles compararam os resultados obtidos com os da formulação mono-objetivo em 17 problemas de agrupamento do mundo real. Os resultados indicaram que a abordagem ECA (*Equal-size Cluster Approach*) é capaz de produzir soluções melhores do que a solução mono-objetivo existente, tanto em termos dos múltiplos objetivos que pretende satisfazer como em termos do objetivo único sobre o qual todos os trabalhos anteriores se concentraram. Barros [11] usou otimização multi-objetivo para resolver o problema de remodularização de software, apresentando um estudo abordando a eficiência e a eficácia da adição de objetivos compostos na busca. O autor empregou a formulação ECA como base para comparar os resultados obtidos por um algoritmo genético multi-objetivo em três configurações. O estudo mostrou evidências de que suprimir a função de *fitness* MQ do processo de busca pode ajudar a encontrar soluções em menos tempo, com taxa de erro aprimorada e menor distância geracional, principalmente para grandes instâncias.

Semaan et al. [101] utilizaram uma heurística baseada na meta-heurística *Iterated Local Search* (ILS) para resolver o problema de remodularização de software. Neste contexto, o software é representado como um grafo MDG ponderado e o processo de busca utiliza a função de *fitness* MQ [35]. Os resultados computacionais mostraram que a utilização do algoritmo proposto é uma alternativa eficiente para a resolução do problema de remodularização de software de forma automática, onde a heurística ILS alcançou resultados superiores e exigiu um reduzido tempo de execução.

Seguindo os resultados de Semaan et al. [101], Pinto et al. [38] apresentaram uma heurística baseada em *Iterated Local Search* para o problema de remodularização de software. Um extenso experimento foi conduzido para encontrar as melhores escolhas para a solução inicial e os componentes de perturbação do algoritmo proposto. Outro estudo comparou o melhor ILS com várias configurações de algoritmos genéticos. Os autores mostraram que, embora a meta-heurística ILS não seja amplamente utilizada para resolver problemas de Engenharia de Software, ela se mostrou eficaz para o problema selecionado, superando a melhor configuração do algoritmo genético em 24 das 40 instâncias analisadas e usando menos tempo de processamento computacional.

Monçores et al. [87] apresentaram uma heurística baseada em um algoritmo construtivo e *Large Neighborhood Search* para abordar o problema de remodularização de software. Também realizaram o maior estudo experimental abordando esse problema até o momento da publicação, envolvendo 124 instâncias de tamanho e complexidade variados e comparando o algoritmo proposto com a heurística que encontrou os melhores resultados para o problema até o momento. Eles mostraram que o algoritmo proposto superou a heurística ILS em 93 de 124 instâncias com nível de confiança de 95%.

2.2.2 Remodularização baseada em Análise Léxica

Kuhn et al. [62] utilizaram *Latent Semantic Indexing* (LSI), uma técnica de recuperação de informação que localiza tópicos linguísticos em um conjunto de documentos [31, 79], para calcular a semelhança linguística entre os artefatos de software, agrupando estes artefatos de acordo com a sua semelhança. Esse agrupamento

divide o sistema em tópicos linguísticos, cada qual representando grupos de documentos que utilizam vocabulário semelhante. Para identificar como os *clusters* estão relacionados entre si, os autores utilizaram uma matriz de correlação. Eles desenvolveram uma ferramenta chamada *Hapax* para realizar esse trabalho.

A abordagem depende de vários parâmetros, como dimensionalidade do espaço LSI, a quebra de um dendrograma em *clusters*, a detecção de *links* semânticos e a ponderação da matriz termo-documento ⁸. O esquema de ponderação mais comum é baseado em TF-IDF, que os autores também usaram nos estudos de caso. Outros exemplos de esquemas de ponderação são a ponderação por entropia e a ponderação logarítmica [36]. Ao avaliar a abordagem, os autores observaram que a escolha do esquema de ponderação tem um efeito considerável sobre os valores de similaridade: dependendo da escolha do esquema de ponderação, os limites de similaridade podem diferir significativamente. Como regra geral, usar ponderação logarítmica, com limite de similaridade de 0,75, é equivalente a um limite de 0,5 com ponderação TF-IDF.

Uma das hipóteses dos autores era que o agrupamento semântico revelaria uma semântica de domínio de sistemas. Porém, os experimentos mostraram que a maioria dos tópicos linguísticos estava relacionada a conceitos de aplicação ou componentes de arquitetura. Os autores concluíram que os resultados não são generalizáveis para diferentes sistemas de software. Uma boa convenção de nomenclatura e identificadores bem escolhidos geram melhores resultados, enquanto a nomenclatura ruim é uma das principais ameaças à validação externa das conclusões a que chegaram.

Corazza et al. [26] exploram apenas informações léxicas extraídas do código-fonte para identificar subsistemas de software. Os autores extraíram informações de quatro regiões (chamadas de zonas) no código-fonte de sistemas Java: comentários, JavaDocs, identificadores de classes/métodos e identificadores de variáveis. As informações fornecidas por cada zona são ponderadas utilizando o indicador TF-IDF, por meio de um modelo probabilístico e do algoritmo *Expectation-Maximization* (EM).

⁸ para equilibrar a influência de termos muito raros e muito comuns é comum ponderar os valores de ocorrência na recuperação de informações

O algoritmo EM é um método iterativo usado em *machine learning*, não supervisionado, para estimar parâmetros desconhecidos em modelos estatísticos. Em seguida, os dados são agrupados por uma versão modificada do algoritmo K-Medoids [76] para lidar melhor com o domínio. A avaliação da qualidade do processo considerou dois critérios definidos por Wu et al. [125]: *authoritativeness* e *non-extremity cluster distribution* (NED). Para avaliar a *authoritativeness* foi utilizada a métrica *MoJoSim*, definida por Bittencourt e Guerrero [16]. Os resultados mostraram que a versão modificada do *K-Medoids* tem um desempenho bom em termos de NED, melhor do que qualquer outro algoritmo de agrupamento considerado.

Risi et al. [100] apresentaram uma abordagem automática para particionar sistemas de software em subsistemas usando *Latent Semantic Indexing* para a fase de análise e cálculo de similaridade entre artefatos de software. Para o agrupamento dos artefatos, utilizaram o algoritmo *k-means*. Para avaliar o desempenho dos algoritmos, os autores utilizaram a métrica *MoJoSim* [16]. Como resultados, os autores apontaram para duas características significativas na abordagem. A primeira é que ela pode ser usada automaticamente para recuperar a visão arquitetônica de um sistema de software. A segunda é que ela pode ser aplicada em sistemas de software implementados em qualquer linguagem de programação orientada a objeto. Porém, observaram que para cada versão dos sistemas considerados há uma diferença entre o número de *clusters* identificados automaticamente e o número de grupos dentro da partição autoritativa (ou seja, a partição proposta pelos desenvolvedores do software).

Corazza et al. [27] analisaram a relevância do léxico extraído de cada uma de seis regiões (zonas) na remodularização de software. As seis regiões analisadas são os nomes das classes, os nomes de atributos, os nomes de métodos, os nomes de parâmetros, os comentários e as declarações do código-fonte. A intenção era obter uma visão mais profunda sobre a contribuição das informações léxicas com o objetivo de agrupar arquivos de código-fonte que implementam funcionalidades relacionadas. Após a vetorização, criada com TF-IDF, os valores dos pesos associados às zonas são estimados

por meio de um modelo probabilístico, onde as probabilidades são estimadas pelo algoritmo *Expectation-Maximization* (EM).

Para agrupar arquivos de código-fonte, foi utilizado o algoritmo *Hierarchical Agglomerative Clustering* (HAC) com uma estratégia de corte *ad-hoc*, onde foi aplicada uma modificação dos *k-Medoids*. Para calcular a métrica *authoritativeness*, os autores usaram uma medida baseada no *MoJo* [114] e *MoJoSim* [16]. Os autores chegaram à conclusão de que considerar as seis zonas como fontes distintas de informação, ao invés de pensar no vocabulário do código-fonte como um único conjunto de todos os lexemas, sempre fornece melhores resultados em termos de *authoritativeness*. Os autores também concluíram que cada projeto tem suas peculiaridades quanto à distribuição e relevância dos termos nas diferentes zonas. Assim, é necessário um mecanismo que pondere automaticamente a importância de cada zona para obter os melhores resultados em termos de agrupamento dos arquivos de código-fonte. A introdução de uma técnica de ponderação sempre melhora os resultados: usar a informação léxica de forma plana, sem levar em consideração as zonas das quais os termos foram extraídos, não é adequado para fins de modularização de software. A solução léxica não é apenas comparável à estrutural, mas também é capaz de fornecer resultados mais “previsíveis”.

Corazza et al. [28] propuseram uma abordagem para a modularização de software que reúne artefatos de código-fonte relacionados com base na informação léxica fornecida por desenvolvedores. A proposta estima a relevância de diferentes zonas de código-fonte (nomes de classes, nomes de atributos, nomes de métodos, nomes de parâmetros, comentários e comandos) por meio de um modelo probabilístico da distribuição do léxico, cujos parâmetros são estimados automaticamente pelo algoritmo *Expectation-Maximization*, que fornece os pesos das zonas.

Os pesos das zonas são usados como multiplicadores no *Vector Space Model* (VSM). Essa similaridade é usada para agrupar classes por meio da customização de um algoritmo de agrupamento *k-Medoids*. No contexto da modularização de software, a medida de similaridade *MojoFM* [131] foi usada para superar desvantagens da medida *MoJo* [114]. Os autores observaram que o uso de zonas melhora o agrupamento em

relação ao uso de um vocabulário plano e o uso de pesos calculados pela aplicação de *Maximum Likelihood Estimation* (MLE) e EM melhora a qualidade do agrupamento. Uma observação constante em todos os trabalhos é que a qualidade das partições melhora se os desenvolvedores escolhem bons identificadores e comentam o código-fonte.

Todos os trabalhos do grupo da Corazza [26, 27, 28] exploram informações léxicas extraídas do código-fonte para identificar subsistemas de software, extraindo informações de regiões (chamadas de zonas) no código-fonte de sistemas Java como comentários, JavaDocs, identificadores de classes, métodos e variáveis. A avaliação da qualidade do processo considera dois critérios – *authoritativeness* e NED – definidos por Wu et al. [125]. Para avaliar o critério de *authoritativeness* utilizaram variações da métrica MoJo [114]. As informações fornecidas por cada zona foram ponderadas utilizando o indicador TF-IDF por meio de um modelo probabilístico e o algoritmo EM. Os dados foram agrupados por uma versão modificada do algoritmo K-Medoids, *k-means* e HAC. Em geral, observaram que o uso de zonas melhora o agrupamento em relação ao uso de um vocabulário plano e o uso de pesos calculados pelo algoritmo EM melhora a qualidade do agrupamento. Os resultados mostraram uma melhoria no agrupamento com o algoritmo HAC. Uma observação constante em todos os trabalhos é que a qualidade das partições melhora se os desenvolvedores escolhem bons identificadores e comentam corretamente o código-fonte.

2.2.3 Remodularização baseada em Informações Estruturais e Léxicas

Maletic e Marcus [72] apresentaram PROCSSI, um sistema que utiliza *Latent Semantic Indexing* (LSI) para identificar semelhanças semânticas entre partes do código-fonte. Essa medida de similaridade semântica foi usada para agrupar componentes de software. PROCSSI utiliza uma abordagem teórica de grafos para definir as métricas que serão utilizadas na tarefa de compreensão de programas. Segundo os autores, a compreensão de um programa é uma tarefa complexa, na qual o engenheiro de software deve examinar tanto o aspecto estrutural do código-fonte quanto o domínio

do problema para extrair as informações necessárias para entender o sistema de software. Para abordar esses dois aspectos, investigaram uma série de ferramentas e métodos.

Os resultados mostraram que a técnica LSI captura porções significativas do significado não apenas de palavras individuais, mas também de passagens inteiras, como sentenças, parágrafos e redações curtas. Os experimentos com o PROCSSI mostraram que a similaridade semântica dos documentos de código-fonte fornece informações que podem ser utilizadas nas tarefas de manutenção e evolução de software. Uma das críticas à técnica LSI, quando aplicada a textos em linguagem natural, é que ela não considera a ordem das palavras, relações sintáticas ou morfologia, ou seja, não utiliza uma gramática ou um vocabulário predefinido.

Andritsos e Tzerpos [3] introduziram uma nova abordagem de remodularização de software baseada na minimização da perda de informação durante o processo de agrupamento. O trabalho formula o problema de remodularização de software dentro do *framework Information Bottleneck*, apresentando a ferramenta *LIMBO*. Para comparação, os autores experimentaram com quatro algoritmos diferentes: *single linkage* (SL), *complete linkage* (CL), *weighted average linkage* (WA) e *unweighted average linkage* (UA). Para comparar a saída dos algoritmos com a decomposição autoritativa (proposta pelos desenvolvedores do software), usaram a medida de distância *MoJo* [114], a medida *Koschke-Eisenbarth* [59] e *EdgeSim* [85].

Os resultados do trabalho confirmaram a hipótese de que a estrutura de diretórios e as informações de propriedade ⁹ são fatores importantes para o processo de remodularização de software, enquanto as linhas de código não são. Porém, não há garantia de que os desenvolvedores de um software legado tenham seguido princípios de Engenharia de Software, como alta coesão e baixo acoplamento. Também não ficou claro quais tipos de atributos não estruturais são apropriados para inclusão em uma abordagem de remodularização de software.

⁹ Informações de propriedade mostram, neste contexto, quem foi o desenvolvedor que implementou cada módulo do software.

Scanniello et al. [105] apresentaram uma abordagem para identificar módulos visando a remodularização de um software e o entendimento de sistemas de software orientados a objeto. A abordagem foi concebida para recuperar arquiteturas hierárquicas de sistemas de software orientados a objeto, onde os módulos identificados dependem apenas dos módulos das camadas de nível inferior. A abordagem foi baseada na combinação de dimensões estruturais e semânticas de sistemas de software. A dimensão estrutural foi usada para decompor os artefatos de software em camadas (decomposição horizontal), usando um algoritmo de análise de *links* conhecido e amplamente empregado [64] para identificar camadas de software a partir da estrutura estática do sistema. A dimensão semântica foi utilizada para particionar cada camada (decomposição vertical), utilizando informações léxicas extraídas do código-fonte para identificar a similaridade entre pares de classes (usando VSM para calcular a dissimilaridade entre um par de classes) e particionando cada camada em módulos usando o algoritmo de agrupamento *k-means*. A abordagem mostrou bom desempenho em termos de *authoritativeness*, melhor do que os outros algoritmos de agrupamento com os quais foi comparado.

Prajapati et al. [?] propuseram uma abordagem considerando vários tipos de dependências estruturais e lexicais, para remodularizar sistemas Orientados a Objetos (OO). O objetivo principal era gerar soluções de remodularização que pudessem refletir a perspectiva dos desenvolvedores de remodularização. O método proposto calcula a força de acoplamento entre classes usando diferentes pesos em termos de vários mecanismos de dependências estruturais e lexicais. O problema de remodularização de software é então formulado como um problema de otimização de objetivo único e multiobjetivo usando Algoritmos Genéticos (AG). Com base nos diferentes tipos de dependências estruturais e lexicais, foram projetados três esquemas de acoplamento: baseados em estruturas, baseados em léxicos e baseados na combinação de estruturas e léxicos.

Os resultados indicaram que as variantes ponderadas baseadas em TF-IDF (ou seja, STFIDFW, LTFIDFW e SLTFIDFW) de cada uma das três categorias amplas superaram o restante das variantes dentro de cada categoria. Onde UW - *Un-Weighted*, W -

Weighted, *S - Structural*, *L - Lexical* e *SL* – ambos. No entanto, a variante ponderada SLTFIDFW superou todas as outras. Os autores concluíram que a abordagem léxico-estrutural combinada (SLTFIDFW) considerando vários tipos de dependências junto com seus pesos relativos tem um bom desempenho e resulta em melhor remodularização em comparação com o restante das alternativas consideradas.

2.3 Considerações Finais

Na remodularização baseada na estrutura do software, alguns autores utilizaram técnicas que tratam o agrupamento como um problema de otimização que pode ser resolvido usando algoritmos meta-heurísticos, como *Hill Climbing* e algoritmos genéticos. Alguns autores utilizaram outras meta-heurísticas, como ILS e LNS, para resolver o problema de remodularização de software onde o software é representado como um grafo MDG ponderado. Em geral, os autores concluíram que um algoritmo pode não produzir agrupamentos significativos. Para ser significativo, o algoritmo deve produzir grupos que ajudem os desenvolvedores a entender o sistema. Uma conclusão importante que os autores destacaram é que a qualidade dos resultados depende não apenas do algoritmo e da medida de similaridade, mas também das características do sistema de software ao qual o algoritmo foi aplicado. Os autores também apontaram que as técnicas de agrupamento automatizadas não podem recuperar sozinhas a arquitetura de um sistema, mas podem fornecer um primeiro passo para acelerar o processo de recuperação de arquitetura assistido por especialistas.

Na remodularização baseada em análise léxica, todos os autores exploraram informações léxicas extraídas do código-fonte para identificar subsistemas de software. Alguns autores usaram LSI para calcular a semelhança linguística, agrupando artefatos de software de acordo com a sua semelhança. Uma das críticas ao método LSI é que ele não faz uso da ordem das palavras, relações sintáticas ou morfologia. Todos os autores utilizaram um esquema de ponderação baseado em TF-IDF. Em geral, os autores concluíram que os resultados não são generalizáveis para outros sistemas de software. Uma boa convenção de nomenclatura e identificadores gera melhores resultados,

enquanto a nomenclatura ruim é uma das principais ameaças à validação externa. Também concluíram que cada projeto tem suas peculiaridades quanto à distribuição e relevância das categorias dos termos do código-fonte processados: usar apenas a informação léxica de forma plana, sem levar em consideração as categorias das quais os termos foram extraídos, não é eficiente para fins de remodularização de software. Uma observação constante em todos os trabalhos é que a qualidade das partições melhora se os desenvolvedores escolhem bons identificadores e comentam o código-fonte.

Na remodularização baseada em informações estruturais e léxicas, alguns autores confirmaram que a estrutura de diretórios e as informações dos arquivos são importantes para a remodularização, enquanto as linhas de código têm menor importância. Em comum ressaltam que não há garantia de que os desenvolvedores de um sistema legado tenham seguido princípios de Engenharia de Software, como alta coesão e baixo acoplamento, e não ficou claro quais tipos de atributos não estruturais são apropriados para inclusão em uma abordagem de remodularização de software. Os algoritmos usados para comparação foram versões do algoritmo *k-means* e de algoritmos de agrupamento hierárquico (SLA, CLA, WLA, ULA, CA). Muitos autores usaram os três critérios descritos por Wu et al. [125] para análise: estabilidade, *authoritativeness*, e *extremity of cluster distribution*. Como métrica para avaliação, a maioria adotou a métrica MoJo e suas variantes (como MoJoSim e MoJoFM), que medem a distância entre dois agrupamentos diferentes de um mesmo conjunto de características.

Todos os trabalhos utilizaram TF-IDF para avaliar a relevância de representações de *strings* (palavras, sentenças, lemas, etc.) em um documento entre uma coleção de documentos (*corpus*). No entanto, esse método tem algumas limitações. Os vetores obtidos terão o mesmo tamanho do vocabulário, o que pode ser um problema quando os vocabulários forem muito grandes. Vetores para grandes documentos seriam de tamanhos grandes e com muitos valores nulos, levando à criação de vetores esparsos. BoW não traz nenhuma informação sobre o significado do texto e a vetorização TF-IDF não ajuda a trazer o significado contextual das palavras, pois é apenas baseada na

frequência. Esses métodos não conseguem captar significado entre palavras, isto é, extraem pouca informação semântica e sintática dos textos.

A partir do desenvolvimento do *Word2vec* em 2013, como uma resposta para tornar o treinamento de incorporações baseadas em redes neurais mais eficiente, é possível notar melhorias no pré-processamento de dados em formato textual para realizar tarefas relacionadas ao processamento de linguagem natural, suprimindo as deficiências das técnicas anteriores (BoW e TF-IDF). No entanto, nenhuma abordagem anteriormente reportada na literatura utilizou modelos de incorporação de palavras para orientar a remodularização de projetos de software. Assim, este é o primeiro trabalho a focar no estudo de modelos de incorporação de palavras na remodularização de software.

3 REMODULARIZAÇÃO DE SOFTWARE E MODELOS DE INCORPORAÇÃO DE PALAVRAS PRÉ-TREINADOS

Resumo. O objetivo deste capítulo é comparar a remodularização de software obtida por meio de aplicação de algoritmos baseados em informações estruturais do projeto do software com a remodularização baseada em informações léxicas e modelos de incorporação de palavras utilizando os nomes de classes e outras regiões do código-fonte.

3.1 Introdução

Neste capítulo apresentamos estudos experimentais relacionados com a remodularização de software baseada em informações léxicas, utilizando diferentes modelos de incorporação de palavras e termos provenientes de diversas regiões do código-fonte. O objetivo é entender se os modelos de incorporação de palavras, uma representação de termos linguísticos mais recente do que as utilizadas nos trabalhos relacionados, traz benefícios para a remodularização de software.

Os experimentos reportados têm como foco a geração de distribuições de classes em pacotes por diferentes algoritmos e a comparação dos resultados da remodularização com a distribuição original de classes utilizada pelos desenvolvedores. Cada distribuição de classes em pacotes será entendida como um agrupamento, dado que as classes são associadas aos pacotes (grupos). O agrupamento chamado de “DEV” descreve a distribuição de classes em pacotes definida pelos desenvolvedores de um software e observada a partir da análise do código-fonte. Esta distribuição será comparada com as distribuições geradas por algoritmos determinísticos baseados em informações léxicas e com as distribuições geradas por um algoritmo meta-heurístico baseado nas dependências entre classes.

Para avaliar a qualidade de um agrupamento, utilizamos a métrica MoJoFM [132]. Esta métrica é uma modificação da métrica Mojo para calcular a distância entre dois agrupamentos, definida por Tzerpos e Holt [114], como:

A distância MoJo entre dois grupos A e B do mesmo sistema de software é definida como o número mínimo de operações "Move" ou "Join" que uma pessoa precisa realizar para transformar A em B ou vice-versa. Quanto menor a distância MoJo entre uma decomposição A criada automaticamente e a decomposição "padrão ouro" B, mais eficaz será o algoritmo que criou A.

A métrica $MoJoFM(A,B)$ foi desenvolvida para superar deficiências da métrica MoJo. Ela mede a distância entre dois agrupamentos, A e B, calculando o número de módulos que devem ser movidos de um grupo para outro e o número de grupos que devem ser mesclados para transformar o agrupamento A em B. Este valor varia de 0 a 1, onde um número maior denota duas soluções que representam aproximadamente o mesmo agrupamento e um número próximo de zero representa duas soluções que são muito diferentes uma da outra [87]. Esta métrica foi escolhida por estar presente nos trabalhos relacionados no capítulo anterior ou representar uma evolução da métrica utilizada nestes trabalhos. Como é uma medida normalizada, os resultados podem ser comparados a projetos com poucas ou muitas classes.

3.2 Algoritmos de Agrupamento

Nos estudos reportados neste capítulo utilizamos diferentes algoritmos de agrupamento para gerar distribuições de classes em pacotes com base em informações estruturais e informações léxicas do software. Nesta seção, descrevemos os algoritmos utilizados para criar estas distribuições de classes em pacotes.

3.2.1 Iterated Local Search

Segundo Glover e Kochenberger [44], meta-heurísticas são métodos que coordenam procedimentos de busca local com estratégias de alto nível, de modo a criar um processo capaz de escapar de mínimos locais e realizar uma busca robusta no espaço de soluções de um problema. Sorensen [112] define meta-heurística como uma estrutura algorítmica independente de problemas de alto nível, que fornece um conjunto de diretrizes ou estratégias para desenvolver algoritmos de otimização heurística. Esta

definição passou a abranger quaisquer procedimentos que empreguem estratégias para escapar de mínimos locais em espaços de busca de soluções complexas. As meta-heurísticas são geralmente aplicadas a problemas para os quais não se conhece algoritmo eficiente para encontrar suas soluções.

A meta-heurística *Iterated Local Search* (ILS), proposta por Lourenço et al. [67], é um algoritmo que explora o espaço de soluções por meio de perturbações em ótimos locais aplicando uma busca local sobre uma solução inicial. Quando a busca atinge um ótimo local, é aplicado um método de perturbação sobre esta solução e a busca local é reiniciada a partir da solução modificada pela perturbação [38]. A meta-heurística do ILS é baseada em uma estratégia de busca local que evolui uma solução por vez ao longo do processo iterativo. Deste modo, o ILS busca obter soluções de melhor qualidade se comparadas com as soluções geradas por repetições aleatórias da busca local [38].

Normalmente começa a partir de uma solução arbitrária e, a cada passo, uma nova solução é escolhida na vizinhança da atual. A vizinhança da solução é um conjunto de soluções que podem ser alcançadas fazendo pequenas alterações na solução atual, geralmente alterando um único elemento na solução. O problema com a busca local é que ela pode ficar presa em uma solução ótima local que pode ser muito mais pobre, em termos de adequação, do que a ótima global. A busca local com várias partidas trata desse problema executando o procedimento várias vezes com diferentes soluções aleatórias e retornando a melhor solução encontrada durante a execução de qualquer uma das buscas. No entanto, reiniciar a busca a partir de uma solução não relacionada ao ótimo local encontrado é uma estratégia que descarta as informações potencialmente relevantes representadas na solução ótima local. O algoritmo ILS melhora a busca local multi-start aplicando um operador de perturbação sobre o ótimo local e reconsiderando-o como uma solução inicial para a próxima iteração [40]

A busca por meio do algoritmo ILS é governada pelos passos de busca local, perturbação e critério de parada [67]. O algoritmo ILS possui quatro componentes principais [38]: o processo de geração da solução inicial, o método de busca local, o método de perturbação e o critério de aceitação de uma nova solução. O agrupamento é

realizado aplicando o algoritmo ILS a partir de informações sobre as dependências entre classes obtidas pela leitura de arquivos no formato *ODEM*. O algoritmo lê as dependências, busca um agrupamento ótimo das classes em pacotes e gera um arquivo texto com o formato RSF¹ para posterior cálculo do MoJoFM.

3.2.2 K-Means

K-means é um dos algoritmos de agrupamento mais conhecidos e utilizados na literatura. Ele envolve a atribuição de dados a grupos em um esforço para minimizar a variação dentro de cada grupo [70]. É um algoritmo iterativo que divide o conjunto de dados não-rotulados em k grupos diferentes de forma que cada dado pertença a apenas um grupo. O algoritmo funciona em cinco etapas: i) definir o número k de grupos nos quais os dados serão agrupados; ii) inicializar os centróides dos grupos; iii) atribuir dados ao grupo mais próximo; iv) recalcular os centróides dos grupos com base nos dados atribuídos a eles; e v) repetir os dois últimos passos até chegar a centróides ideais e as atribuições de dados para grupos que não mudem mais [107].

Muitos trabalhos foram propostos para tratar agrupamentos de elementos textuais utilizando o algoritmo *K-means*. Para superar a deficiência do algoritmo com dados de alta dimensão e esparsos, Wu et al. [124] propuseram um algoritmo *K-means* baseado em *Sim-Hash*. Após o pré-processamento do texto, o *Sim-Hash* foi usado para calcular os vetores de características extraídos e, em seguida, obter uma identidade digital de cada texto. *Sim-Hash* é um algoritmo utilizado para remover duplicatas em uma página Web que é amplamente utilizado devido à sua rápida velocidade de processamento [9]. A intenção é reduzir a dimensão e, como resultado, o vetor de características, que pode ser mapeado em um valor exclusivo (*fingerprint*). Este valor não apenas fornece a informação igual ao texto original, mas também fornece a informação do grau de diferença do que não é igual [9]. Os resultados experimentais mostraram que o algoritmo garantiu a qualidade do agrupamento, mas reduziu bastante a velocidade do *K-means* [124].

¹ Rigi Standard Format - <http://www.rigi.cs.uvic.ca/downloads/rigi/doc/node52.html>

Xiong et al [126] apresentaram um algoritmo de agrupamento de texto *K-means* melhorado, otimizando os centróides iniciais. O algoritmo primeiro calcula a densidade de cada dado e, em seguida, julga qual dado se caracteriza como um dado isolado. Após a remoção de todos os dados isolados, obtém-se um conjunto de dados com alta densidade. Em seguida, o algoritmo escolhe k dados de alta densidade como os centróides iniciais de cada grupo, de modo a maximizar a distância entre os dados selecionados. Resultados experimentais mostraram que o algoritmo *K-means* melhorado pode melhorar a estabilidade e a precisão do agrupamento de elementos de texto [126].

3.2.3 Spectral Clustering

O algoritmo *Spectral Clustering* é uma técnica de análise exploratória de dados, com raízes na teoria dos grafos [39]. Neste algoritmo, os dados que devem ser classificados em grupos são representados como nós e a similaridade entre eles é representada por uma aresta [46]. A ideia principal é agrupar todo o espectro de pontos de dados não organizados em vários grupos com base em sua singularidade. O algoritmo usa uma abordagem de conectividade em que nós (pontos de dados) que estejam conectados ou imediatamente próximos uns dos outros sejam identificados em um grafo. Estes nós são então mapeados para um espaço de baixa dimensão que pode ser facilmente segregado para formar os *clusters*. O agrupamento espectral usa informações dos autovalores (espectro) de matrizes especiais (ou seja, matriz de afinidade, matriz de grau e matriz laplaciana) derivadas do grafo ou do conjunto de dados [23].

O algoritmo pode ser implementado de forma eficiente por software de álgebra linear padrão e muitas vezes supera algoritmos tradicionais, como o *k-means* [7]. Uma implementação típica consiste em três etapas: (i) construir o grafo de similaridade entre os N dados; (ii) projetar os dados em um espaço dimensional reduzido, calculando os primeiros k autovetores da matriz Laplaciana para definir um vetor de características para cada dado; e (iii) agrupar os dados executando o algoritmo *k-means* nessas características para separar os dados em k classes [7, 46].

Gallardo García et al. [42] compararam a eficácia e a precisão de vários algoritmos de agrupamento em tarefas de agrupamento de texto. O pré-processamento do texto foi realizado com TF-IDF. As tarefas de agrupamento foram realizadas com três algoritmos diferentes: *Affinity Propagation*, *K-means* e *Spectral Clustering*. Enquanto o *Affinity Propagation* teve a melhor média de precisão, o *Spectral Clustering* teve as maiores médias *F-measure*, ficando o *K-means* como o pior dos três algoritmos. Os algoritmos foram testados com pequenos conjuntos de dados. Possivelmente, esta é a razão pela qual o *Spectral Clustering* obteve os melhores resultados, porque o algoritmo funciona melhor com poucos grupos, enquanto o *Affinity Propagation* é mais eficiente com muitos grupos e grandes conjuntos de dados [42].

Na classificação de texto, a dimensionalidade do vetor de características geralmente é alta. Várias abordagens foram propostas para reduzir a dimensionalidade do vetor de características durante a classificação automática de texto. Suganthi e Manimekalai [110] apresentaram um algoritmo de classificação baseado no *Spectral Clustering* que reduz a dimensionalidade de um vetor de características. O algoritmo foi aplicado a vários problemas de classificação de texto e os resultados mostraram que é mais eficaz e mais preciso do que o algoritmo *K-means* [110].

Janani e Vijayarani [52] propuseram uma nova combinação do algoritmo *Spectral Clustering* com o algoritmo *Particle Swarm Optimization* para melhorar o agrupamento de grandes volumes de documentos de texto. O algoritmo proposto foi comparado com o *K-means* esférico, o método *Expectation Maximization* e o algoritmo *Particle Swarm Optimization* padrão. Os resultados mostraram que a aplicação do algoritmo proposto resultou em melhor precisão de agrupamento do que as outras técnicas [52].

3.2.4 BIRCH Clustering

BIRCH é uma abreviação para “*Balanced Iterative Reducing and Clustering using Hierarchies*”. O algoritmo envolve a construção de uma estrutura de árvore da qual os centróides do grupo são extraídos. O *BIRCH* agrupa dados multidimensionais de forma incremental e dinâmica para tentar produzir um agrupamento de boa qualidade

dentro dos recursos de memória disponíveis e restrições de tempo [129]. O *BIRCH* é frequentemente usado para complementar outros algoritmos de agrupamento, criando um resumo do conjunto de dados que o outro algoritmo de agrupamento processa. No entanto, o *BIRCH* tem uma grande desvantagem: o algoritmo só pode processar atributos cujos valores possam ser representados em um espaço euclidiano, ou seja, nenhum atributo categórico pode estar presente [18].

Karpov e Goroslavskiy [57] apresentaram uma técnica de agrupamento baseada no algoritmo *BIRCH* e em métodos de *Latent Semantic Analysis* para agrupamento de grandes conjuntos de dados de alta dimensão. O trabalho apresentou um modelo de documento e uma ferramenta de agrupamento para processamento de textos nos idiomas russo e inglês a partir de uma versão modificada do *BIRCH*. Os resultados foram comparados com métodos de agrupamento tradicionais, como *k-means* e *Expectation–Maximization*. Na comparação com estes métodos, o algoritmo *k-means* mostrou resultados de baixa qualidade. Além disso, o tempo de processamento exigido pela abordagem proposta é menor em mais de duas ordens de grandeza com a mesma precisão se comparado com o *k-means* [57].

Basavaraju e Prabhakar [75] apresentaram um algoritmo de agrupamento eficiente que incorpora as características do algoritmo *K-means* e do algoritmo *BIRCH* para detecção de *spam*. As distâncias do vizinho mais próximo e as distâncias dos vizinhos *K-próximos* podem servir como base para a classificação dos dados de teste com base no aprendizado supervisionado. Diferentes medidas de avaliação foram usadas para analisar a eficácia do algoritmo de agrupamento desenvolvido em comparação com vários classificadores. Como resultado, os autores observaram que o algoritmo de agrupamento *K-means* funciona bem para conjuntos de dados menores (tamanho de 120). A combinação do *BIRCH* com *K-NNC* (*K-nearest neighbour classifier*) foi a que apresentou os melhores resultados, pois funciona melhor com grandes conjuntos de dados (tamanho de 400). No agrupamento *BIRCH*, as decisões de agrupamento são tomadas sem varrer todos os dados, utilizando apenas informações locais. O *BIRCH*

demonstrou ser um algoritmo de agrupamento melhor, requerendo uma única varredura de todo o conjunto de dados [75].

3.2.5 Agrupamento Hierárquico

Agrupamento hierárquico refere-se a uma classe de algoritmos de aprendizado não-supervisionado que buscam construir uma hierarquia a partir de um conjunto de dados enquanto distribuem estes dados em grupos. Esses algoritmos possuem duas vantagens sobre algoritmos de agrupamento não-hierárquicos: não exigem que o número de grupos seja especificado a priori e geram uma hierarquia com todas as amostras no conjunto de dados como resultado do processo de agrupamento [24].

Os dados a serem agrupados são tipicamente representados como vetores numéricos, de modo que seja fácil calcular a distância entre dois dados quaisquer. No entanto, há diversas formas de calcular a distância entre dois dados:

- **Distância Euclidiana** - É uma fórmula que mede a distância de dois pontos x_1 e x_2 em um espaço n -dimensional. É a distância em linha reta entre dois pontos em um plano [?].

$$d_{Euclidean}(x_1, x_2) = \sqrt{\sum_{i=1}^n (x_1^i - x_2^i)^2}$$

- **Distância Manhattan** - A distância de Manhattan também é conhecida como distância do quarteirão da cidade (*city block distance*). Ela é obtida ao percorrer o caminho entre dois pontos (x_1 e x_2) usando apenas retas ortogonais. O nome vem da ilha de Manhattan, onde as ruas são organizadas em uma grade retangular [?]. A distância é calculada como a soma do módulo das diferenças entre as dimensões.

$$d_{Manhattan}(x_1, x_2) = \sum_{i=1}^n |x_1^i - x_2^i|$$

- **Distância Cosseno** - Em tarefas de mineração de texto, a distância de cosseno, é calculada como o cosseno do ângulo θ entre dois vetores x_1 e x_2 [?]. Ela é calculada pela fórmula abaixo, onde $x_1 \cdot x_2$ é o produto escalar dos vetores x_1 e x_2 e $\|x_i\|$ é o comprimento do vetor x_i .

$$d_{\cosine}(x_1, x_2) = \cos(\theta) = \frac{x_1 \cdot x_2}{\|x_1\| \|x_2\|} = \frac{\sum_{i=1}^n x_1^i \cdot x_2^i}{\sqrt{\sum_{i=1}^n (x_1^i)^2} \cdot \sqrt{\sum_{i=1}^n (x_2^i)^2}}$$

O índice de Jaccard é usada para medir a similaridade e a diversidade de conjuntos de amostras quando as variáveis são nominais. Como temos dados com distribuição contínua, não utilizaremos esta estatística.

O critério de ligação determina qual distância entre conjuntos de dados deve ser utilizada na formação dos grupos. O algoritmo mesclará os pares de grupos que minimizam este critério. Os critérios de ligação mais utilizados na literatura são:

- **Single-linkage** - Usa o mínimo das distâncias entre todos os pares de dados dos dois grupos;
- **Complete-linkage** - Usa a distância máxima entre todos os pares de dados dos dois grupos;
- **Average-linkage** - Usa a média das distâncias entre todos os pares de dados dos dois grupos;
- **Ward** - Minimiza a variação dos grupos que estão sendo formados. É calculado como o aumento na soma dos quadrados do erro (*error sum of squares* - ESS) após a fusão de dois grupos [?]. O ESS é a soma dos quadrados dos desvios do valor médio ou do vetor médio (centróide), dado por:

$$ESS(x) = \sum_{i=1}^n \left| x_i - \frac{1}{n} \sum_{j=1}^n x_j \right|^2$$

A distância é descrita pela seguinte expressão:

$$D(x, y) = ESS(xy) - [ESS(x) + ESS(y)]$$

Nos algoritmos de agrupamento hierárquico, a hierarquia de grupos é desenvolvida na forma de uma árvore, conhecida como *dendrograma* [53] que mostra as relações taxonômicas dos grupos produzidos pelo algoritmo [113]. O agrupamento hierárquico tem duas variantes: *aglomerativo* (de baixo-para-cima, *bottom-up*), na qual o algoritmo começa com cada dado representado em um grupo individual e mescla estes grupos até que reste apenas um; e *divisivo* (de cima-para-baixo, *top-down*), que de forma progressiva divide um grupo que no início continha todos os dados de interesse [53].

O algoritmo *Hierarchical Agglomerative Clustering* (HAC) utiliza um parâmetro que indica o número de grupos do agrupamento desejado e uma estratégia de corte *ad-hoc*. O HAC constrói o dendrograma e, em seguida, determina onde cortá-lo para obter o número de grupos indicado por este parâmetro. O algoritmo de agrupamento em si não depende de um parâmetro que informe o número de grupos desejados. Entretanto, para formar um agrupamento compatível com o agrupamento proposto pelos desenvolvedores, o número de grupos é utilizado para encontrar pontos de corte no dendrograma.

A Figura 3.1 mostra um dendrograma criado pelo HAC para o software *joda-money*, mostrando um corte para obter dois grupos. O ponto de corte foi baseado na distribuição DEV, ou seja, foram criados tantos grupos no agrupamento hierárquico quanto existem na distribuição de classes e pacotes proposta pelos desenvolvedores. Assim, para o *joda-money* teremos dois pacotes: um com 28 classes e outro com uma única classe.

3.3 Seleção dos Sistemas de Software

Os sistemas de software analisados neste estudo foram selecionados por possuírem licenças abertas (*open-source*), código-fonte disponível em repositórios acessíveis e uma quantidade variada de classes, envolvendo desde sistemas pequenos (com poucas dezenas de classes) até sistemas com milhares de classes.

Todos os sistemas selecionados foram desenvolvidos na linguagem de programação Java. Esta linguagem foi escolhida devido à disponibilidade de ferramentas para a coleta dos dados necessários para a modularização de software e porque todos

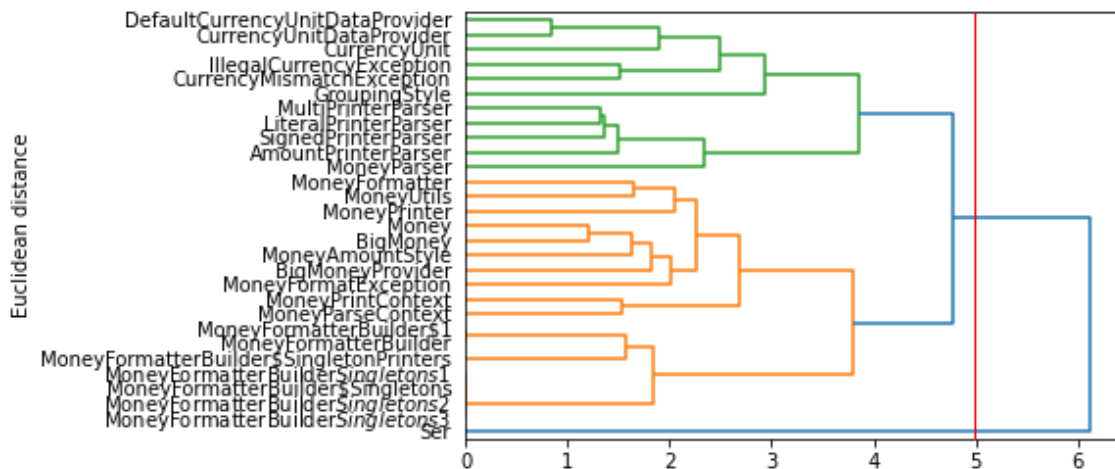


Figura 3.1: Dendrograma do agrupamento hierárquico para o software *joda-money*, mostrando um corte em dois grupos.

os trabalhos relacionados no capítulo anterior utilizaram sistemas desenvolvidos com esta linguagem de programação. A Tabela 3.1 mostra a lista de sistemas de software selecionados para o estudo.

Tabela 3.1: Sistemas de software selecionados para o estudo. A coluna “Pacotes” apresenta o número de pacotes da implementação original dos softwares e a coluna “Classes” mostra o número de classes nesta implementação.

Sistema de software	Versão	Classes	Pacotes
Apache Ant	1.10.9	1.161	62
Apache Jena Core	4.0.0	1.160	51
Apache Lucene	8.8.2	2.079	30
Joda Beans	2.0	184	17
Joda Convert	2.1.1	120	2
Joda Money	1.0	29	2
Joda Time	2.10	247	7
Jung Algorithms	2.0	139	16
Stanford Corenlp	4.2.2	3.304	106

Após a seleção dos softwares, usamos a ferramenta *pf-cda*² para extração das definições das classes e suas dependências a partir dos arquivos compilados dos projetos (arquivos *JAR*). Estas definições são armazenadas em arquivos no formato *ODEM*³. Os dados para os experimentos que utilizam incorporação de palavras foram processados usando ferramentas de NLP, como *NLTK*⁴, para processamento de *stop-words*; *Scikit-learn*⁵, para algoritmos de agrupamento como *K-Means*, *BIRCH*, *Spectral Clustering* e *Agglomerative Clustering*; *Gensim*⁶, para processamento dos vetores pré-treinados e algoritmos de geração de modelos de incorporação, como *Word2Vec*, *FastText* e *Doc2Vec*; e *Scipy*⁷, para cálculo de métricas e matrizes de distância.

3.4 Seleção de Modelos de Incorporação de Palavras Pré-treinados

Modelos pré-treinados de incorporação de palavras (*pre-trained Word Embeddings*) são aplicações de aprendizado por transferência treinadas em uma tarefa e utilizadas para resolver outra tarefa semelhante. Aprendizado por transferência (*transfer learning*) é um método de aprendizado de máquina que reutiliza um modelo pré-treinado como ponto de partida para a criação de um modelo para uma nova tarefa. Esses modelos de incorporação são treinados em grandes conjuntos de dados, são armazenados e ficam disponíveis para que possam ser utilizados por desenvolvedores para resolver outras tarefas. Os modelos de incorporação de palavras utilizados neste trabalho são *Word2Vec*, desenvolvido pela *Google*, *GloVe*, desenvolvido pela Universidade de Stanford, e *FastText* desenvolvido pelo *Facebook*.

² Programmer's Friend Class Dependency Analyzer 2.6.0, <http://www.dependency-analyzer.org/>

³ Object Dependency Exploration Model (ODEM), <http://www.programmers-friend.org/ODEM/apidoc/index.html>

⁴ Natural Language Toolkit (NLTK) - <https://www.nltk.org/>

⁵ Scikit-learn, <https://scikit-learn.org/stable/>

⁶ Gensim, <https://radimrehurek.com/gensim/>

⁷ Fundamental algorithms for scientific computing in Python, <https://scipy.org/>

O modelo *Word2Vec* foi treinado no conjunto de dados do *Google News* ⁸ (cerca de 100 bilhões de palavras) contendo 3 milhões de palavras e ajustado para vetores de 300 dimensões.

O modelo *Glove* (*glove.42B.300d.zip*)⁹ foi treinado a partir de palavras da *Wikipedia* (2014) e da *Gigaword 5*¹⁰ com 400k palavras do vocabulário. O modelo *glove.840B.300d.zip*, treinado na *Common Crawl* ¹¹ possui cerca de 2.2M termos em seu vocabulário e vetores de 300 dimensões.

O modelo *FastText Wiki-en* (*Wiki word vectors*)¹² foi treinado a partir da *Common Crawl* e *Wikipedia*. Nesta pesquisa, foram utilizadas as versões disponíveis com 300 dimensões e o idioma inglês. O modelo *Wiki-news* (*English word vectors*) representa 1 milhão de vetores de palavras treinadas da *Wikipedia* 2017, *corpus* da *webbase UMBC*¹³ e conjunto de dados de notícias *Statistical and Neural Machine Translation*¹⁴.

O modelo *SO_word2vec*¹⁵ se baseia no *Word2Vec* e foi treinado com base em um *corpus* de 1.787.145 palavras-chave construído a partir de conteúdo proveniente do *Stack Overflow*, uma plataforma pública de perguntas e respostas sobre desenvolvimento de software. Este modelo disponibiliza apenas uma versão com 200 dimensões.

3.5 Experimentos baseados nos Nomes de Classes

Nesta seção investigamos o uso de informações léxicas provenientes dos nomes das classes e dos pacotes onde elas estão localizadas para agrupar artefatos de software relacionados. O agrupamento é realizado pelo algoritmo HAC, utilizando a distância euclidiana e o critério de ligação *Ward*. Mais adiante, outros algoritmos, medidas de distância e critérios de ligação serão comparados com estes resultados.

⁸ Word2Vec pre-trained word/phrase vectors: <https://code.google.com/archive/p/word2vec>

⁹ GloVe: Global Vectors for Word Representation, <https://nlp.stanford.edu/projects/glove/>

¹⁰ English Gigaword Fifth Edition (<https://catalog.ldc.upenn.edu/LDC2011T07>)

¹¹ Common Crawl - <https://commoncrawl.org/>

¹² Wiki word vectors - <https://fasttext.cc/docs/en/pretrained-vectors.html>

¹³ UMBC WebBase corpus of 3B English words, <https://ebiquity.umbc.edu/blogger/2013/05/01/umbc-webbase-corpus-of-3b-english-words/>

¹⁴ Statistical Machine Translation, <http://statmt.org/>

¹⁵ https://github.com/vefstathiou/SO_word2vec

3.5.1 Experimentos com Variações de Modalidade

A utilização de ferramentas de NLP para agrupar artefatos de software requer a definição de sentenças ou palavras a partir das informações disponíveis nas classes componentes dos sistemas. Neste primeiro grupo de estudos, apenas os componentes dos nomes das classes serão utilizados na coleta de palavras e sentenças. Para tanto, definimos as seguintes modalidades:

1. A primeira modalidade, chamada “*single*”, utiliza apenas o nome da classe como um termo léxico (Figura 3.2). No caso de classes internas (*inner classes*), nas quais o nome da classe contida é separado do nome da classe que a contém por um cifrão (\$), considera-se apenas o nome depois do cifrão (Figura 3.3);
2. A segunda modalidade, chamada “*complete*”, utiliza o nome dos pacotes e o nome da classe, excluindo apenas o nome do domínio (Figura 3.2). No caso das classes internas, o cifrão é substituído por um ponto (Figura 3.3);

Um nome de domínio é uma *string* que identifica um campo de autonomia administrativa, autoridade ou controle¹⁶. Os nomes de domínio são representados em uma estrutura hierárquica, onde um domínio de nível superior (*top-level domain* - TLD) representa a primeira parada após a zona raiz. Em termos mais simples, um TLD é tudo o que segue o ponto final de um nome de domínio. Por exemplo, o nome de domínio “whois.icann.org” tem três níveis de hierarquia: “org” é o nível superior, “icann” é o segundo nível e “whois” é o terceiro nível.

Para nomear componentes na linguagem Java, segundo as convenções oficiais, o prefixo de um nome de pacote é sempre escrito em letras minúsculas e deve ser um dos nomes TLD, como por exemplo, com, edu, gov, mil, net, org¹⁷. Os componentes subsequentes do nome do pacote variam de acordo com as próprias convenções internas de nomenclatura da organização. Essas convenções podem

¹⁶ About Domain Names - <https://www.icann.org/resources/pages/about-domain-names-2018-08-30-en>

¹⁷ Naming Conventions, <https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html>

especificar que determinados componentes de nome de diretório sejam nomes de divisão, departamento, projeto, máquina ou login. As empresas usam o nome de domínio da Internet invertido para iniciar seus nomes de pacote. Por exemplo, tipicamente se utiliza “com.example.mypackage” para um nome de pacote chamado “mypackage” criado por um desenvolvedor em “example.com”¹⁸.

3. A última modalidade, chamada “full”, considera o nome completo da classe, incluindo os pacotes e o nome do domínio. Como no caso anterior, o cifrão usado nas classes internas é substituído por um ponto (Figura 3.3).

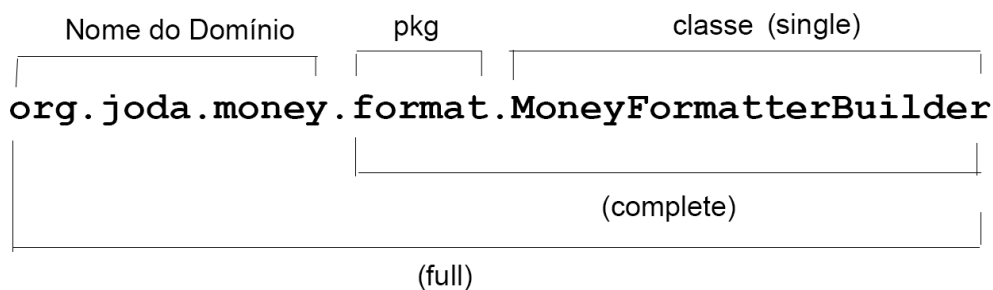


Figura 3.2: Identificação dos termos léxicos para classes convencionais nas modalidades “single”, “complete” e “full”.

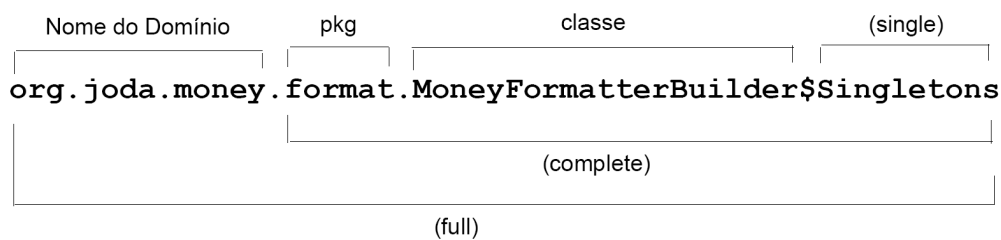


Figura 3.3: Identificação dos termos léxicos para classes internas nas modalidades “single”, “complete” e “full”.

Depois de identificados os termos léxicos que serão utilizados para representar a classe no agrupamento, o próximo passo consiste em dividir estes termos em palavras,

¹⁸ Naming a Package, <https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html>

identificando e separando os pontos e assumindo que os nomes de classes e pacote utilizam o padrão de nomenclatura *Camel Case*¹⁹, que é recomendado pelo padrão de codificação da linguagem de programação Java. Por exemplo, o conjunto de palavras obtido para a classe *ModifiedSelector\$ComparatorName* é *[modified,selector\$,comparator,name]*. No caso de múltiplos caracteres, como “XMLOutputter” e “IOException”, é possível a divisão resultando em “XML” / “Outputter” e “IO” / “Exception”. Porém, este procedimento falha em algumas construções, como “DEFAULT_OUTPUTTER” e “BUILD_SYSCLASSPATH”. Por fim, os sinais de pontuação são removidos. No exemplo acima, obtemos o conjunto final de palavras *[modified,selector,comparator,name]*.

Para comparar os agrupamentos gerados pelo algoritmo HAC com os agrupamentos DEV e ILS, nomeamos os experimentos como a modalidade (“s” para “single”, “c” para “complete” e “f” para “full”) seguida por um traço e pelo algoritmo que determina o número de grupos que devem ser gerados pelo HAC (DEV ou ILS). Nas comparações com o agrupamento DEV, configuramos o HAC para gerar um número de grupos igual ao número de grupos do agrupamento DEV. De forma similar, nas comparações com o agrupamento ILS, configuramos o HAC para gerar um número de grupos igual ao número de grupos do agrupamento ILS orientado pela métrica MQ para o software sob análise. Por exemplo, temos:

- Os experimentos s-DEV, c-DEV e f-DEV mostram os resultados do agrupamento HAC com base no número de grupos da distribuição DEV;
- Os experimentos s-ILS, c-ILS e f-ILS mostram os resultados do agrupamento HAC com base no número de grupos da distribuição ILS.

Para estes experimentos, utilizamos inicialmente o agrupamento aglomerativo com a métrica euclidiana e o critério de ligação *Ward*. O objetivo é estudar estas modalidades no agrupamento HAC para observar se as diferenças entre os resultados das diferentes modalidades são relevantes com duas configurações de número de grupos.

¹⁹ Camel Case - <https://techterms.com/definition/camelcase>

Para uma primeira avaliação, selecionamos os nove sistemas de software apresentados na Seção 3.3 e o modelo de incorporação de palavras *Google News*.

Para calcular o vetor que representa cada classe a partir dos vetores de palavras do modelo de incorporação, comparamos as estratégias de média, máximo, mínimo, mediana e desvio padrão. Na estratégia de média, calculamos a média de cada posição do vetor que representa as palavras envolvidas e usamos o vetor resultante como representação da classe no agrupamento. As demais estratégias foram implementadas de forma similar, cada qual utilizando um método diferente para compor o vetor resultante a partir dos vetores com as incorporações das palavras envolvidas. A estratégia de média apresentou melhores resultados nos estudos e será reportada deste ponto em diante.

Para representar o valor do MoJoFM, comparamos as estratégias de média e mediana. Ambas as medidas apresentaram valores muito próximos, com desvio padrão abaixo de 0,01. Portanto, somente a média será utilizada deste ponto em diante.

A Tabela 3.2 mostra o resultado do MoJoFM para as três modalidades baseadas no número de grupos do agrupamento DEV. O agrupamento c-DEV apresenta maior valor de MoJoFM que o s-DEV em oito das nove instâncias, perdendo na instância *joda-convert*. Por outro lado, o agrupamento c-DEV apresenta maior valor de MoJoFM que o f-DEV em apenas duas das nove instâncias, empatando em outras duas instâncias. Por fim, o agrupamento f-DEV vence o s-DEV em oito das nove instâncias. Concluimos que a distribuição f-DEV apresentou resultados com maiores valores de MoJoFM, indicando que esta distribuição está mais próxima da distribuição de classes em pacotes proposta pelos desenvolvedores.

Com relação aos agrupamentos baseados no número de grupos encontrados pelo algoritmo ILS (Tabela 3.3), observamos que o s-ILS apresenta os piores resultados, perdendo para o c-ILS em sete das nove instâncias e empatando em uma instância. O agrupamento c-ILS, por sua vez, vence o f-ILS em seis das nove instâncias e empata em duas instâncias. Sendo assim, percebemos que o agrupamento c-ILS apresenta os melhores resultados com o número de grupos do algoritmo ILS. Por outro lado, os

Tabela 3.2: Valores MoJoFM para o agrupamento HAC aplicado às séries s-DEV, c-DEV e f-DEV, utilizando o modelo de incorporação de palavras *Google News*.

Software	s-DEV	c-DEV	f-DEV
apache-ant	36,70	70,71	73,51
apache-jena	31,41	69,57	70,43
apache-lucene	44,01	86,79	87,01
joda-beans	48,98	75,51	75,51
joda-convert	75,86	75,86	86,21
joda-money	57,14	57,14	100,00
joda-time	56,60	71,07	85,53
jung-algorithms	39,60	68,32	72,28
stanford-corenlp	34,76	66,67	67,79
Média	47,23	71,29	79,81

Tabela 3.3: Valores MoJoFM para o agrupamento HAC aplicado às séries s-ILS, c-ILS e f-ILS, utilizando o modelo de incorporação de palavras *Google News*.

Software	s-ILS	c-ILS	f-ILS	ILS	$\frac{ ILS - c_{ILS} }{c_{ILS}}$
apache-ant	18,02	23,57	22,29	22,24	5,64%
apache-jena	12,34	15,93	16,27	15,75	1,13%
apache-lucene	8,12	8,66	8,33	8,06	6,93%
joda-beans	21,70	25,47	24,53	25,00	1,85%
joda-convert	3,45	3,45	3,45	3,85	11,59%
joda-money	15,00	10,53	10,53	9,26	12,06%
joda-time	11,95	14,38	11,80	10,69	25,66%
jung-algorithms	20,59	24,75	24,51	23,00	7,07%
stanford-corenlp	10,75	13,66	13,15	15,34	12,30%
Média	13,55	15,60	14,98	14,80	9,04%

resultados de todos os agrupamentos baseados no ILS são muito inferiores aos resultados dos agrupamentos com o número de grupos da distribuição DEV.

A Tabela 3.3 apresenta ainda os valores de MoJoFM calculados para as distribuições resultantes da aplicação do algoritmo ILS orientado pela métrica MQ sobre as instâncias selecionadas, ou seja, considerando as dependências entre as classes e suas posições nos pacotes. O valor do MoJoFM apresentado na coluna ILS é o melhor valor de 30 execuções independentes do algoritmo ILS. As trinta execuções foram realizadas para atenuar os efeitos positivos ou negativos do componente aleatório do algoritmo ILS. Elas são desnecessárias em algoritmos determinísticos, como o agrupamento hierárquico. Podemos perceber que os valores resultantes do algoritmo ILS estão próximos dos valores apresentados na coluna c-ILS. A diferença média de 9,04% é pequena, ainda que o pior caso apresente uma grande diferença (25,66%). Nesta instância específica, o agrupamento f-ILS obteve um valor de MoJoFM próximo à distribuição do ILS (11,80).

3.5.2 Experimentos com Variações de Modelos de Incorporação

Para verificar se diferentes modelos de incorporação de palavras podem melhorar os resultados do agrupamento f-DEV, repetimos os experimentos com este agrupamento utilizando os modelos listados na Seção 3.4. A Tabela 3.4 mostra os valores de MoJoFM para os sistemas de software e os modelos de incorporação.

Nem todas as palavras contidas nos nomes das classes dos sistemas de software estudados estão representadas nos modelos de incorporação. As palavras que não foram encontradas em um modelo são ignoradas e isto pode afetar o cálculo do MoJoFM para um agrupamento. Para avaliar a frequência de palavras ignoradas, calculamos a razão entre o número de palavras ignoradas em cada modelo e o número total de palavras em cada software. A Tabela 3.5 mostra o total de palavras processadas e o número de palavras ignoradas para os sistemas de software estudados usando os diferentes modelos de incorporação de palavras. A média da razão entre o número total de palavras e o número de palavras ignoradas também está apresentada no rodapé da tabela.

Tabela 3.4: Cálculo do MoJoFM para o agrupamento f-DEV utilizando os modelos de incorporação de palavras *Word2Vec Google News (GNews)*, *Glove.6B (Glove)*, *FastText-wiki-news (Wiki-News)*, *FastText-wiki-en (Wiki-en)* e *SO_Word2Vec (SO)*.

Software	GNews	Glove	Wiki-News	Wiki-en	SO	Média	SD
apache-ant	59,48	40,15	57,82	59,95	54,77	55,39	8,56
apache-jena	57,49	46,55	61,92	67,30	55,83	58,79	7,70
apache-lucene	49,46	33,67	53,93	54,87	61,32	51,72	11,92
joda-beans	61,51	46,12	59,92	57,54	56,33	57,26	6,42
joda-convert	34,52	53,34	41,67	38,10	90,00	49,53	18,44
joda-money	71,43	84,00	97,14	100,00	90,00	86,51	11,95
joda-time	82,78	65,37	77,81	78,15	51,84	72,09	10,92
jung-algorithms	67,02	54,19	66,67	78,72	53,24	64,90	9,52
stanford-corenlp	46,15	35,76	49,06	53,20	49,19	47,53	7,25
Média	58,87	51,02	62,88	65,31	62,50	60,41	10,30

O modelo *Google News* mostrou a maior porcentagem de palavras ignoradas, talvez por capturar termos que não são palavras em nenhum idioma, mas marcações de texto, símbolos ou concatenações de palavras. Esta não é uma exclusividade deste modelo de incorporação. A Tabela 3.6 mostra palavras encontradas nos modelos de incorporação *Google News* e *Wiki-en* como exemplos de termos que não são efetivamente palavras. As linhas “DATE”, “Date” e “date” mostram uma lista de palavras similares obtidas ao calcular a similaridade de cosseno entre o vetor que representa a palavra no cabeçalho e os vetores para cada palavra no modelo²⁰. Podemos observar que nem todas as palavras similares do ponto de vista do *corpus* utilizado no modelo são de fato próximas às variações do termo “date”.

Cabe ressaltar que o modelo SO, construído a partir de informações associadas à Engenharia de Software, apresentou a melhor taxa no contexto geral. Entretanto, este é um modelo experimental, com tamanho inferior aos outros modelos.

Observando as Tabelas 3.4 e 3.5, concluímos que o modelo de incorporação de palavras que apresentou a melhor eficácia neste experimento foi o *Wiki-en*. Portanto, este

²⁰ Gensim.Keyedvectors.most_similar https://tedboy.github.io/nlps/generated/generated/gensim.models.Word2Vec.most_similar.html

Tabela 3.5: Proporção de palavras ignoradas (Ign.) para os sistemas de software, calculada para os diferentes modelos de incorporação de palavras (*GNews*, *Glove*, *SO*, *Wiki News* e *Wiki-en*).

Software	<i>Gnews</i>		<i>Glove</i>		<i>SO</i>		<i>Wiki-News</i>		<i>Wiki-en</i>	
	Total	Ign.	Total	Ign.	Total	Ign.	Total	Ign.	Total	Ign.
apache-ant	501	85	500	86	569	17	539	47	548	38
apache-jena	426	50	430	46	463	13	448	28	455	21
apache-lucene	449	32	456	25	475	6	464	17	471	10
joda-beans	80	5	78	7	85	0	83	2	84	1
joda-convert	30	2	32	0	32	0	32	0	32	0
joda-money	27	1	28	0	28	0	28	0	28	0
joda-time	93	6	98	1	99	0	97	2	98	1
jung-algorithms	118	18	129	7	135	1	127	9	133	3
stanford-corenlp	741	126	779	88	842	25	802	65	822	45
Média	10,10%		6,70%		1,20%		4,30%		2,60%	

modelo será utilizado nos próximos experimentos. Por outro lado, a boa eficácia com o pequeno percentual de palavras ignoradas no modelo experimental *SO* em comparação com o percentual de palavras ignoradas nos outros modelos indica que faz sentido analisar a construção de um modelo de incorporação voltado para a distribuição de classes em pacotes.

Com base nas Tabelas 3.2 a 3.4 podemos concluir que a modalidade f-DEV gera distribuições de classes em pacotes mais próximas da distribuição proposta pelos desenvolvedores dentre as modalidades analisadas. No entanto, esta conclusão traz poucas surpresas: a modalidade f-DEV utiliza termos vindos dos nomes dos pacotes e a tendência é que classes que estejam no mesmo pacote na distribuição proposta pelos desenvolvedores participem do mesmo grupo proposto pelo algoritmo de agrupamento por compartilharem os termos originários dos nomes dos pacotes.

Os resultados positivos da modalidade f-DEV indicam que o cálculo de distância entre termos vindos de modelos de incorporação de palavras e a consolidação pela média destes termos para fins de representação das classes podem ser utilizados para orientar a remodularização de software. No entanto, as modalidades que utilizam nomes de pacotes não são uma base de comparação justa com outras modalidades de identificação dos termos utilizados para representar as classes para fins de remodularização, uma vez

Tabela 3.6: Exemplos de palavras encontradas nos modelos de incorporação de palavras *Google News* e *Wiki-en* e palavras similares para variações do termo “date”.

Modelos	<i>Google News</i>	<i>Wiki-en</i>
‘DATE’	‘EVENT’, ‘EVENT_NATIONAL’, ‘EVENT_THE_CANADIAN’	‘DATES’, ‘DATE-’, ‘EXPIRY’
‘Date’	‘Penn_Nearly_Ditches’, ‘_____Date’, ‘Type_Bits_KeyID’	‘2Date’, ‘4Date’, ‘Date-’
‘date’	‘dates’, ‘Ligand_judgment’, ‘WCDMA_3G_Nokia’	‘dates’, ‘date.’, ‘date.So’

que os nomes dos pacotes estão relacionados aos nomes dos diretórios de um programa escrito em Java e, conseqüentemente, ao agrupamento proposto pelos desenvolvedores.

A modalidade c-DEV sofre do mesmo viés apontado em f-DEV: ainda que desconsidere o nome do domínio, os nomes dos pacotes dentro do domínio da aplicação estão associados aos diretórios em que as classes foram distribuídas pelos desenvolvedores. Podemos considerar que o MoJoFM resultante de f-DEV pode funcionar como um limite superior do que pode ser atingido com termos vindos de outras regiões do código-fonte, uma vez que f-DEV utiliza apenas o nome da classe para diferenciar a representação de duas classes que estão no mesmo pacote.

Assim, os próximos estudos vão analisar se informações léxicas provenientes de diferentes regiões do código-fonte das classes podem aprimorar os resultados obtidos com a modalidade s-DEV. Embora esta modalidade tenha apresentado os piores resultados entre as modalidades baseadas no número de grupos da distribuição DEV, ela não utiliza os nomes dos pacotes na representação das classes e, conseqüentemente, não induz a distribuição proposta pelos desenvolvedores.

3.6 Experimentos baseados em outras Informações da Classe

Depois de analisar as possibilidades de remodularização de software com base nos nomes das classes, passamos a analisar a remodularização com base no conteúdo de outras regiões dos arquivos de código-fonte. Para realizar estas análises, desenvolvemos um *parser* com base em expressões regulares para capturar as classes, implementações de interfaces ou herança, atributos, métodos, comentários e importações de classes apresentados em um arquivo de código-fonte Java. Conforme os experimentos discutidos na seção anterior, os agrupamentos desta seção foram realizados usando o algoritmo HAC, a distância euclidiana, o critério de ligação *Ward* e o modelo de incorporação de palavras *Wiki-en*.

3.6.1 Experimentos com Palavras de Outras Áreas do Código

Definimos diferentes modalidades com base nas informações supracitadas e denominadas M1, M2, M3 e M4. M1 considerou os nomes das classes (como o *s-DEV*), as implementações de interfaces e as heranças. M2 considerou os nomes das classes, as implementações de interfaces, as heranças e os atributos. M3 considerou os elementos de M2 e os nomes dos métodos. Por fim, M4 considerou os elementos de M3, as diretivas de importação (*import*) e os comentários.

No processamento de linguagem natural de textos comuns, palavras irrelevantes (*stop words*) ocorrem com frequência e não acrescentam significado ao texto. Para verificar se as *stop words* têm alguma influência sobre os resultados da remodularização de software, realizamos os experimentos seguintes incluindo e removendo *stop words*. A lista de *stop words* utilizada nos experimentos é uma compilação de palavras comuns em inglês²¹ e as palavras reservadas da linguagem de programação Java.

A Tabela 3.7 mostra o percentual de palavras ignoradas em relação ao número total de palavras nos experimentos com as modalidades M1 a M4 ao remover as *stop words* e sem remover as *stop words* no processo de decomposição dos nomes encontrados no código-fonte em palavras. O experimento M1 mostra um percentual de

²¹ NLTK's list of english stop words, <https://gist.github.com/sebleier/554280>

palavras ignoradas menor que os demais em remover ou considerar as *stop words*. Mais palavras são ignoradas nestas modalidades devido à inclusão de tipos de palavras que não são tipicamente usados nos nomes das classes (por exemplo, artigos definidos, artigos indefinidos, pronomes, entre outros) ou composição de palavras não desmembradas pelo analisador sintático (constantes, identificadores em *Camel Case* com várias letras maiúsculas, etc.). **

A Figura 3.4 mostra exemplos de palavras malformadas em trechos do código-fonte, como *constantpool* (linha 1), *msvsscheckin* (linha 2), *modifiedselector* (linha 3), *mmlbtype* (linha 4), *retryable* (linha 5) e *componentdef* (linha 6). Em geral, a modalidade M1 apresenta um percentual menor de palavras ignoradas do que as demais modalidades. Notamos também que o percentual não varia de forma expressiva quando as *stop words* são removidas antes da busca no modelo de incorporação de palavras.

```
(1) import org.apache.tools.ant.taskdefs.optional.depend.constantpool.ClassCPInfo;  
(2) public class MSVSSCHECKIN extends MSVSS  
(3) import org.apache.tools.ant.types.selectors.modifiedselector.ModifiedSelector;  
(4) public static final String COMMAND_MKLBTYP = "mklbtype";  
(5) public interface Retryable  
(6) public class Componentdef extends Definer
```

Figura 3.4: Exemplos de palavras malformadas gerando palavras ignoradas no processo.

A Tabela 3.8 mostra os valores de MoJoFM obtidos para as modalidades M1 e M4, considerando e removendo *stop words*. A modalidade M4 removendo as *stop words* obteve a melhor média geral de MoJoFM. Esta modalidade venceu ou empatou com as demais modalidade em 57 das 63 comparações instância-a-instância. M4 perdeu para M1 na instância *apache-ant* nas configurações considerando e removendo *stop words*. M4 perdeu para M2 removendo *stop words* em duas instâncias (*apache-lucene* e *joda-beans*), para M3 removendo *stop words* em uma instância (*stanford-nlp*) e considerando *stop words* em outra instância (*jung-algorithms*). A segunda melhor modalidade, M2 removendo *stop words*, venceu ou empatou em apenas 45 comparações instância-a-instância, quase 20% menos do que a modalidade M4.

Tabela 3.7: Percentual de palavras ignoradas, removendo e considerando as *stop words*, e usando o modelo de incorporação de palavras *Wiki-en* nas modalidades M1 a M4.

Software	Removendo <i>stop words</i>				Considerando <i>stop words</i>			
	M1	M2	M3	M4	M1	M2	M3	M4
apache-ant	4,87%	21,80%	24,50%	20,23%	4,75%	21,25%	23,90%	19,90%
apache-jena	16,37%	10,97%	10,97%	10,96%	16,07%	10,85%	10,85%	10,84%
apache-lucene	6,67%	5,99%	6,09%	6,11%	6,50%	5,90%	6,01%	6,02%
joda-beans	5,64%	3,26%	3,32%	3,37%	5,39%	3,11%	3,16%	3,21%
joda-convert	1,67%	1,04%	1,03%	1,03%	1,56%	0,97%	0,96%	0,96%
joda-money	0,98%	0,48%	0,47%	0,47%	0,91%	0,44%	0,44%	0,44%
joda-time	3,02%	2,28%	2,28%	2,27%	2,89%	2,20%	2,19%	2,19%
jung-algorithms	4,59%	2,57%	2,55%	2,69%	4,40%	2,46%	2,45%	2,58%
stanford-corenlp	27,98%	23,97%	24,13%	24,15%	27,71%	23,79%	23,95%	23,96%
Média	7,98%	8,04%	8,37%	7,92%	7,80%	7,89%	8,21%	7,79%

A média geral dos resultados de MoJoFM removendo *stop words* (50,30) é levemente superior à média dos resultados considerando *stop words* (49,17). Além disso, a modalidade M4 teve seus melhores resultados removendo *stop words* e estas palavras também estavam ausentes na modalidade que teve o segundo maior número de vitórias em uma comparação instância-a-instância. Assim, as próximas seções reportarão apenas resultados sem *stop words*.

3.6.2 Experimentos com Recombinações de Áreas do Código

Considerando os resultados dos experimentos com as modalidades M1 a M4, decidimos definir novos experimentos intercalando elementos ao invés de acumulá-los entre as modalidades. Para verificar se diferentes combinações dos elementos utilizados podem melhorar os valores de MoJoFM na comparação com a distribuição de classes em pacotes proposta pelos desenvolvedores, criamos novas modalidades, denominadas M5 a M12, da forma abaixo. Todas as modalidades consideram os nomes das classes conforme s-DEV.

- M5 considera os nomes das classes e seus atributos;
- M6 considera os nomes das classes e seus métodos;

Tabela 3.8: Valores de MoJoFM obtidos nos experimentos M1 a M4 com o modelo de incorporação de palavras *Wiki-en*.

Software	Considerando stop words				Removendo stop words			
	M1	M2	M3	M4	M1	M2	M3	M4
apache-ant	54,43	41,64	41,86	45,12	52,30	41,86	42,31	45,57
apache-jena	43,56	45,03	46,75	47,61	46,26	46,26	47,85	48,59
apache-lucene	46,36	49,50	47,82	49,83	45,69	51,51	50,73	50,95
joda-beans	29,59	40,82	37,76	41,84	33,67	44,90	42,86	43,88
joda-convert	75,86	75,86	75,86	75,86	75,86	75,86	75,86	75,86
joda-money	47,62	57,14	57,14	52,38	52,38	57,14	57,14	57,14
joda-time	66,67	62,26	66,04	57,23	66,04	62,89	58,49	68,55
jung-algorithms	32,67	40,59	46,53	41,58	31,68	41,58	39,60	44,55
stanford-nlp	31,26	32,33	32,44	33,21	32,74	32,74	36,06	33,27
Média	47,56	49,46	50,24	49,41	48,51	50,53	50,10	52,04

- M7 considera os nomes das classes, atributos e métodos;
- M8 considera os nomes das classes, as implementações de interfaces, as heranças, atributos, métodos e os comentários;
- M9 considera os nomes das classes e os comentários;
- M10 considera os nomes das classes, os comentários e as diretivas de importação;
- M11 considera os nomes das classes, as implementações de interfaces, as heranças e os comentários;
- M12 considera os nomes das classes, as implementações de interfaces, as heranças, os comentários e as diretivas de importação.

A Tabela 3.9 mostra o percentual de palavras ignoradas para os modelos M5 a M12 utilizando o modelo de incorporação de palavras *Wiki-en* e com a remoção de *stop words*. Observamos uma grande variação nos percentuais entre as instâncias: entre 15% e 25% das palavras da instância *apache-ant* são descartadas em todas as modalidades,

Tabela 3.9: Percentual de palavras ignoradas usando o modelo de incorporação *Wiki-en* para as modalidades M5 a M12 com a remoção de *stop words*.

Software	M5	M6	M7	M8	M9	M10	M11	M12
apache-ant	21,82%	19,50%	24,53%	20,22%	14,49%	14,52%	14,49%	14,52%
apache-jena	6,02%	3,16%	6,11%	10,96%	3,74%	3,72%	6,03%	6,02%
apache-lucene	4,27%	2,28%	4,42%	6,11%	1,66%	1,65%	4,26%	4,29%
joda-beans	0,73%	0,98%	0,80%	3,37%	0,93%	0,93%	0,73%	0,78%
joda-convert	0,32%	0,00%	0,31%	1,03%	0,00%	0,00%	0,32%	0,31%
joda-money	0,13%	0,00%	0,13%	0,47%	1,41%	1,39%	0,26%	0,26%
joda-time	1,81%	1,22%	1,80%	2,27%	0,55%	0,54%	1,81%	1,80%
jung-algorithms	1,58%	0,57%	1,61%	2,69%	2,45%	2,43%	1,58%	1,72%
stanford-nlp	10,81%	6,77%	11,47%	24,15%	4,89%	4,93%	10,91%	10,99%
Média	5,28%	3,83%	5,69%	7,92%	3,35%	3,35%	4,49%	4,52%

enquanto as instâncias *joda-money* e *joda-convert* descartam menos de 2% das suas palavras como *stop words*.

A Tabela 3.10 mostra os valores do MoJoFM para os experimentos M5 a M12 sem *stop words* e com o modelo de incorporação de palavras *Wiki-en*. Os valores baixos de MoJoFM para as modalidades M5, M6 e M7 mostram que os nomes dos atributos e métodos não contribuem isoladamente para a modularização do software. Por outro lado, os valores mais altos de MoJoFM para a modalidade M8 permitem concluir que os atributos, métodos, implementações de interface e heranças adicionam termos que, unidos aos termos provenientes dos comentários, são positivos para orientar a distribuição de classes em pacotes. Ainda assim, a média geral de M4 (52,04) fica acima da média de M8 (51,08) e M4 vence ou iguala as demais modalidades em 89 das 99 comparações com outras instâncias (90%), enquanto M8 vence ou iguala as demais modalidades em 77 das 99 comparações com outras instâncias (78%). Assim, M4 continua sendo a modalidade que apresenta os melhores resultados, indicando que todas as regiões do código-fonte contribuem com termos para a distribuição de classes em pacotes.

Tabela 3.10: Valores de MoJoFM para os experimentos M5 a M12 utilizando o modelo de incorporação *Wiki-en* sem *stop words*.

Software	M5	M6	M7	M8	M9	M10	M11	M12
apache-ant	37,26	39,96	38,16	46,35	44,11	44,11	46,58	46,02
apache-jena	45,28	40,61	43,80	47,12	42,21	40,86	45,77	46,01
apache-lucene	41,32	44,23	43,45	48,71	45,80	44,23	46,70	46,81
joda-beans	38,78	32,65	42,86	40,82	31,63	29,59	45,92	46,94
joda-convert	75,86	75,86	75,86	75,86	75,86	75,86	86,21	86,55
joda-money	47,62	61,90	61,90	57,14	52,38	52,38	52,38	57,14
joda-time	53,46	44,65	54,72	67,30	44,65	45,28	57,86	61,01
jung-algorithms	43,56	36,63	38,61	44,55	39,60	38,61	47,52	42,57
stanford-nlp	26,33	31,49	28,53	31,91	29,42	29,00	25,98	27,64
Média	45,50	45,33	47,54	51,08	45,07	44,44	50,55	51,18

3.6.3 Filtro por Classes Gramaticais

Part-of-speech tagging (PoS tagging) é o processo de classificar e rotular palavras em partes do discurso, como substantivo, verbo, adjetivo, advérbio, conjunção, pronome e outras categorias [95]. As partes do discurso também são conhecidas como “classes de palavras” ou “categorias lexicais”. Uma *tag*, neste contexto, corresponde a uma classe de palavras, como substantivo, adjetivo, verbo e assim por diante.

Nestes experimentos, aplicamos a técnica de *PoS Tagging* nos termos coletados dos sistemas de software selecionados, utilizando apenas as palavras classificadas como variações de substantivos pela biblioteca *NLTK* [15] no agrupamento. Substantivos foram escolhidos porque eles são a classe gramatical que melhor representa conceitos em um discurso. As variações dos substantivos na *NLTK* incluem: NN (substantivo, comum, singular), NNP (substantivo, comum, plural), NNPS (nome próprio, plural) e NNS (nome próprio, singular).

O resultado da aplicação do *PoS Tagging* e seleção apenas dos substantivos está apresentado na Tabela 3.11. Na média, os resultados limitados aos substantivos foram superiores aos resultados gerais nas modalidades M3, M5, M7 e M10 e inferiores nas oito modalidades remanescentes. Os resultados limitados a substantivos venceram ou

Tabela 3.11: Valores de MoJoFM para as modalidades M1 a M12 considerando somente os termos linguísticos da classe gramatical “substantivo”.

Software	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12
apache-ant	49,61	43,77	41,19	46,91	37,49	39,96	39,28	45,45	43,55	47,14	51,52	49,94
apache-jena	40,25	48,59	47,61	47,85	39,88	34,23	40,86	49,08	39,39	41,23	44,05	44,91
apache-lucene	49,61	50,06	49,05	50,95	46,36	42,44	46,92	49,05	40,87	39,42	48,82	47,37
joda-beans	26,53	44,90	46,94	39,80	38,78	28,57	43,88	39,80	30,61	29,59	25,51	31,63
joda-convert	75,86	75,86	75,86	75,86	75,86	75,86	75,86	75,86	75,86	75,86	75,86	75,86
joda-money	47,62	52,38	66,67	47,62	57,14	47,62	57,14	47,62	47,62	47,62	47,62	47,62
joda-time	51,57	63,52	55,35	52,20	55,97	55,97	56,60	49,69	52,83	57,23	51,57	46,54
jung-algorithms	37,62	42,57	45,54	45,54	42,57	33,66	45,54	42,57	38,61	41,58	37,62	37,62
stanford-nlp	29,54	33,10	33,87	33,75	25,15	30,37	28,77	35,23	27,34	28,53	29,66	31,44
Média	45,36	50,53	51,34	48,94	46,58	43,19	48,32	48,26	44,08	45,36	45,80	45,88

igualaram as respectivas modalidades sem a restrição do tipo de termo linguístico em 55 das 108 comparações instância-a-instância. Por fim, a média da modalidade M4 limitada a substantivos (48,94) ficou abaixo da média sem a imposição de limitação gramatical (52,04), esta última mantendo a superioridade em seis das nove instâncias.

Observando a Tabela 3.11, podemos perceber que M4 sem *stop words* e baseada no modelo de incorporação de palavras *Wiki-en* continua com o maior valor entre as modalidades analisadas, superior a s-DEV e menor que f-DEV. Ainda que esta modalidade não tenha atingido valores de MoJoFM tão altos quanto aqueles obtidos por f-DEV, observamos que os termos linguísticos utilizados em outras regiões do código-fonte aprimoram os resultados da remodelarização de software quando comparados aos resultados obtidos considerando apenas o nome da classe (s-DEV).

3.7 Experimentos com outros Algoritmos

O algoritmo utilizado nos experimentos das seções anteriores foi o agrupamento aglomerativo (HAC) com a métrica de distância euclidiana e o critério de ligação *Ward*. Esta seção tem como objetivo avaliar outros algoritmos de agrupamento apresentados na literatura e diferentes combinações de métricas e critérios de ligação para o algoritmo aglomerativo. Os algoritmos testados a seguir estão descritos na Seção 3.2.

A Tabela 3.12 mostra os valores de MoJoFM calculados para a modalidade M4 com o modelo de incorporação de palavras *Wiki-en*, o algoritmo de agrupamento

Tabela 3.12: Resultados do MoJoFM para a modalidade M4 usando o agrupamento aglomerativo, o modelo de incorporação de palavras *Wiki-en*, diferentes medidas de distância e critérios de ligação.

Software	EW	EC	EA	ES	MC	MA	MS	CC	CA	CS
apache-ant	45,57	17,17	31,99	36,70	17,28	33,33	38,05	17,40	33,33	40,74
apache-jena	48,59	6,01	32,02	41,96	6,26	34,48	40,00	5,64	34,23	42,09
apache-lucene	50,95	18,37	27,77	38,97	18,37	29,12	37,63	18,25	34,04	40,87
joda-beans	43,88	19,39	23,47	31,63	20,41	22,45	30,61	19,39	28,57	34,69
joda-convert	75,86	75,86	75,86	75,86	75,86	75,86	75,86	75,86	75,86	75,86
joda-money	57,14	47,62	47,62	52,38	47,62	52,38	61,90	47,62	47,62	47,62
joda-time	68,55	26,42	44,65	39,62	26,42	40,88	49,69	25,16	45,28	44,03
jung-algorithms	44,55	12,87	24,75	27,72	12,87	18,81	38,61	14,85	21,78	33,66
stanford-nlp	33,27	7,89	11,03	19,40	8,07	10,68	19,75	7,53	11,86	24,38
Média	52,04	25,73	35,46	40,47	25,91	35,33	43,57	25,74	36,95	42,66
Vitórias:	80	22	42	58	26	43	66	19	49	66

aglomerativo, diferentes medidas de distância e critérios de ligação. A primeira letra no nome de cada coluna representa a medida de distância, sendo “E” para distância euclidiana, “M” para distância Manhattan e “C” para distância do Cosseno. A segunda letra representa o critério de ligação utilizado, sendo “W” para *Ward*, “C” para *complete-linkage*, “A” para *average-linkage* e “S” para *single-linkage*.

Cabe notar que o uso do critério de ligação *Ward* tem sido tipicamente limitado à métrica de distância euclidiana entre as observações. Isso ocorre porque a função objetivo geralmente é escolhida para ser a variância mínima ou erro quadrado mínimo. A distância euclidiana está relacionada à soma dos erros quadrados e daí o uso dessa métrica com o critério Ward [?].

Observando a tabela, percebemos que os melhores resultados foram obtidos com a configuração EW, seguida pela configuração MS. Considerando que temos 90 resultados, sendo nove por configuração, a configuração EW superou ou igualou 80 das 81 adversárias (99%), enquanto a configuração MS superou ou igualou 66 adversárias (81%). Além disso, a configuração EW apresentou a maior média de MoJoFM para a modalidade M4, quase 20% maior que a média de MoJoFM da configuração MS. Assim,

Tabela 3.13: Resultados do MoJoFM para a modalidade M4 usando os algoritmos *HAC* com a configuração EW, *K-Means* (KM), *Birch* (BI) e *Spectral Clustering* (SC) e o modelo de incorporação *Wiki-en*

Software	HAC	KM	BI	SC
apache-ant	45,57	44,67	12,23	44,16
apache-jena	48,59	48,10	2,21	49,50
apache-lucene	50,95	49,05	17,58	50,64
joda-beans	43,88	42,86	8,16	42,84
joda-convert	75,86	75,86	79,31	75,85
joda-money	57,14	52,38	52,38	54,38
joda-time	68,55	55,97	25,79	66,26
jung-algorithms	44,55	40,59	6,93	46,44
stanford-nlp	33,27	31,14	14,71	29,53
Média:	52,04	48,95	24,36	51,06

a configuração do algoritmo HAC mais adequada para encontrar a distribuição de classes em pacotes de maior qualidade, utiliza a distância euclidiana e o critério de ligação *Ward*.

A Tabela 3.13 mostra os valores de MoJoFM para a modalidade M4 resultantes da distribuição de classes em pacotes pelos algoritmos *K-Means* (KM), *Spectral Clustering* (SC) e *Birch* (BI) utilizando o modelo de incorporação de palavras *Wiki-en*. Todos os algoritmos experimentados utilizam a métrica euclidiana para distância. Neste experimento, os melhores resultados com base na média geral foram obtidos com o algoritmo HAC (52,04) utilizando a métrica de distância euclidiana e o critério de ligação *Ward* e, em segundo lugar, bem próximo, o algoritmo *Spectral Clustering* (51,06). O algoritmo *K-Means* apresentou uma média cerca de 10% abaixo destes algoritmos, enquanto o algoritmo *BIRCH* apresentou uma média muito inferior.

3.8 Ameaças à Validade

De acordo com Wohlin et al. [122] existem quatro categorias de ameaças à validade do estudo: construção, interna, externa e conclusão.

A **validade de construção** trata da relação entre teoria e observação [38]. Para garantir esta diretiva, tanto o estudo teórico do problema como a métrica de eficácia (MoJoFM) foram tratadas no Capítulo 2 e os detalhes dos experimentos foram apresentados acima. O procedimento seguido para a seleção dos projetos a ser analisados buscou softwares conhecidos na comunidade de Engenharia de Software, estáveis, com disponibilidade dos código-fonte e organização bem estruturada dos pacotes (por exemplo, softwares com nomes de módulos de código ininteligíveis, como “*a.java*”, “*b.java*”, foram descartados).

A **validade interna** de um estudo é definida como a capacidade de um novo estudo replicar o comportamento do estudo atual [122]. Para garantir a reprodutibilidade do estudo atual, o código-fonte e as instâncias utilizadas nos experimentos serão disponibilizados em um repositório público ao final do estudo.

A **validade externa** é definida como a capacidade de generalização do estudo e se o estudo pode ser utilizado em um escopo diferente do definido [38]. Neste estudo, apenas sistemas de software abertos (*open-source*) foram explorados. Isso pode ameaçar a validade dos resultados. Modelos de desenvolvimento centralizados, como os sistemas usados em empresas de software comercial, são desenvolvidos com uma colaboração distribuída e organizada. Em comunidades abertas, os esforços de um grande número de desenvolvedores são distribuídos com objetivo de desenvolver softwares com pouca ou nenhuma centralização e controle. Neste caso, o ideal seria realizar uma investigação mais abrangente em diferentes sistemas de softwares comerciais para ampliar o escopo de generalização estendendo, com novas instâncias, com objetivos diversos e tamanhos variados, agrupadas em pequeno, médio e grande porte. No entanto, é mais difícil ter acesso ao código-fonte de sistemas de software comerciais. Além disso, as instâncias utilizadas nos experimentos realizados foram coletadas a partir de softwares escritos na linguagem Java, que pode limitar a generalização para sistemas escritos em outras linguagens de programação.

A **validade de conclusão** mede a relação entre os tratamentos e os resultados, determinando a capacidade do estudo em gerar alguma conclusão [122]. Neste sentido,

apenas nove sistemas de software foram testados nos experimentos reportados acima, quantidade que inviabiliza adotar testes de inferência estatística. Pretendemos aumentar e diversificar a quantidade de instâncias dos sistemas para garantir uma avaliação com a adoção dos testes estatísticos. Alguns parâmetros usados na vetorização foram definidos através da execução de experimentos sem uma heurística adequada. Os experimentos utilizaram um único valor para dimensão dos vetores. Não foi realizada redução da dimensionalidade do modelo para fazer a comparação.

3.9 Considerações Finais

Esta pesquisa apresentou uma sequência de estudos experimentais com a aplicação de um algoritmo hierárquico para a distribuição de classes em pacotes com base nos termos léxicos utilizados em diferentes regiões do código-fonte e em diferentes modelos de incorporação de palavras.

Nas primeiras avaliações, notamos que as modalidades baseadas no nome completo de classes, sem nome de domínio (s-DEV) e com nome de domínio (f-DEV), apresentaram resultados mais próximos às distribuições de classes em pacotes propostas pelos desenvolvedores dos softwares analisados. No entanto, a distribuição f-DEV carrega termos vindos dos nomes dos pacotes e, conseqüentemente, é influenciada pela distribuição proposta pelos desenvolvedores. A tendência é que classes que estejam no mesmo pacote na distribuição proposta pelos desenvolvedores participem do mesmo grupo proposto pelo algoritmo de agrupamento orientado pela modalidade f-DEV por compartilharem os termos originários dos nomes dos pacotes. Assim, esta modalidade foi considerada o limite superior que se poderia esperar chegar em relação à distribuição dos desenvolvedores a partir da inclusão de mais termos linguísticos em s-DEV.

Para os experimentos utilizando termos provenientes de outras regiões do código-fonte foram considerados os nomes das classes, as implementações de interfaces, heranças, atributos, métodos, comentários e importações de classes. Percebemos que os melhores resultados foram obtidos com a modalidade que considerou termos linguísticos provenientes de todas estas regiões do código-fonte. Observamos que a remoção de *stop*

words não implicou em variação substancial nos valores do MoJoFM resultantes das comparações das distribuições de classes em pacotes geradas pelo algoritmo de agrupamento hierárquico com a distribuição proposta pelos desenvolvedores. Por fim, nos estudos utilizando classificação gramatical (*PoS Tagging*) e processando apenas variações de substantivos pode-se perceber que estes elementos não mostraram melhoria no resultado final, o que sugere a realização de novos estudos futuros utilizando outras classes gramaticais.

O algoritmo utilizado inicialmente para os experimentos foi o agrupamento aglomerativo (HAC) com a métrica euclidiana e o critério de ligação *Ward*. Para comparar a eficácia deste algoritmo com outros algoritmos de agrupamento e outras configurações de métrica de distância e critério de ligação, realizamos uma série de experimentos envolvendo os algoritmos descritos na Seção 3.2 e diferentes combinações de métricas e critérios de ligação para o algoritmo aglomerativo. A partir dos resultados destes experimentos, percebemos que os melhores resultados foram obtidos com a aplicação do algoritmo aglomerativo com a distância Euclidiana e métrica *Ward* que será utilizado nas próximas etapas desta pesquisa.

4 REMODULARIZAÇÃO DE SOFTWARE E MODELOS DE INCORPORAÇÃO DE PALAVRAS ESPECÍFICOS

Resumo. O objetivo deste capítulo é apresentar e avaliar uma proposta para remodularização de software baseada em informações lexicais por meio de novos estudos baseados em diferentes conjuntos de sistemas de software e a criação de *corpus* para o treinamento de modelos de incorporação de palavras.

4.1 Introdução

O objetivo desta pesquisa é verificar se algoritmos de agrupamento baseados em informações léxicas representadas de forma vetorial conforme indicado por modelos de incorporação de palavras podem gerar distribuições de classes em pacotes compatíveis com as distribuições propostas pelos desenvolvedores de software. Realizamos experimentos iniciais utilizando o algoritmo *Iterated Local Search*, baseado em dados sobre os relacionamentos entre classes, e os algoritmos HAC e *Spectral Clustering*, baseados em termos léxicos colhidos de diferentes áreas do código-fonte do software.

Os resultados destes experimentos indicaram que a distância entre as distribuições de classes em pacotes geradas pelos algoritmos baseados em informações léxicas e a distribuição proposta pelos desenvolvedores, medida pela métrica *MoJoFM*, é grande. Para verificar se é possível chegar a distâncias menores, ou seja, valores de *MoJoFM* mais próximos de 100%, os próximos passos desta pesquisa envolvem o treinamento de modelos de incorporação de palavras próprios para este fim. Nos experimentos iniciais, utilizamos diferentes modelos de incorporação de palavras de uso geral. No entanto, é possível que um modelo treinado para a distribuição de classes em pacotes leve a resultados melhores do que os modelos de uso geral.

Para o treinamento de modelos de incorporação de palavras, um conjunto de projetos de software deve ser selecionado para montar o *corpus* usado no treinamento. Nos estudos reportados neste capítulo, usaremos projetos de software da *Apache*

Software Foundation. Nesta estratégia, a geração do *corpus* será feita extraíndo os termos linguísticos contidos em diferentes áreas do código-fonte dos projetos selecionados, organizando os termos provenientes de projetos de diferentes categorias. Os *corpora* criados com base em diferentes grupos de software serão então utilizados para o treinamento de modelos de incorporação de palavras baseados em modelos populares, como *Word2Vec*, *Doc2Vec* e *FastText*.

4.2 Geração de Modelos de Incorporação de Palavras

Um elemento essencial para tarefas de processamento de linguagem natural é um “corpus”. Em linguística, *corpus* (latim para *corpo*) refere-se a uma coleção de textos que pode ser formada por um único idioma ou pode abranger vários idiomas. Um corpus também pode consistir em textos temáticos (históricos, bíblicos, técnicos, entre outros) [82].

As características básicas de um *corpus* são o seu tamanho e a sua qualidade. Geralmente, considera-se que quanto maior o tamanho de um *corpus*, melhor. No entanto, é importante não priorizar a quantidade sobre a qualidade, pois o *corpus* ainda precisa considerar metadados precisos e conter informações anotadas. É importante observar que fornecer informações demais a um algoritmo de aprendizado de máquina, que é utilizado para o processamento do *corpus*, pode levar a resultados imprecisos e fazer com que o modelo se torne superajustado a um conjunto de dados (*overfitting*) [82]. O *overfitting* ocorre quando um modelo aprende os detalhes e o ruído tão bem que afeta negativamente a eficácia do modelo quando recebe novos dados, ou seja, a sua capacidade de generalização.

A qualidade dos dados de um *corpus* é de grande importância. Devido ao grande volume de dados necessário para construir um *corpus*, mesmo erros minúsculos nos dados de treinamento podem levar a erros em grande escala na saída do algoritmo de aprendizado de máquina [65]. Um bom *corpus* deve ter as seguintes características [33]:

- **Profundidade:** deve incluir uma grande quantidade de palavras que apareçam com frequência nos documentos que serão analisados utilizando o *corpus*;

- **Atualidade:** deve ser baseado em textos atualizados, evitando termos que se tornaram irrelevantes com o passar do tempo;
- **Gênero:** deve incluir diferentes origens de documentos (como jornais, revistas, *blogs*, periódicos acadêmicos, entre outros), desde que estas origens sejam relevantes para a aplicação pretendida para o *corpus*;
- **Tamanho:** um grande *corpus* aumenta as chances de que palavras de baixa frequência também sejam representadas adequadamente;
- **Clareza:** uma lista de palavras derivadas de um mesmo lema pode ser difícil de processar. Um *corpus* deve incluir apenas o lema, a parte invariante, desta lista.

Construído um *corpus*, é possível treinar um modelo de incorporação de palavras a partir dele. Uma das principais desvantagens do uso de modelos pré-treinados para tarefas de recuperação de informação é que eles podem ser treinados sobre dados que carregam significados irrelevantes para os domínios nos quais se destinam a ser usados, levando a resultados ambíguos [37]. Assim, o que se pretende nesta etapa da pesquisa é a criação de um modelo pré-treinado para a captura de conhecimento específico para o domínio da Engenharia de Software e voltado para o problema da modularização de software.

Para viabilizar a construção de um modelo de incorporação de palavras, utilizamos bibliotecas da linguagem Python, como *gensim*, *NLTK*, *scipy* e *sk-learn*. Uma descrição simplificada do processo para a criação de um modelo de incorporação de palavras envolve os seguintes passos:

- Selecionar os documentos (código-fonte de cada componente do sistema) que serão utilizados para compor o *corpus*;
- Definir as áreas do código-fonte de onde serão retirados os termos linguísticos, como os nomes de classes, métodos, atributos, entre outros;
- Extrair as sentenças dos documentos;

- Remover símbolos e pontuação;
- Extrair os termos linguísticos das sentenças;
- Criar um modelo, treinando a coleção e codificando os termos linguísticos como vetores;
- Salvar o modelo treinado.

Para aplicar os passos acima e criar novos modelos de incorporação de palavras, precisamos dos termos utilizados em projetos de software de boa qualidade. Para tanto, usamos os sistemas da *Apache Software Foundation*. A intenção é treinar modelos de incorporação de palavras para todos os projetos da Fundação e para os projetos de cada categoria, avaliando se os resultados da distribuição de classes em pacotes com os novos modelos superam os resultados com modelos de incorporação de palavras de uso geral.

4.3 Sistemas da *Apache Foundation*

A *Apache Software Foundation*¹ é uma instituição sem fins lucrativos, localizada nos EUA e administrada por um Conselho de Administração eleito por seus membros. Sua missão é criar sistemas de software para o bem público, que sejam distribuídos gratuitamente. A missão do projeto *Apache Community Development* é criar e fornecer ferramentas, processos e conselhos para ajudar projetos de software de código aberto a melhorar a sua comunidade de desenvolvedores e usuários.

O *Apache Projects Directory*² é um catálogo projetado para ajudar os usuários a encontrar projetos que atendam aos seus interesses e obter uma visão geral da variedade de trabalhos em andamento na comunidade Apache. Em fevereiro de 2023, a página que lista os projetos por seu nome mostrava 383 projetos, cada qual com um *link* para uma página com informações detalhadas a respeito do projeto. Os projetos também podem ser acessados por comitê, categoria, linguagem de programação e número de colaboradores.

¹ Apache Software Foundation - <https://apache.org/>

² Apache Projects Directory - <https://projects.apache.org/>

Por exemplo, ao acessar o projeto “Apache .NET Ant Library” obtemos uma página com a categoria do projeto, a URL do *website* do projeto, o *status* do projeto, as linguagens de programação com que ele foi desenvolvido e uma página com *links* para baixar as versões do seu código-fonte e executáveis. As versões do projeto podem ser baixadas utilizando o serviço próprio do Apache ou pela plataforma GitHub. Entretanto, nem todos os *links* dos projetos do diretório estão operacionais devido aos seguintes problemas mais recorrentes:

- **Endereços errados.** Diversos projetos indicam endereços errados para as suas páginas descritivas, seja por erro de digitação ou por endereços inválidos. Por exemplo, o projeto *Apache Commons RDF* apresenta no catálogo o endereço <https://commons.apache.org/rng/>. Porém, o endereço correto do projeto é <https://commons.apache.org/rdf/>;
- **Informação não encontrada.** Por exemplo, o projeto *Ambari Airavata* participa de três categorias, possui *website*, está com *status* ativo, foi desenvolvido primordialmente na linguagem *Java* e possui um *link* para baixar o código-fonte. Porém, ao clicar no *link*, obtém-se o erro *Not Found*. Neste caso, o *link* para baixar o código pode ser obtido acessando o *website* do projeto ³;
- **Dados duplicados.** Por exemplo, o projeto *DataFu* está descrito com dois estados diferentes (ativo e encubado) em diferentes *websites*;
- **Múltiplos sub-projetos.** O *link* do projeto *Directory* presente no catálogo aponta para uma página contendo para dois subprojetos: *Directory Server* e *Directory Studio*. Estes dois projetos foram tratados como projetos separados nas análises realizadas;
- **Código-fonte não disponível.** Os *links* para baixar o projeto (*Releases* e *Website*) estão inválidos. Por exemplo, o *website* do projeto “*Chainsaw*” mostra o erro “*Not Found*”. O *link* para baixar versões do projeto aponta para o índice dos produtos

³ <http://airavata.apache.org>

que podem ser baixados via projetos da Fundação, porém, não existe um *link* para o projeto “*chainsaw*” na lista.

A partir dos projetos listados por nome no *Apache Project Directory*, criamos uma base de dados com um registro para cada projeto, contendo as seguintes informações: nome do projeto, categorias do projeto, *website* do projeto, *status* do projeto, linguagem de programação, *links* para baixar as versões e seu código-fonte, sendo que os campos de categoria e linguagem podem conter múltiplos itens.

Em uma primeira leitura, foram contabilizados 378 projetos, sendo que 32 tinham o *status Incubating* e 54 tinham o *status Retired*. Um projeto *Incubating* é um projeto proposto para inclusão na Fundação, em desenvolvimento ativo, mas que ainda não está incluído no processo de liberação de versões. Um projeto *Retired* foi encerrado por iniciativa do conselho e não é mais desenvolvido. Somente projetos ativos foram considerados a partir deste ponto da pesquisa, restando 292 projetos.

Contabilizando apenas os projetos com código-fonte disponível, a lista inicial ficou restrita a 275 projetos. Porém, cinquenta e quatro destes projetos não possuem categoria definida (segundo o *website* da fundação em agosto de 2022). Portanto, existem 221 projetos ativos, com código-fonte disponível e categoria definida. Estes projetos são classificados em 43 categorias, conforme a Tabela 4.1, que mostra o número de projetos por categoria. A disponibilidade de código-fonte permite extrair os termos linguísticos das diversas áreas do código. A associação a pelo menos uma categoria é importante porque permitirá que sejam treinados modelos de incorporação de palavras para cada categoria, além de um modelo baseado em todos os projetos. Portanto, somente os 221 projetos ativos, com código-fonte disponível e categoria definida seguirão sob análise.

Depois de baixar o código-fonte dos 221 projetos remanescentes, eles foram avaliados utilizando a ferramenta CLOC⁴. Esta ferramenta identificou o uso de 49 linguagens de programação distintas nos projetos. A ferramenta também identificou o

⁴ A ferramenta *CLOC* conta linhas em branco, linhas de comentários e linhas físicas de código-fonte em muitas linguagens de programação (<https://github.com/AlDanial/cloc>).

Tabela 4.1: Número de projetos por categoria na Fundação Apache.

Categoria	#	Categoria	#	Categoria	#	Categoria	#
library	87	java	6	python	3	html	1
bigdata	35	graphics	4	testing	3	ide	1
network-server	33	httpd-module	4	identity-provisioning	2	kerberos	1
database	24	search	4	integration	2	mobile	1
xml	24	security	4	mail	2	observability	1
network-client	19	c++	3	sdk	2	php	1
build-management	16	geospatial	3	sql	2	templating	1
web-framework	16	go	3	c	1		
content	15	hadoop	3	data-management-platform	1		
cloud	13	identity-management	3	distributed-sql-database	1		
http	12	iot	3	ftp	1		
javaee	7	osgi	3	groovy	1		

número de arquivos escritos com determinada linguagem de programação, o número de linhas em branco, o número de linhas em comentário e o número de linhas de código por linguagem de programação.

A Tabela 4.2 mostra o número de projetos por linguagem de programação, enquanto a Tabela 4.3 mostra o número de linhas de código no conjunto dos projetos por linguagem de programação. Percebemos a predominância da linguagem de programação Java, com mais de 4,5 vezes o número de linhas de código do que a segunda linguagem de programação (C/C++). Existem, por sua vez, 4,9 vezes mais linhas de código escritas em C/C++ do que na próxima linguagem de programação, JavaScript. A partir daí, a concentração em outras linguagens de programação não é tão evidente.

Nesta etapa da pesquisa, decidimos que somente as categorias com mais de dez projetos seriam utilizadas em nossas análises. O racional é que categorias com poucos projetos oferecerão pouca informação para a criação de modelos de incorporação de palavras próprios, que poderiam ser muito específicos para os poucos projetos destas categorias. Permanecem, então, as categorias: *library*, *big-data*, *network-server*, *database*, *xml*, *network-client*, *build-management*, *web-framework*, *content*, *cloud* e *http*.

Aplicando o mesmo critério para as linguagens de programação, ou seja, analisar apenas as linguagens usadas em dez ou mais projetos, foram selecionadas *Java*, *JavaScript*, *C/C++*, *Python*, *Groovy*, *Perl*, *Ruby*, *Scala*, *PHP*, *TypeScript*, *C#* e *Go*.

Tabela 4.2: Número de projetos por linguagem de programação na Fundação Apache.

Linguagem	#	Linguagem	#	Linguagem	#	Linguagem	#
Java	194	Lisp	7	C_Shell	2	Html	1
JavaScript	102	Kotlin	6	CoffeeScript	2	Haskell	1
C/C++	102	Lua	5	Dart	2	MATLAB	1
Python	79	Pascal	5	Elixir	2	Objective-C	1
Groovy	41	Vue.js_Component	5	Erlang	2	Objective-C++	1
Perl	33	Clojure	4	Swift	2	OCaml	1
Ruby	32	Prolog	4	Tcl/Tk	2	Oracle_Reports	1
Scala	22	Assembly	3	ASP	1	SAS	1
PHP	16	Cython	3	AspectJ	1	Smalltalk	1
TypeScript	15	Rust	3	Blade	1	Visual_Basic	1
C#	12	Scheme	3	Delphi_Form	1		
Go	12	Visual_Basic_.NET	3	F#	1		
R	8	ASP.NET	2	Grails	1		

Tabela 4.3: Número de linhas de código (LOC) por linguagem de programação nos projetos da Fundação Apache.

Linguagem	LOC	Linguagem	LOC	Linguagem	LOC	Linguagem	LOC
Java	40.484.743	Ruby	93.107	Assembly	4.525	C_Shell	155
C/C++	8.928.577	Rust	23.558	Visual_Basic_.NET	3.606	MATLAB	139
JavaScript	1.796.113	Dart	22.882	Swift	3.552	Oracle_Reports	127
Scala	1.661.921	R	18.447	Objective-C	2.702	Grails	113
Python	1.655.740	Elixir	17.805	CoffeeScript	2.028	ASP.NET	106
Go	1.149.615	Visual_Basic	16.292	Clojure	1.288	Haskell	93
C#	885.756	Pascal	14.927	OCaml	962	SAS	88
PHP	535.239	Blade	12.239	Lisp	802	AspectJ	16
Perl	388.597	Tcl/Tk	8.354	Smalltalk	618	Delphi_Form	4
Groovy	311.832	Kotlin	8.077	Html	489	F#	2
Erlang	258.738	Objective-C++	6.171	Scheme	321		
TypeScript	111.965	Cython	6.095	ASP	284		
Vue.js	102.575	Lua	5.403	Prolog	257		

Selecionando apenas os projetos que contêm pelo menos uma destas linguagens de programação, continuamos com 221 projetos.

Para analisar se um modelo de incorporação de palavras desenvolvido para uma categoria pode ser útil para a modularização de software, precisamos considerar projetos desenvolvidos segundo a orientação a objetos e que usem linguagens de programação que definam os conceitos de classes e pacotes (ou equivalentes). Adicionando esta restrição às linguagens de programação selecionadas acima, finalizamos a lista de linguagens com *Java*, *C++*, *Groovy*, *Perl*, *Ruby*, *Scala*, *TypeScript*, *C#* e *Go*. Após a aplicação destes critérios, ficamos com um total de 186 projetos.

Analisando os 186 projetos remanescentes, observamos que todos eles utilizam a linguagem de programação Java. Considerando o número de arquivos de código-fonte, Java representa em média 92,5% do tamanho dos projetos, variando de 0,2% a 100% dos seus arquivos. A concentração dos projetos na linguagem de programação Java é intensa a ponto de que mais de 50% dos arquivos de código-fonte de 175 dos 186 projetos (94%) estão implementados em Java. Por outro lado, nenhum projeto contém mais de 50% dos seus arquivos de código-fonte implementados em *C#*, *Perl*, *Ruby* ou *TypeScript*. Apenas um projeto tem mais de 50% do código-fonte implementado em *Groovy* (projeto *Groovy*). Apenas dois projetos têm mais de 50% do código-fonte implementado em *Scala* (projetos *Daffodil* e *Spark*) e *Go* (projetos *PLC4X* e *Traffic Control*). Por fim, apenas quatro projetos têm mais de 50% dos arquivos de código-fonte implementados em *C++* (projetos *Kudu*, *Mesos*, *OpenOffice* e *Traffic Server*).

Considerando a distribuição dos projetos por linguagem de programação e o esforço necessário para processar o código-fonte de cada linguagem de interesse, optamos por analisar apenas o código-fonte escrito em Java e projetos com pelo menos 10 arquivos de código-fonte nesta linguagem. Neste recorte, três projetos foram descartados. Os 183 projetos remanescentes estão listados no Apêndice A.

Em seguida, realizamos uma busca manual pelo pacote raiz das classes de interesse em cada projeto selecionado. Por exemplo, o projeto *apache-yetus-0.14.1* tem como pacote raiz “org.apache.yetus”. Esta ação foi necessária pois, apesar de

existirem regras de nomenclatura ⁵ para os projetos da Apache Foundation, alguns projetos não seguem estas regras. Por exemplo, uma das recomendações indica o uso da palavra “*apache*” no nome do projeto e do seu pacote raiz. No entanto, projetos como *opennlp* e *org.jclouds* não seguem esta recomendação. Somente as classes abaixo do pacote raiz de cada projeto serão consideradas nas análises a seguir.

Para cada projeto selecionado, analisamos todos os arquivos de classes utilizando um analisador sintático que carrega os arquivos com extensão “.java” dos projetos e gera um arquivo CSV contendo as seguintes informações: (i) nome do projeto; (ii) nome do pacote; (iii) nome da classe principal sem processamento; (iv) palavras dos nomes de todas as classes no arquivo; (v) palavras da seção de *imports*; (vi) palavras da seção *implements*; (vii) palavras da seção *extends*; (viii) palavras dos métodos das classes; (ix) palavras dos atributos das classes; (x) palavras dos comentários; (xi) erros do analisador sintático; e (xii) número de classes.

Os projetos selecionados têm em média 1.498 arquivos de código-fonte (mediana 634, mínimo 12 e máximo 21.539). Destes arquivos, foram extraídos em média 315.209 termos linguísticos por projeto (mediana 131.044, mínimo 137 e máximo 4.052.348). O número de classes e o número de termos por projeto estão listados no Apêndice A.

4.4 Desenvolvimento dos modelos de incorporação de palavras

Para a criação de modelos de incorporação de palavras, precisaremos de dois conjuntos de dados: o **conjunto de treinamento**, que contém os dados que serão alimentados no modelo durante o processo de treinamento, e o **conjunto de teste**, que contém os dados com os quais o modelo treinado será validado. Para produzir estes dois conjuntos, digitalizamos o código-fonte dos projetos selecionados na seção anterior e, para cada projeto, geramos um arquivo no formato **CSV** onde cada linha contém as palavras capturadas das diversas áreas de código-fonte de uma classe.

Para cada projeto, o analisador sintático identifica seções como o nome do pacote, o nome da classe, importações de arquivos e comentários. As colunas

⁵ Project and product names - Apache Infrastructure Website (<https://infra.apache.org/project>)

projectname, *packagename* e *mainclassname* mantêm a sintaxe original dos respectivos nomes de projeto, pacote e classes (sem divisões). A partir da coluna *classnames*, os elementos são quebrados em palavras simples usando o padrão *camel-case*. A Figura 4.1 mostra um exemplo de arquivo processado para o projeto `org.apache.anakia`, com os dados extraídos de uma classe do projeto, removendo *stop words* e palavras duplicadas.

mainclassname	classnames	imports	extends
AnakiaElement	element, anakia	org, element, namespace, xml,	element
AnakiaJDOMFactory	anakia, jdom, factory	org, element, namespace, fact	jdom, factory
AnakiaTask	anakia, task, context	utils, directory, scanner, builde	task, matching
Escape	escape		
NodeList	attribute, node, xml, list, o		outputter, xml
OutputWrapper	output, wrapper	org, outputter, element, format	outputter, xml
TreeWalker	walker, tree	org, element, collection, util, ite	
XPathCache	path, cache	hash, weak, map, path, werken,	
XPathTool	path, tool	org, element, document, util, lis	
AnakiaTestCase	anakia, test	test, org, file, framework, junit,	test
Log	log		
TestUtil	test, util	file, io	

Figura 4.1: Um exemplo de um arquivo CSV para o projeto `org.apache.anakia`.

Dos 183 projetos selecionados na Seção 4.3, escolhemos os 78 projetos pertencentes à categoria “*library*” dividindo nos dois conjuntos: 80% dos projetos para o conjunto de treinamento e 20% para o conjunto de testes. A partir destes conjuntos, geramos modelos de incorporação de palavras pelos algoritmos *Word2Vec*, *Doc2Vec* e *FastText* e avaliados na modalidade M4.

O racional subjacente aos modelos de incorporação de palavras é que palavras que ocorrem em contextos semelhantes tendem a estar mais próximas umas das outras no espaço vetorial. Os algoritmos aprendem palavras prevendo o contexto circundante. Para tanto, é necessário definir a unidade de treinamento dos modelos. Segundo Kuchenbecker [61], *sentença* é uma unidade mínima de comunicação que produz efeitos de sentido em um contexto específico. Em termos linguísticos, *sentença* é o mesmo que *frase*. Segundo Marques [80], *frase* é um enunciado de sentido completo e pode ser formada por uma ou mais palavras. Usaremos a expressão “**sentença**” para definir uma *coleção de palavras* a partir deste ponto. De forma análoga, definimos **parágrafo** como uma coleção de sentenças e **documento** como uma coleção de *parágrafos*.

Os modelos de incorporação de palavras foram desenvolvidos inicialmente para trabalhar com textos convencionais, isto é, textos legíveis por humanos com significados bem definidos, onde a semântica infere expressivamente no resultado vetorial. No entanto, hoje são aplicados a textos não-convencionais, como códigos-fonte de linguagens de programação. As diferenças semânticas entre textos convencionais e códigos de linguagens de programação estão principalmente na **função**, na **ambiguidade**, na **gradação de significados** e na **sintaxe**.

- **Função**

- **Textos convencionais** têm como principal função a comunicação humana. Eles buscam transmitir ideias, emoções, informações ou histórias, sendo normalmente interpretados com base no contexto, cultura e experiência do leitor. A semântica de um texto convencional é mais fluida e depende de nuances, ambiguidade e interpretação subjetiva.
- **Códigos de programação** tem como função executar tarefas específicas em um computador. A semântica de um código é bem mais estrita e formal, já que o código deve ser interpretado exatamente da maneira como foi escrito, de acordo com a sintaxe e a lógica da linguagem de programação utilizada. Um pequeno erro no código pode impedir a execução ou gerar resultados inesperados.

- **Contexto e Ambiguidade**

- Nos **textos convencionais** as palavras podem ter múltiplos significados dependendo do contexto. Por exemplo, a palavra “banco” pode se referir a uma instituição financeira ou a um assento. A ambiguidade é comum e depende do entendimento humano.
- Nos **códigos de programação** as palavras (*tokens*) e comandos são precisas e têm significados fixos. Por exemplo, “print” em Python sempre indicará uma

instrução para exibir algo na tela, independentemente do contexto de uso deste comando, e “if” é sempre uma estrutura de decisão.

- **Gradação de Significados**

- O significado dos **textos convencionais** pode ser rico, múltiplo e aberto à interpretação. O mesmo texto pode ser lido e compreendido de diferentes formas por diferentes pessoas, ou até ter interpretações distintas ao longo do tempo.
- No caso dos **códigos de programação**, a semântica é muito mais rígida. Um código tem um objetivo bem definido, e seu significado é determinado pelas regras da linguagem de programação, sem espaço para múltiplas interpretações.

- **Sintaxe**

- A sintaxe de **textos convencionais** pode ser flexível. Em muitos casos, há uma variedade de formas de construir uma frase ou expressar uma ideia.
- Nos **códigos de programação**, a sintaxe é estrita. Cada linguagem de programação possui regras rigorosas de como os comandos e as estruturas devem ser organizados. Um erro sintático (como esquecer um ponto e vírgula ou uma chave) pode invalidar o código.

Enquanto a semântica de um texto convencional é subjetiva, aberta à interpretação e depende do contexto humano, a semântica de um código de programação é precisa, determinística e segue um conjunto de regras formais para garantir que as instruções sejam executadas corretamente por uma máquina.

Assim, definimos duas unidades de treinamento para cada algoritmo gerador de modelos de incorporação de palavras: **parágrafo** e **documento**. Na unidade de treinamento por parágrafos, cada projeto é representado por um conjunto de parágrafos, sendo um parágrafo para cada classe. O parágrafo de uma classe conterá uma sentença

para cada seção do arquivo CSV que representa o projeto. Em seguida, os parágrafos serão submetidos aos algoritmos *Doc2Vec*, *Word2Vec* e *FastText*. Portanto, para cada projeto, teremos uma matriz de n classes, onde cada classe é representada por um parágrafo. Na unidade de treinamento por documentos, cada projeto é representado por um único documento, cujas sentenças são construídas a partir do texto completo extraído dos arquivos de classes. Portanto, teremos uma matriz de n projetos, onde cada projeto é representado pelo documento construído. Em seguida, os documentos são submetidos aos algoritmos *Doc2Vec*, *Word2Vec* e *FastText*.

Além da unidade de treinamento, é necessário decidir sobre o uso de palavras repetidas e o descarte de palavras sem significado (*stop words*), uma vez que estas decisões podem afetar os resultados do processo de treinamento dos modelos de incorporação de palavras. Por exemplo, considere as sentenças na Figura 4.2, que foram utilizadas no treinamento de três modelos de incorporação de palavras. O modelo Tipo 1 utilizou as sentenças conforme apresentadas na figura; o modelo Tipo 2 descartou as palavras repetidas; por fim, o modelo Tipo 3 colocou as palavras em ordem alfabética antes de apresentar ao treinamento. A Tabela 4.4 apresenta a distância (cosseno) entre os vetores que representam as palavras “*pie*” e “*dessert*” nos três modelos de incorporação de palavras treinados, mostrando a diferença entre estes modelos. Em função destas diferenças, serão treinados modelos com e sem palavras sem significado e modelos com e sem palavras repetidas.

```
(sentença-1): I like apple pie for dessert and I like banana pie too  
(sentença-2): I don't drive fast cars I prefer to drive slow cars  
(sentença-3): data science is very very fun  
(sentença-4): chocolate is my favorite  
(sentença-5): my favorite movie is Starwars rather than Predator
```

Figura 4.2: Texto exemplo para cálculo de similaridades

Por fim, os algoritmos de treinamento de modelos de incorporação de palavras disponibilizam vários parâmetros para a configuração do treinamento e construção dos

Tabela 4.4: Distâncias entre as palavras “*pie*” e “*dessert*” em diferentes configurações de treinamento de modelos de incorporação de palavras.

Tipo 1	Tipo 2	Tipo 3
1,0837	1,0689	0,6810

Tabela 4.5: Alguns parâmetros usados na vetorização

Parâmetro	Valor	Definição
<i>Window (geral)</i>	5	Distância máxima entre a palavra atual e a prevista dentro de uma sentença.
<i>min_count (geral)</i>	10	Ignora as palavras com frequência total inferior a esta.
<i>sg (Word2Vec e FastText)</i>	SG	Algoritmo de treinamento: SG (skip-gram) ou CBOW.
<i>dm (Doc2Vec)</i>	1	Algoritmo de treinamento: DM (memória distribuída) ou DBOW (saco de palavras distribuído).

vetores que representam as palavras. Vários valores foram testados para os diferentes parâmetros em diversos experimentos, dentro do domínio de cada parâmetro, para obter a melhor eficácia em cada algoritmo. A Tabela 4.5 mostra os parâmetros com os valores escolhidos para os treinamentos que serão relatados a partir da próxima seção.

4.4.1 Validação cruzada

A validação cruzada (*cross-validation*) é um procedimento de reamostragem usado para avaliar modelos de aprendizado de máquina em uma amostra de dados limitada. O procedimento tem um único parâmetro, chamado k , que se refere ao número de grupos em que uma determinada amostra de dados deve ser dividida. Como tal, o procedimento é muitas vezes chamado de validação cruzada *k-fold* (*k-fold cross-validation*). Quando um valor específico para k é escolhido, ele pode ser usado no lugar de k na referência ao modelo. Por exemplo, se $k = 10$ o procedimento também pode ser referenciado como validação cruzada de 10 vezes [20].

Dos projetos selecionados, sorteamos 10 pares de conjuntos de treinamento e teste na proporção de 80% e 20%, respectivamente. Cada conjunto de treinamento totalizou 62 projetos e cada conjunto de testes incluiu 16 projetos. Aplicando os

algoritmos *Word2Vec*, *Doc2Vec* e *FastText*, executamos 24 experimentos que produziram modelos de incorporação de palavras nomeados de acordo com a estrutura “[unidade]-[stopwords]-[palavras]”, onde unidade indica a unidade de treinamento (PAR para parágrafo ou DOC para documento), stopwords indica se as *stop words* serão removidas ou mantidas como parte dos modelos (KEEP para manter ou REMV para remover) e palavras indica se as palavras repetidas serão descartadas ou mantidas (UNIQ e REPT, respectivamente). Cada um destes modelos será doravante chamado de uma configuração de treinamento. O tempo total de execução dos experimentos foi de aproximadamente cinco dias.

A Tabela 4.6 mostra a eficácia dos algoritmos *Word2Vec*, *Doc2Vec* e *FastText* na modalidade M4, usada na avaliação. Os resultados de cada algoritmo estão representados no padrão $\mu \pm \sigma$, onde μ representa a média do *MoJoFM* do conjunto de teste ao longo das 10 rodadas da validação cruzada e σ representa o desvio padrão do *MoJoFM* ao longo destas 10 rodadas. O melhor resultado entre os três algoritmos de treinamento para cada configuração, considerando as médias, é apresentado em negrito.

Observando a Tabela 4.6, podemos verificar que a pior eficácia ficou com o algoritmo *FastText*, que obteve apenas uma pontuação superior aos demais algoritmos (na configuração de treinamento PAR-KEEP-UNIQ). A melhor eficácia ficou com o algoritmo *Word2Vec*, com cinco pontuações maiores do que os outros algoritmos. No entanto, aplicando-se um teste de inferência estatística de Wilcoxon [58] entre as séries de *MoJoFM* produzidas pelas dez rodadas de cada algoritmo em cada configuração, observamos diferenças estatisticamente significativas entre as médias dos algoritmos *Word2Vec* e *Doc2Vec* em três configurações (PAR-KEEP-UNIQ, PAR-REMV-UNIQ e DOC-KEEP-UNIQ) e entre as médias dos algoritmos *Word2Vec* e *FastText* em duas configurações (PAR-KEEP-UNIQ e DOC-KEEP-UNIQ), nesses casos contrárias ao algoritmo *Word2Vec*. Nenhuma outra comparação entre as médias apresentou diferenças estatisticamente significativas.

Assim, o algoritmo *Word2Vec* possui um maior número de vitórias entre os algoritmos, mas essas vitórias não são estatisticamente significativas. Por outro lado,

quando este algoritmo produz resultados inferiores aos seus pares, a diferença dos demais é estatisticamente significativa. A Figura 4.3 apresenta um *boxplot* para a configuração PAR-REMV-UNIQ, mostrando que as diferenças entre os resultados produzidos pelo algoritmo *Word2Vec* e os resultados do *Doc2Vec* e do *FastText* são estatisticamente significativas e desfavoráveis ao *Word2Vec*. Assim, a decisão sobre o melhor algoritmo depende de uma escolha prévia da configuração de treinamento, que será realizada na próxima seção.

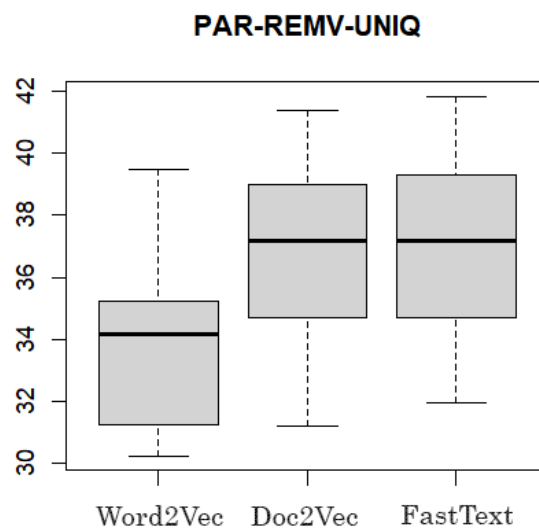


Figura 4.3: Um *boxplot* dos valores de MoJoFM produzidos pelos três algoritmos para a configuração PAR-REMV-UNIQ.

Podemos observar que as médias dos valores de *MoJoFM* reportados na Tabela 4.6 variam entre 33,57 e 44,37. Para verificar se o treinamento de modelos de incorporação de palavras traz algum benefício para a modularização de projetos de software quando comparado com o uso de representações aleatórias para os termos linguísticos, realizamos um experimento em que cada palavra identificada na modalidade M4 nos projetos envolvidos foi representada por um vetor de 300 números gerados aleatoriamente (mesmo tamanho do vetor usado pelo algoritmo *Word2Vec*). Em seguida, estes vetores foram utilizados como entrada para o algoritmo HAC visando gerar novas

Tabela 4.6: Eficácia dos algoritmos de treinamento de modelos de incorporação de palavras na validação cruzada com a modalidade M4.

Unidade	Stopwords	Repetidas	Word2Vec	Doc2Vec	FastText
PAR	KEEP	REPT	35,54 ± 3,23	34,74 ± 3,06	34,48 ± 3,17
PAR	KEEP	UNIQ	39,43 ± 4,26	44,24 ± 3,52	44,37 ± 3,79
PAR	REMV	REPT	36,34 ± 2,79	35,73 ± 3,01	36,23 ± 3,10
PAR	REMV	UNIQ	34,436 ± 3,19	37,32 ± 3,23	37,25 ± 2,95
DOC	KEEP	REPT	34,39 ± 3,68	34,16 ± 2,63	34,19 ± 2,67
DOC	KEEP	UNIQ	33,57 ± 2,86	36,50 ± 3,16	36,12 ± 2,85
DOC	REMV	REPT	35,06 ± 2,92	34,60 ± 2,72	34,69 ± 2,89
DOC	REMV	UNIQ	35,31 ± 3,13	34,73 ± 2,71	34,60 ± 2,77

distribuições de classes em pacotes para os dez conjuntos de 16 projetos da Apache Software Foundation utilizados como teste na validação cruzada.

Geramos dez vetores aleatórios para cada palavra, formando dez modelos de incorporação de palavras aleatórios utilizados no agrupamento, para reduzir um possível viés decorrente do uso de números aleatórios. Para cada conjunto, o *MoJoFM* do agrupamento resultante de cada projeto foi anotado. Observamos uma variação entre 25,66 e 27,77 nestes valores de *MoJoFM*, com média 26,77 e desvio padrão de 0,59. Assim, concluímos que o treinamento de modelos de incorporação de palavras gera resultados substancialmente melhores do que o uso de representações aleatórias para os termos linguísticos.

A Figura 4.4 mostra a distribuição de médias, mínimo e máximo dos valores de *MoJoFM* para cada projeto nas rodadas do estudo experimental na modalidade M4. Cada linha do gráfico representa um projeto, com seu nome no lado esquerdo e seu número de classes do lado direito. Os projetos estão ordenados pelo número de classes, a média dos valores de *MoJoFM* é representada como um ponto escuro e o intervalo entre o mínimo e o máximo do *MoJoFM* é representado por um traço contínuo. Podemos notar que dois projetos com quantidade próxima de classes podem apresentar valores de *MoJoFM* muito diferentes. Este comportamento irregular se deve às características

individuais de cada projeto, como a distribuição das funcionalidades pelas classes, o uso de palavras representativas na codificação, entre outras. Como os três algoritmos analisados (*Word2Vec*, *Doc2Vec* e *FastText*) apresentam um comportamento similar, mostramos apenas a figura para o algoritmo *Word2Vec*.

Também podemos observar na Figura 4.4 que a média do *MoJoFM* decresce a medida que o número de classes aumenta. Para avaliar esta hipótese, realizamos um teste não paramétrico usado para medir o grau de associação entre duas variáveis. O teste de correlação de *Spearman* não contém suposições sobre a distribuição dos dados e, desta maneira, podemos verificar se o valor do *MoJoFM* diminui com o aumento no número de classes, mesmo que a distribuição em frequência destes dados não seja próxima a uma curva normal. Para este teste, usamos os projetos da categoria *library* nas configurações KEEP-REPT, KEEP-UNIQ, REMV-REPT e REMV-UNIQ para as unidades de treinamento PAR e DOC e os algoritmos *Word2Vec*, *Doc2Vec* e *FastText*.

Calculamos 24 correlações entre o número de classes de um projeto e a média dos valores de *MoJoFM* gerados pela otimização deste projeto com os modelos de incorporação de palavras resultantes das dez rodadas de treinamento com determinada configuração e algoritmo. Os valores observados nestas correlações variam de -0,620 a -0,509, ou seja, podem ser considerados como correlação moderada ou forte e negativa. O coeficiente negativo indica que uma variável tende a aumentar à medida que a outra diminui. Estes valores confirmam a suposição representada na Figura 4.4 de que os valores do *MoJoFM* diminuem conforme o número de classes aumenta.

4.4.2 Análise de dominância das configurações

Apesar dos pequenos desvios padrão apresentados na Tabela 4.6, observamos uma grande variação no *MoJoFM* entre as rodadas de treinamento de cada algoritmo para algumas instâncias. Por exemplo, o projeto `org.apache.anakia` obteve *MoJoFM* entre 50 e 90 nas dez rodadas. O projeto `org.apache.commons.scxml` variou de 17,6 a 35,3 e o projeto `org.apache.yetus` teve uma variação de 37,5 a 100. Assim, esta seção investigará o comportamento de cada configuração de treinamento utilizada nos estudos

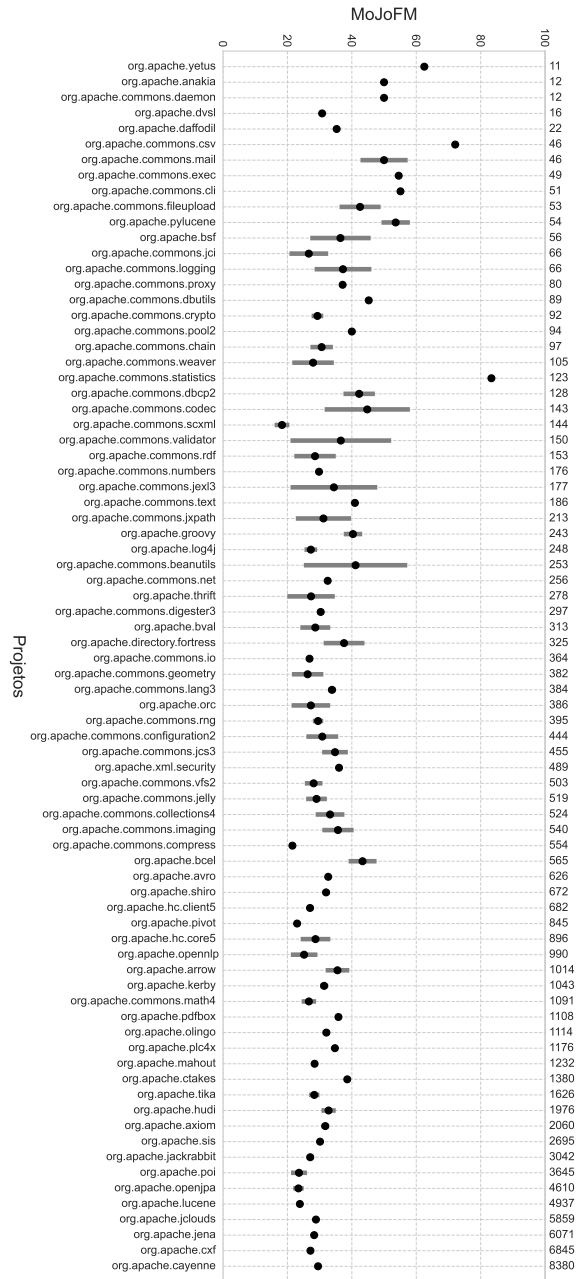


Figura 4.4: Média, mínimo e máximo do *MoJoFM* de cada projeto para o algoritmo *Word2Vec* de acordo com o número de classes.

experimentais para selecionar aquela que gera distribuições de classes em pacotes mais próximas às utilizadas pelos desenvolvedores.

No estudo aqui reportado comparamos as configurações de treinamento em pares, projeto a projeto. Uma configuração “A” é considerada *dominante* sobre uma configuração “B” em um projeto se o *MoJoFM* gerado para aquele projeto a partir de todos os modelos de incorporação de palavras na configuração “A” for maior ou igual ao *MoJoFM* gerado para aquele projeto a partir do mesmo modelo de incorporação de palavras na configuração “B”, sendo que a condição de igualdade deve cair para pelo menos um dos modelos. Caso contrário, o par de configurações é considerado *não-dominante* no projeto.

Formalmente, podemos dizer que o resultado da otimização da distribuição das classes em pacotes de um projeto P_i utilizando os modelos de incorporação de palavras gerados pelo treinamento baseado no algoritmo A_j e na configuração C_k é a média do *MoJoFM* gerado pelo otimizador usando os dez modelos de incorporação de palavras produzidos nas rodadas de treinamento para o algoritmo A_j e a configuração de treinamento C_k .

$$O(P_i, C_k, A_j) = \frac{1}{10} \times \sum_{r=1}^{10} MoJoFM(P_i, C_k, A_j, r) \quad (4.1)$$

Neste estudo, o domínio do conjunto de projeto $P = \{P_i\}$ será formado por todos os 78 projetos da categoria “library” da Apache Software Foundation. O domínio do conjunto de configurações $C = \{C_k\}$ será formado pelos oito tipos de configuração analisados na seção anterior: PAR-KEEP-UNIQ, PAR-KEEP-REPT, PAR-REMV-UNIQ, PAR-REMV-REPT, DOC-KEEP-UNIQ, DOC-KEEP-REPT, DOC-REMV-UNIQ e DOC-REMV-REPT. O domínio do conjunto de algoritmos $A = \{A_j\}$ será formado pelos três algoritmos analisados na seção anterior: Word2Vec, Doc2Vec e FastText. Por fim, os valores de $MoJoFM(P_i, C_k, A_j, r)$ são calculados submetendo cada projeto $P_i \in P$ ao otimizador configurado com o modelo de incorporação de palavras gerado na r -ésima rodada de treinamento baseada no algoritmo A_j e configuração C_k . Estes modelos de

incorporação de palavras foram reaproveitados dos 24 ciclos de treinamento mencionados na seção anterior.

Dizemos que uma configuração $C_{k_1} \in C$ é dominante sobre uma configuração $C_{k_2} \in C$, $C_{k_1} \neq C_{k_2}$, em um projeto $P_i \in P$ se e somente se as relações representadas nas equações 4.2 e 4.3 foram válidas. Esta dominância será representada como $C_{k_1} \sqsupset C_{k_2}$. A ausência de dominância será representada como $C_{k_1} \not\sqsupset C_{k_2}$. A dominância indica que uma configuração C_{k_1} superou os resultados da configuração C_{k_2} em pelo menos um algoritmo e empatou ou superou nos demais, representando assim uma configuração superior à outra para a otimização do projeto sob análise.

$$O(P_i, C_{k_1}, a) \geq O(P_i, C_{k_2}, a), \forall a \in A \quad (4.2)$$

$$\exists a \in A : O(P_i, C_{k_1}, a) > O(P_i, C_{k_2}, a) \quad (4.3)$$

A Tabela 4.7 mostra um exemplo dos dados utilizados para a análise de dominância. Ela apresenta a comparação entre as configurações PAR-KEEP-REPT e PAR-KEEP-UNIQ em três projetos. No cruzamento entre cada configuração, algoritmo e projeto é apresentada a média do *MoJoFM* resultante da otimização da distribuição de classes em pacotes nas dez rodadas de treinamento com a configuração, algoritmo e projeto em questão. Por exemplo, para o projeto groovy, todos os algoritmos na configuração PAR-KEEP-REPT produziram médias de *MoJoFM* superiores quando comparadas aos seus respectivos algoritmos na configuração PAR-KEEP-UNIQ (valores em negrito). Neste caso, podemos dizer que a configuração PAR-KEEP-REPT domina a configuração PAR-KEEP-UNIQ no projeto groovy. Esta relação se inverte no projeto anakia, onde todos os resultados da configuração PAR-KEEP-UNIQ são superiores aos respectivos algoritmos na configuração PAR-KEEP-REPT, indicando que PAR-KEEP-UNIQ domina PAR-KEEP-REPT neste projeto. Já para o projeto arrow (última linha), apenas um algoritmo na configuração PAR-KEEP-REPT produziu média superior a respectiva na comparação com a configuração PAR-KEEP-UNIQ. Neste caso, podemos dizer que as configurações não apresentam dominância entre si no projeto arrow.

Tabela 4.7: Relação de dominância entre as configurações PAR-KEEP-REPT e PAR-KEEP-UNIQ em três projetos.

Projeto	PAR-KEEP-REPT			PAR-KEEP-UNIQ		
	Word2Vec	Doc2Vec	FastText	Word2Vec	Doc2Vec	FastText
anakia	50,00	50,00	50,00	60,00	53,30	60,00
groovy	43,78	41,88	43,79	41,68	39,18	41,80
arrow	33,06	34,69	32,72	34,58	34,21	35,59

A Tabela 4.8 mostra os resultados da análise de dominância entre os pares de configurações de treinamento. A tabela contém duas colunas principais, PAR e DOC, que representam as diferentes unidades de treinamento das configurações. Os pares de configuração são formados dentro de uma mesma unidade de treinamento, perfazendo doze pares por unidade. Estes pares são apresentados nas colunas *Configuração #1* e *Configuração #2*. Os resultados são apresentados nas colunas “D” e “%”, que dizem respeito ao número de projetos analisados (os 78 projetos da categoria “library”). Assim, vemos que a configuração PAR-KEEP-REPT dominou a configuração PAR-KEEP-UNIQ em 10 dos 78 projetos (ou 12,82% dos projetos).

Podemos notar que, tanto para configurações baseadas na unidade de treinamento PAR quanto na unidade DOC, os maiores valores foram obtidos usando a configuração REMV-UNIQ, que domina as demais configurações dentro da mesma unidade um maior número de vezes que as demais configurações lhe dominam. Este resultado confirma o obtido na seção anterior, que já havia apontado as configurações que removem palavras repetidas como superiores às que mantêm as repetições.

Também podemos observar que a configuração DOC-REMV-UNIQ supera a configuração PAR-REMV-UNIQ em duas comparações (KEEP-REPT e REMV-REPT), perdendo apenas na configuração KEEP-UNIQ. Além disso, o número total de vitórias de DOC-REMV-UNIQ (143) supera o número de vitórias de PAR-REMV-UNIQ (118) em mais de 20%. Assim, considerando os resultados observados, podemos concluir que a melhor configuração de treinamento de modelos de incorporação de palavras para a distribuição de classes em pacotes é a DOC-REMV-UNIQ.

Tabela 4.8: Resultado para o número de dominâncias dos pares de parágrafo e documento (D = número de dominâncias (min = 0, máx = 78); % = percentual de dominância, entre 0 e 100%).

		PAR		DOC	
Configuração #1	Configuração #2	D	%	D	%
KEEP-REPT	KEEP-UNIQ	10	12,82	4	5,13
	REMV-REPT	6	7,69	7	8,97
	REMV-UNIQ	7	8,97	4	5,13
KEEP-UNIQ	KEEP-REPT	35	44,87	49	62,82
	REMV-REPT	17	21,79	34	43,59
	REMV-UNIQ	1	1,28	6	7,69
REMV-REPT	KEEP-REPT	45	57,69	44	56,41
	KEEP-UNIQ	17	21,79	13	16,67
	REMV-UNIQ	10	12,82	11	14,10
REMV-UNIQ	KEEP-REPT	40	51,28	58	74,36
	KEEP-UNIQ	48	61,54	35	44,87
	REMV-REPT	30	38,46	50	64,10

4.4.3 Análise dos algoritmos

Selecionada a configuração DOC-REMV-UNIQ, investigamos o algoritmo que produz os melhores resultados para esta configuração. Neste estudo, para cada projeto identificamos o algoritmo que gerou o maior valor de *MoJoFM* em cada rodada (r_0 a r_9) usando a configuração DOC-REMV-UNIQ. Em caso de empate entre dois ou mais algoritmos, todos os empatados são considerados vencedores. O resultado final é o número de vitórias de cada algoritmo. Como temos 78 projetos e 10 rodadas, um algoritmo pode atingir o máximo de 780 vitórias.

Tabela 4.9: Número de vitórias para cada algoritmo (*Word2Vec*, *Doc2Vec* e *FastText*) para cada projeto. A coluna “Palavras” indica o número de palavras únicas no projeto.

Software	Classes	Palavras	Vitórias		
			Word2Vec	Doc2Vec	FastText
org.apache.yetus	11	434	10	10	2
org.apache.anakia	12	689	10	10	10
org.apache.commons.daemon	12	752	0	10	0
org.apache.dvsl	16	744	10	10	10

Table 4.9 continued from previous page

org.apache.daffodil	22	722	3	6	8
org.apache.commons.csv	46	1.399	0	10	0
org.apache.commons.mail	46	1.879	2	6	5
org.apache.commons.exec	49	1.376	5	5	2
org.apache.commons.cli	51	1.301	1	8	1
org.apache.commons.fileupload	53	1.313	9	1	6
org.apache.pylucene	54	45.818	7	0	5
org.apache.bsf	56	2.869	0	9	2
org.apache.commons.jci	66	3.163	4	5	5
org.apache.commons.logging	66	5.306	9	2	2
org.apache.commons.proxy	80	695	0	10	0
org.apache.commons.dbutils	89	1.044	4	3	4
org.apache.commons.crypto	92	1.426	5	2	6
org.apache.commons.pool2	94	2.017	5	4	3
org.apache.commons.chain	97	1.124	2	5	5
org.apache.commons.weaver	105	1.182	6	0	6
org.apache.commons.statistics	123	1.903	7	7	8
org.apache.commons.dbcp2	128	2.046	6	1	3
org.apache.commons.codec	143	9.832	4	3	5
org.apache.commons.scxml	144	1.885	1	4	6
org.apache.commons.validator	150	4.680	2	5	6
org.apache.commons.rdf	153	1.596	7	2	3
org.apache.commons.numbers	176	3.272	5	4	5
org.apache.commons.jexl3	177	2.932	5	2	5
org.apache.commons.text	186	3.148	7	1	2
org.apache.commons.jxpath	213	3.127	2	0	9
org.apache.groovy	243	7.685	0	5	5
org.apache.log4j	248	7.388	3	4	3
org.apache.commons.beanutils	253	1.671	7	3	0
org.apache.commons.net	256	4.154	3	3	4
org.apache.thrift	278	2.682	3	5	2
org.apache.commons.digester3	297	2.488	2	4	4
org.apache.bval	313	9.914	6	2	2
org.apache.directory.fortress	325	3.359	4	4	3
org.apache.commons.io	364	3.866	4	3	3
org.apache.commons.geometry	382	1.377	3	0	7
org.apache.commons.lang3	384	2.028	7	1	3
org.apache.orc	386	3.530	1	7	2
org.apache.commons.rng	395	5.586	8	1	6
org.apache.commons.configuration2	444	3.605	1	0	9
org.apache.commons.jcs3	455	843	5	1	5
org.apache.xml.security	489	4.158	2	4	5
org.apache.commons.vfs2	503	2.904	6	0	4
org.apache.commons.jelly	519	3.455	8	0	2
org.apache.commons.collections4	524	3.149	6	2	3
org.apache.commons.imaging	540	2.857	3	0	7
org.apache.commons.compress	554	5.223	6	0	4
org.apache.bcel	565	3.620	2	3	6
org.apache.avro	626	4.109	4	1	5
org.apache.shiro	672	4.362	6	1	3

Table 4.9 continued from previous page

org.apache.hc.client5	682	5.713	5	4	1
org.apache.pivot	845	3.449	9	1	0
org.apache.hc.core5	896	3.176	2	2	6
opennlp	990	6.004	8	2	0
org.apache.arrow	1.014	4.425	2	3	5
org.apache.kerby	1.043	4.003	3	0	7
org.apache.commons.math4	1.091	6.800	7	0	3
org.apache.pdfbox	1.108	8.182	10	0	0
org.apache.olingo	1.114	3.398	2	8	0
org.apache.plc4x	1.176	4.955	6	0	4
org.apache.mahout	1.232	6.825	9	0	1
org.apache.ctakes	1.380	7.481	4	0	6
org.apache.tika	1.626	10.828	8	0	2
org.apache.hudi	1.976	5.982	6	1	3
org.apache.axiom	2.060	4.521	3	5	2
org.apache.sis	2.695	10.552	4	1	5
org.apache.jackrabbit	3.042	8.061	0	10	0
org.apache.poi	3.645	19.156	7	2	1
org.apache.openjpa	4.610	10.493	6	1	3
org.apache.lucene	4.937	45.877	10	0	0
org.jclouds	5.859	12.389	2	8	0
org.apache.jena	6.071	13.559	6	0	4
org.apache.cxf	6.845	14.051	5	5	1
org.apache.cayenne	8.380	2.431	5	3	2
Número de vitórias por algoritmo			367	255	287

A Tabela 4.9 apresenta o número de vitórias de cada algoritmo em cada projeto. Verificamos que o algoritmo *Word2Vec* obteve o maior número de vitórias (367 vitórias). É importante observar que o algoritmo *FastText* não foi bem-sucedido na validação cruzada, quando consideramos todas as configurações. No entanto, ao restringir as análises para a configuração DOC-REMV-UNIQ e considerar o número de vitórias e empates projeto a projeto (ao invés da média de *MoJoFM* entre os projetos), o resultado se altera e o algoritmo *FastText* supera o *Doc2Vec* (287 versus 255 vitórias).

Na Seção 4.4.2, verificamos que o *MoJoFM* diminui com o aumento do número de classes. Para realizar uma análise sob o ponto de vista dos algoritmos, dividimos os 78 projetos em quatro grupos com 20 projetos (sendo que o último ficou com apenas 18 projetos) em ordem crescente do número de classes. Neste experimento, podemos observar que o algoritmo *Doc2Vec* tem o maior número de vitórias (40,0%) para uma seleção entre 11 e 105 classes (grupo 1). Para um intervalo entre 123 e 382 classes (grupo 2), o algoritmo *FastText* que assume a ponta com 37,28% das vitórias. Para projetos com

mais de 384 classes (grupos 3 e 4), notamos que *Word2Vec* tem a melhor eficácia (44,76% das vitórias para o grupo 3 e 55,24% para o grupo 4).

Assim, concluímos que o algoritmo *Word2Vec* produz modelos de incorporação de palavras mais adequados ao problema de modularização de software, especialmente para projetos de larga escala. Em função disso, os modelos produzidos por este algoritmo serão utilizados nas próximas análises.

4.4.4 Comparação com o modelo Wiki-en

Com base nos últimos resultados, percebemos que o melhor modelo foi treinado na configuração DOC-REMV-UNIQ com o algoritmo *Word2Vec*. Precisamos agora avaliar se os resultados para este algoritmo são melhores do que os resultados obtidos a partir de um modelo pré-treinado baseado em um *corpus* mais amplo, porém mais genérico. No caso, selecionamos o modelo *Wiki-en*, que foi identificado como o melhor modelo pelas análises realizadas no Capítulo 3.

Como resultado desta comparação, percebemos que o modelo de incorporação de palavras *Wiki-en* obteve a melhor eficácia na configuração DOC-REMV-UNIQ e modalidade M4: o *MoJoFM* foi $40,82 \pm 4,65$. A variação nos resultados do modelo decorre da seleção de diferentes projetos na análise de cada rodada da validação cruzada. Para efeito de comparação, os algoritmos *Word2Vec*, *Doc2Vec* e *FastText* obtiveram, respectivamente, valores de *MoJoFM* iguais a $35,31 \pm 3,13$, $34,73 \pm 2,71$ e $34,60 \pm 2,77$. A Figura 4.5 apresenta um *boxplot* com os resultados de *MoJoFM* gerados por cada algoritmo e pelo modelo *Wiki-en*. Aplicando um teste de Wilcoxon nas séries de dados dos algoritmos em comparação com a série de dados do *Wiki-en* observamos que a mediana do *MoJoFM* obtido pelo *Wiki-en* é significativamente superior aos *MoJoFM* obtidos pelos demais algoritmos.

A Tabela 4.10 apresenta o número de vitórias dos modelos de incorporação de palavras gerados pelos algoritmos *Word2Vec*, *Doc2Vec* e *FastText* em cada rodada de treinamento e do modelo de incorporação de palavras *Wiki-en*. O modelo *Wiki-en* obteve 373 vitórias, seguido dos modelos *Word2Vec* (205 vitórias), *Doc2Vec* (159 vitórias) e *FastText* (142 vitórias). As variações no número de vitórias de cada algoritmo em diferentes projetos refletem as diferentes representações vetoriais no espaço multidimensional computadas para as palavras componentes do modelo. Por exemplo, o

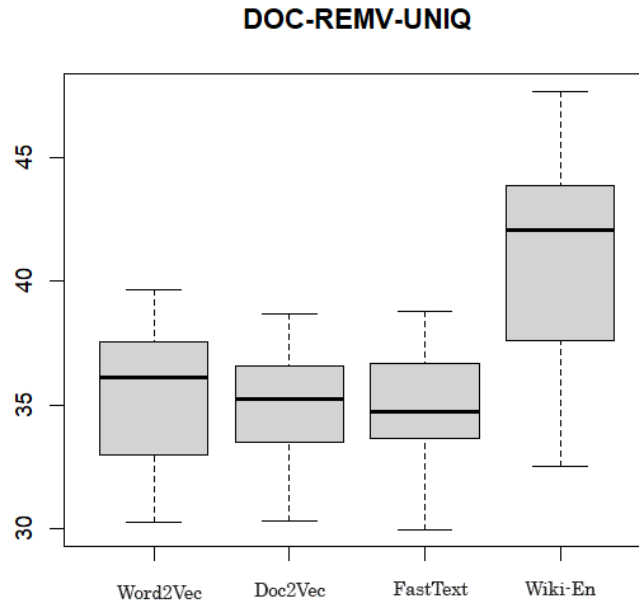


Figura 4.5: Um *boxplot* dos valores de MoJoFM produzidos pelos três algoritmos e pelo modelo *Wiki-en* para a configuração DOC-REMV-UNIQ.

projeto *org.apache.anakia* tem zero vitórias para os algoritmos e 10 vitórias para o modelo *Wiki-en*, enquanto o projeto *org.apache.dvsl* tem zero vitórias para o *Wiki-en* e empate com 10 vitórias para os demais algoritmos.

Para ilustrar as diferenças entre os modelos, a Figura 4.6 mostra as palavras mais próximas a palavra “file” obtidas com o modelo *Word2Vec* na configuração DOC-REMV-UNIQ (a esquerda) e com o modelo *Wiki-en* (a direita). A Figura 4.7 apresenta um diagrama similar, focalizando a palavra “name”. Os vetores de 300 dimensões de cada palavra são apresentados em um plano cartesiano bidimensional utilizando uma projeção t-SNE. *t-Distributed Stochastic Neighbor Embedding* (t-SNE) é uma técnica para redução de dimensionalidade adequada para a visualização de conjuntos de dados de alta dimensão. A redução não linear da dimensionalidade significa que o algoritmo permite separar dados que não podem ser separados por uma linha reta. Esta técnica é usada com frequência para visualizar conjuntos de dados complexos em duas e três dimensões, permitindo entender melhor os padrões e as relações subjacentes aos dados [69]. Podemos perceber que as palavras mais próximas têm contextos bastante diferentes

em relação a palavra pivô de cada gráfico. Isso mostra que os algoritmos geram, de fato, resultados muito diferentes para representar cada palavra.

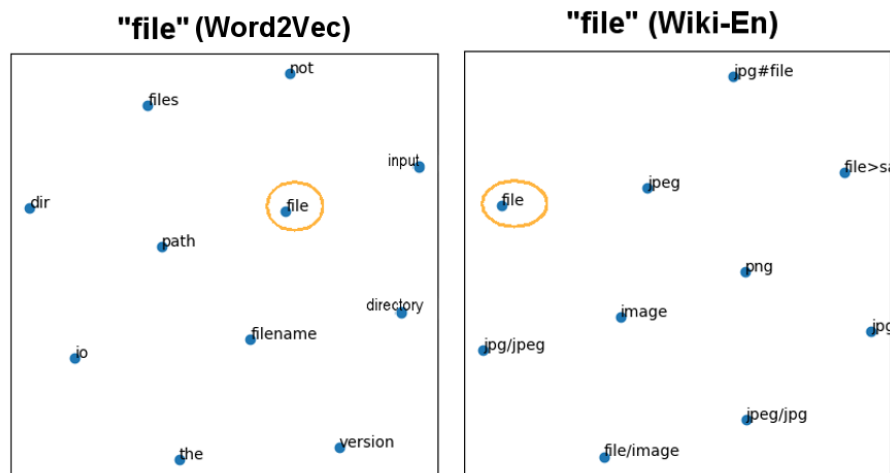


Figura 4.6: Projeção t-SNE das proximidades da palavra *file* para um modelo de incorporação de palavras gerado pelo algoritmo *Word2Vec* e o modelo *Wiki-en*.

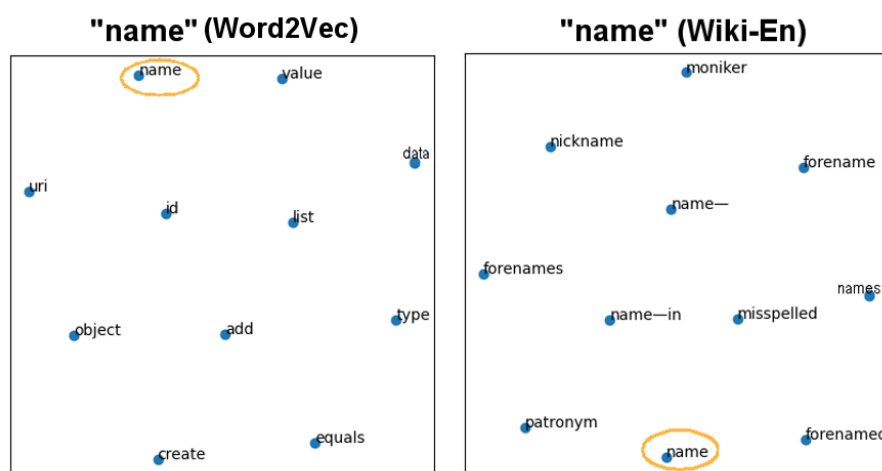


Figura 4.7: Projeção t-SNE das proximidades da palavra *name* para um modelo de incorporação de palavras gerado pelo algoritmo *Word2Vec* e o modelo *Wiki-en*.

Repetindo o experimento com os quatro grupos de projetos definidos na Seção 4.4.3, porém incluindo o modelo *Wiki-en* na contagem de vitórias, observamos que o modelo *Wiki-en* ganha em todos os grupos com 38,93% das vitórias para uma seleção de

projetos entre 11 e 105 classes (grupo 1), 34,35% para o intervalo entre 123 e 382 classes (grupo 2), 40,08% (grupo 3) e 29,01% (grupo 4) para projetos com mais de 384 classes.

Tabela 4.10: Número de vitórias dos modelos de incorporação de palavras gerados por cada algoritmo (*Word2Vec*, *Doc2Vec* e *FastText*) para cada projeto e do modelo de incorporação de palavras *Wiki-en*. O número de palavras de cada projeto indica o número de palavras na representação do seu documento, sem repetição. O número de palavras do *Wiki-en* indica o número de palavras do vocabulário do projeto que também são encontradas no modelo *Wiki-en*.

Software	Classes	Palavras	Palavras Wiki-en	Vitórias			
				Word2Vec	Doc2Vec	FastText	Wiki-en
org.apache.yetus	11	434	428	10	10	2	0
org.apache.anakia	12	689	680	0	0	0	10
org.apache.commons.daemon	12	752	733	0	10	0	10
org.apache.dvsl	16	744	706	10	10	10	0
org.apache.daffodil	22	722	706	0	0	0	10
org.apache.commons.csv	46	1.399	1.363	0	9	0	3
org.apache.commons.mail	46	1.879	1.829	2	6	5	0
org.apache.commons.exec	49	1.376	1.335	0	0	0	10
org.apache.commons.cli	51	1.301	1.225	0	0	0	10
org.apache.commons.fileupload	53	1.313	1.286	9	1	6	0
org.apache.py lucene	54	45.818	39.590	7	0	5	0
org.apache.bsf	56	2.869	2.820	0	9	2	0
org.apache.commons.jci	66	3.163	3.070	1	2	0	7
org.apache.commons.logging	66	5.306	4.873	8	1	1	3
org.apache.commons.proxy	80	695	688	0	10	0	0
org.apache.commons.dbutils	89	1.044	1.032	2	3	2	5
org.apache.commons.crypto	92	1.426	1.383	0	0	0	10
org.apache.commons.pool2	94	2.017	1.987	0	0	0	10
org.apache.commons.chain	97	1.124	1.098	0	3	0	7
org.apache.commons.weaver	105	1.182	1.147	1	0	3	7
org.apache.commons.statistics	123	1.903	1.846	7	7	8	0
org.apache.commons.dbcp2	128	2.046	1.983	0	0	0	10
org.apache.commons.codec	143	9.832	7.929	1	0	2	8
org.apache.commons.scxml	144	1.885	1.834	0	0	0	10
org.apache.commons.validator	150	4.680	4.381	0	2	0	8
org.apache.commons.rdf	153	1.596	1.546	5	2	3	3
org.apache.commons.numbers	176	3.272	3.160	4	4	5	2
org.apache.commons.jexl3	177	2.932	2.808	1	0	4	5
org.apache.commons.text	186	3.148	2.959	7	1	2	0
org.apache.commons.xpath	213	3.127	2.855	0	0	0	10
org.apache.groovy	243	7.685	6.991	0	5	5	0
org.apache.log4j	248	7.388	6.643	3	4	3	0
org.apache.commons.beanutils	253	1.671	1.588	6	2	0	3
org.apache.commons.net	256	4.154	3.807	3	2	4	2
org.apache.thrift	278	2.682	2.566	3	4	1	3

Table 4.10 continued from previous page

org.apache.commons.digester3	297	2.488	2.407	2	4	4	0
org.apache.bval	313	9.914	9.141	4	1	0	6
org.apache.directory.fortress	325	3.359	2.843	4	4	3	0
org.apache.commons.io	364	3.866	3.660	0	0	0	10
org.apache.commons.geometry	382	1.377	1.343	0	0	0	10
org.apache.commons.lang3	384	2.028	1.927	7	1	3	0
org.apache.orc	386	3.530	3.429	0	5	0	6
org.apache.commons.rng	395	5.586	4.549	8	1	6	0
org.apache.commons.configuration2	444	3.605	3.427	0	0	2	9
org.apache.commons.jcs3	455	843	818	5	1	5	0
org.apache.xml.security	489	4.158	3.640	2	4	5	0
org.apache.commons.vfs2	503	2.904	2.755	0	0	0	10
org.apache.commons.jelly	519	3.455	3.202	2	0	1	7
org.apache.commons.collections4	524	3.149	3.047	0	0	0	10
org.apache.commons.imaging	540	2.857	2.812	0	0	0	10
org.apache.commons.compress	554	5.223	4.671	6	0	4	0
org.apache.bcel	565	3.620	3.279	0	2	3	5
org.apache.avro	626	4.109	3.841	4	1	5	0
org.apache.shiro	672	4.362	4.206	0	0	0	10
org.apache.hc.client5	682	5.713	4.913	5	3	1	1
org.apache.pivot	845	3.449	3.403	0	0	0	10
org.apache.hc.core5	896	3.176	2.994	1	1	3	5
opennlp	990	6.004	5.293	0	0	0	10
org.apache.arrow	1.014	4.425	4.159	0	0	0	10
org.apache.kerby	1.043	4.003	3.548	3	0	5	2
org.apache.commons.math4	1.091	6.800	6.213	6	0	3	1
org.apache.pdfbox	1.108	8.182	7.095	7	0	0	3
org.apache.olingo	1.114	3.398	3.075	0	0	0	10
org.apache.plc4x	1.176	4.955	4.572	0	0	0	10
org.apache.mahout	1.232	6.825	6.300	8	0	0	2
org.apache.ctakes	1.380	7.481	6.662	4	0	5	1
org.apache.tika	1.626	10.828	9.379	0	0	0	10
org.apache.hudi	1.976	5.982	5.550	3	1	1	5
org.apache.axiom	2.060	4.521	4.038	0	0	0	10
org.apache.sis	2.695	10.552	9.930	4	1	5	0
org.apache.jackrabbit	3.042	8.061	6.810	0	7	0	3
org.apache.poi	3.645	19.156	14.204	7	2	1	0
org.apache.openjpa	4.610	10.493	8.520	6	1	3	0
org.apache.lucene	4.937	45.877	39.633	9	0	0	1
org.jclouds	5.859	12.389	9.459	2	6	0	2
org.apache.jena	6.071	13.559	11.613	0	0	0	10
org.apache.cxf	6.845	14.051	10.775	2	3	0	6
org.apache.cayenne	8.380	2.431	2.342	4	3	1	2
Número de vitórias por algoritmo				205	159	142	373

Em resumo, podemos afirmar que o algoritmo *Wiki-en* manteve a melhor eficácia, seguido do algoritmo *Word2Vec*. Este resultado indica que o treinamento de modelos de incorporação de palavras com base no conjunto de projetos da categoria *Library* da

Apache Software Foundation não supera o treinamento de um modelo de incorporação de palavras com base em um grande volume de textos gerais, ao menos no sentido de orientar a distribuição de classes em pacotes.

4.4.5 Comparação com um modelo reduzido

Para verificar se a redução do número de projetos afeta a eficácia do treinamento, repetimos a validação cruzada com metade dos projetos utilizados no teste anterior, selecionando apenas os projetos com maior número de palavras no vocabulário. A intenção é verificar se a redução do número de projetos envolvidos no treinamento leva uma redução na qualidade dos resultados do algoritmo *Word2Vec*, ainda que usando projetos com vocabulário extenso. A redução na qualidade, se observada, levaria a crer que uma análise inversa (ou seja, aumentando o número de projetos envolvidos no treinamento) poderia trazer resultados melhores para a qualidade dos algoritmos.

Sorteamos dez pares de conjuntos de treinamento e teste na proporção de 80% e 20%, respectivamente. Cada conjunto de treinamento totalizou 31 projetos e cada conjunto de teste incluiu 8 projetos. Os dez pares de conjuntos foram utilizados nas dez rodadas de validação cruzada com dez grupos. O resultado final mostra o valor de MoJoFM de $34,65 \pm 3,50$ para a configuração DOC-REMV-UNIQ com o algoritmo *Word2Vec*. Este resultado mostra um valor menor do que o obtido pelo mesmo algoritmo na Tabela 4.6, embora a diferença entre estes valores não seja estatisticamente significativa (segundo um teste de Wilcoxon com $\alpha = 0.05$).

Assim, concluímos que a redução do *corpus* utilizado no treinamento do modelo de incorporação de palavras pelo algoritmo *Word2Vec* não afetou de forma relevante os resultados obtidos em termos de MoJoFM com as distribuições propostas pelos desenvolvedores.

4.5 Um modelo de incorporação de palavras de larga escala

Por mais que se treine modelos de incorporação de palavras com as oito configurações apresentadas na seção anterior e usando os dados da *Apache Software Foundation*, os resultados da distribuição de classes em pacotes orientada por estes modelos não se mostraram melhores do que os resultados obtidos com o treinamento em larga escala com informações vindas de notícias em geral (modelo *Wiki-en*).

Em função destas observações, treinamos um modelo de incorporação de palavras mais amplo, baseado não apenas nas palavras encontradas no código-fonte dos sistemas da *Apache Software Foundation*, mas também em sistemas vindos de outras empresas. Para este propósito, criamos um novo modelo, chamado *Organizações*, baseado em um maior número de projetos implementados na linguagem *Java*⁶, a partir dos repositórios GitHub das organizações apresentadas na Tabela 4.11.

Estes repositórios foram identificados através de uma consulta ao GitHub por projetos escritos em Java, selecionando-se de forma *ad hoc* organizações de desenvolvimento de software conhecidas. Selecionados os repositórios, baixamos seu código-fonte e removemos arquivos de código escritos em linguagens de programação diferentes de *Java*, assim como diretórios e arquivos contendo as palavras “test”, “example” e “demo” em seus nomes. Este procedimento tem como objetivo descartar códigos repetidos e palavras não relacionadas com os projetos. Em seguida, descartamos organizações com menos de dez projetos de modo que apenas grandes organizações, com projetos distribuídos por canais de larga escala, fizessem parte do modelo de treinamento. Ao final deste procedimento, restaram 2.723 projetos compostos por 1.864.580 arquivos de código-fonte.

A Tabela 4.11 mostra a distribuição dos projetos utilizados no treinamento do novo modelo entre as empresas desenvolvedoras de software selecionadas. O modelo foi treinado usando a configuração DOC-REMV-UNIQ na modalidade M4 e o algoritmo *Word2Vec* com vetores de 300 dimensões. Realizamos uma única rodada de treinamento que levou cerca de 75 minutos. A Tabela 4.12 mostra os resultados da otimização para o modelo *Organizações*, comparados com os melhores resultados produzidos pelos algoritmos de treinamento *Word2Vec*, *Doc2Vec*, *FastText* e *Wiki-en* na configuração DOC-REMV-UNIQ. Podemos perceber que o modelo *Wiki-en* se mantém com melhor eficácia. Também podemos perceber que o modelo *Organizações* ficou ligeiramente abaixo do melhor algoritmo nos experimentos com a configuração DOC-REMV-UNIQ e considerando os 183 projetos selecionados da *Apache Software Foundation*.

⁶ Projetos escritos em Java foram selecionados para que reutilizássemos as ferramentas desenvolvidas para realizar as análises relatadas nas seções anteriores.

Tabela 4.11: Número e tamanho dos projetos das organizações selecionadas para a criação do primeiro modelo, totalizando 1.864.580 arquivos Java.

Repositório	Projetos	Classes	Repositório	Projetos	Classes
Ericsson	12	2.169	ElasticNV	36	65.721
MongoDB	12	3.208	Salesforce	37	4.397
UnitySoftware	12	16.948	Oracle	38	58.401
Dropbox	12	23.324	Amazon	42	9.241
AllegroTech	16	1.815	Knime	42	199.72
Lyft	16	33.225	Jfrog	51	9.573
Vmware	18	6.536	Confluent	55	7.006
NASA	18	16.054	Netflix	56	11.414
Splunk	18	18.978	Palantir	59	6.881
Squarespace	19	5.044	Atlassian	65	37.139
Xiaomi	19	13.217	Adobe	69	8.324
Tencent	19	136.565	Alibaba	88	52.939
Uber	20	2.431	HubSpot	92	21.408
Intuit	24	4.837	Microsoft	99	222.432
Spotify	26	3.285	Google	112	38.986
eBay	28	2.982	EPAMSystems	115	22.431
ExpediaGroup	33	5.067	Eclipse	119	107.881
Airbnb	33	10.748	IBM	203	28.109
Nokia	35	36.661	Apache	919	728.802
Sap	36	60.429			

Tabela 4.12: Média dos valores *MoJoFM* para a seleção dos 78 projetos da categoria “library” e os 183 projetos da *Apache Software Foundation* selecionados, calculados usando os diversos modelos estudados.

Modelo	Vocabulário tokens (K)	Tamanho (MB)	MoJoFM (78)	MoJoFM (183)
DOC-REMOV-UNIQ/FastText	24,58	29,68	37,11	38,12
DOC-REMOV-UNIQ/Doc2Vec	=	=	37,39	35,62
DOC-REMOV-UNIQ/Word2Vec	=	=	37,66	36,96
Wiki-en	2.519,37	2.263,42	40,33	40,00
Organizações	103,85	125,46	37,73	36,79

4.6 Um modelo de incorporação de palavras gerado por ajuste fino

Utilizando uma rede neural convolucional (CNN), Orozco [90] comparou o algoritmo *Word2Vec* com o algoritmo Máquina de Vetores-Suporte (SVM), que usa diferentes representações de texto. A pesquisa utilizou o algoritmo *Word2Vec* em CNNs, comparando a eficácia da CNN sem *Word2Vec* para tarefas de classificação binária em conjuntos de dados pequenos e desbalanceados. O objetivo do autor era utilizar transcrições e processamento de linguagem natural, juntamente com ML e CNNs, para descobrir o que é conteúdo conspiratório e o que não é no YouTube. Embora a rede neural com *Word2Vec* não tenha superado o SVM com TF-IDF, foi demonstrado que ela pode ajudar a prever melhor, mesmo com um conjunto de dados limitado e desbalanceado, quando comparada à CNN sem *Word2Vec*. Estes resultados confirmam que os modelos de incorporação de palavras ajudam a executar tarefas de classificação.

Anikin et al. [4] trabalharam com métodos de construção de dicionários *online* baseando-se no treinamento adicional de um modelo de incorporação de palavras já existente com o *corpus* de textos de um domínio específico que o usuário precisa para realizar uma determinada tarefa. Para este trabalho, os autores discutiram o uso do *Word2Vec* com a atualização do vocabulário *online* e extensão do *corpus* de dados original com treinamento adicional para a extração de conceitos de domínio visando automatizar a criação de ontologias de domínios. Os autores mostraram vantagens e desvantagens na utilização de métodos de treinamento adicional de modelos de incorporação pré-treinados (como o *Word2Vec*).

Com base nos bons resultados dos trabalhos de Anikin et al. [4] e Orozco [90], resolvemos testar uma abordagem que mescla a matriz de pesos de um modelo de incorporação de palavras pré-treinado com um modelo criado a partir de um conjunto de projetos de software, cruzando o vocabulário destes modelos. Neste procedimento, nenhuma palavra é adicionada ao vocabulário do modelo de incorporação de palavras pré-treinado, mas as palavras encontradas nos dois modelos adotam os pesos do modelo pré-treinado, enquanto as palavras que não se cruzam são ignoradas. Por exemplo, considerando o modelo de incorporação de palavras pré-treinado *Wiki-en* e o modelo *Organizações*, foi formado um novo modelo inicial composto das 103.850 palavras do modelo *Organizações*, sendo que as palavras encontradas no modelo *Wiki-en* receberam os pesos do modelo *Wiki-en* e as demais palavras (não encontradas no *Wiki-en*) mantiveram os pesos do modelo *Organizações*.

A intenção ao construir este modelo inicial é trazer do modelo pré-treinado apenas os pesos das palavras existentes no modelo construído a partir dos projetos de software, mantendo os pesos originais quando as palavras não forem encontradas no modelo pré-treinado. Estes pesos são utilizados para inicializar a rede neural e executar o processo de treinamento. O modelo resultante deste processo representa um ajuste-fino do modelo de incorporação de palavras pré-treinado com informações trazidas do novo modelo específico para os projetos de software. A taxa de substituição (percentual de palavras do modelo específico encontradas no modelo de incorporação pré-treinado) ficou em torno de 64% no caso dos modelos *Organizações* e *Wiki-en*, usando o algoritmo *Word2Vec* com vetores de 300 dimensões.

A Tabela 4.13 mostra os resultados da otimização com os modelos estudados (78 projetos da categoria “library” e os 183 projetos selecionados da *Apache Software Foundation*), onde *Organizações* representa o modelo treinado a partir dos projetos das organizações citadas na Tabela 4.11 e *Organizações-i-WikiEn* representa a interseção do modelo *Organizações* com o modelo pré-treinado *Wiki-en*. A coluna *Vocabulário* indica o número de palavras no modelo e a coluna *Tamanho* indica o tamanho físico do modelo alocado em disco. Os modelos modificados apresentaram resultados melhores em termos de *MoJoFM* quando comparados com as distribuições de classes em pacotes utilizadas pelos desenvolvedores dos softwares. A melhor eficácia observada foi do modelo

Tabela 4.13: Média dos valores *MoJoFM* para a seleção dos 78 projetos da categoria “library” e os 183 projetos selecionados calculados usando os diversos modelos estudados, incluindo o modelo de ajuste-fino.

Modelo	Vocabulário tokens (K)	Tamanho (MB)	MojoFM (78)	MojoFM (183)
DOC-REMOV-UNIQ/FastText	24,58	29,68	37,11	38,12
DOC-REMOV-UNIQ/Doc2Vec	=	=	37,39	35,62
DOC-REMOV-UNIQ/Word2Vec	=	=	37,66	36,96
Wiki-en	2.519,37	2.263,42	40,33	40,00
Organizações	103,85	125,46	37,73	36,79
Organizações-i-WikiEn	103,85	125,46	42,44	40,16

modificado de *Organizações* com a interseção de *Wiki-en* com 42,44 de média do *MoJoFM*, cerca de 12,5% maior que a média do modelo original.

4.7 Considerações finais

Este capítulo explorou a ideia de treinar modelos de incorporação de palavras específicos para um conjunto de projetos com base nos resultados do Capítulo 3, que apontou o algoritmo de clusterização mais adequado, os diferentes algoritmos de treinamento de modelos de incorporação de palavras e as regiões do código-fonte de onde os termos linguísticos devem ser extraídos. Realizamos diversos estudos experimentais para comparar modelos treinados com base em projetos da *Apache Software Foundation* entre si, considerando diversas configurações, e com o modelo pré-treinado *Wiki-en*.

Nesta comparação, percebemos que o modelo *Wiki-en* obteve a melhor eficácia na configuração DOC-REMOV-UNIQ e modalidade M4, seguido do algoritmo *Word2Vec*. Para tentar ultrapassar os resultados do modelo pré-treinado, foi criado um modelo de incorporação de palavras mais amplo, chamado *Organizações*, baseado em um maior número de projetos e também usando a configuração DOC-REMOV-UNIQ na modalidade M4 com o algoritmo *Word2Vec*. Ainda assim, o modelo de incorporação de palavras pré-treinado *Wiki-En* se manteve com melhor eficácia.

Com base nos bons resultados do modelo de incorporação de palavras pré-treinado *Wiki-en*, decidimos testar uma abordagem de ajuste fino, que mescla a matriz de pesos do modelo de incorporação de palavras pré-treinado *Wiki-en* com o modelo *Organizações*. O modelo resultante apresentou a melhor eficácia, com 42,44 de média do *MoJoFM*, quase 5% maior que a média do modelo original e 2% maior que o modelo *Wiki-en*.

Com base nestes estudos, é possível afirmar que usar modelos de incorporação de palavras oferece uma boa eficácia para reduzir as distâncias entre as distribuições de classes em pacotes geradas pelos algoritmos e a distribuição proposta pelos desenvolvedores com um modelo híbrido construído com projetos selecionados e o modelo de incorporação de palavras pré-treinado *Wiki-en*, utilizando a configuração DOC-REMV-UNIQ com o algoritmo *Word2Vec* na modalidade M4.

5 CONCLUSÕES

Resumo. O objetivo deste capítulo é apresentar as conclusões dos estudos realizados como parte desta pesquisa.

5.1 Introdução

Software Module Clustering (SMC) é uma técnica para reorganizar sistemas de software, agrupando classes e módulos em *clusters*, com o objetivo de aumentar a coesão e reduzir o acoplamento do sistema [30]. Esse processo de modularização é apoiado pela Engenharia de Software Baseada em Buscas, que usa algoritmos meta-heurísticos para encontrar soluções eficientes [12]. A maioria das abordagens de clusterização se baseia na estrutura do software (as dependências entre as classes), mas algumas abordagens utilizam técnicas de processamento de linguagem natural para analisar informações léxicas do código-fonte e identificar subsistemas.

Esta Tese tem como objetivo avaliar como os termos linguísticos podem ser utilizados pelos desenvolvedores com o propósito de orientar a distribuição de classes em pacotes, aplicando modelos de incorporação de palavras (*word embeddings*) para chegar a um agrupamento compatível com aquele que os desenvolvedores de sistemas amplamente utilizados propuseram. A intenção não é propor ao desenvolvedor uma solução diferente, mas estudar o espectro de soluções para verificar se, por meio do uso de termos linguísticos, é possível explicar a divisão que foi proposta pelo desenvolvedor.

5.2 Resultados encontrados

A análise comparou algoritmos estruturais e léxicos, considerando variações de nomes de classes e pacotes, em modalidades de agrupamento como “*single*” (nome da classe), “*complete*” (nome da classe e pacotes), e “*full*” (nome completo incluindo domínio). A modalidade completa (“f-DEV”) demonstrou proximidade com as distribuições de pacotes propostas, ainda que influenciada por nomes de pacotes. Em seguida, foram criadas diferentes configurações (M1 a M4), incorporando variados

elementos do código, como métodos e atributos, com a modalidade M4, mais completa, alcançando melhores resultados.

Estudos experimentais identificaram que o HAC (*Hierarchical Agglomerative Clustering*) com métrica euclidiana e o *Spectral Clustering* são os algoritmos de agrupamento mais eficazes para esta tarefa, principalmente na configuração DOC-REMV-UNIQ e usando o algoritmo de treinamento de modelos de incorporação de palavras *Word2Vec*. Experimentos analisando diferentes projetos mostraram que o algoritmo *Word2Vec* teve melhor eficácia em projetos grandes, enquanto *Doc2Vec* e *FastText* se destacaram em projetos pequenos.

Testes adicionais com modelos pré-treinados como *Wiki-en* indicaram melhorias na precisão dos agrupamentos ao serem comparados com modelos específicos criados para o estudo. Por fim, o estudo concluiu que a incorporação de palavras é uma técnica viável para aproximar a distribuição de classes em pacotes sugerida pelos desenvolvedores, evidenciando que modelos pré-treinados podem aumentar a eficácia de algoritmos de modularização.

O trabalho destaca as limitações introduzidas pela qualidade dos identificadores no código e sugere que bons identificadores melhoram a qualidade dos agrupamentos.

- **Um bom projeto deve garantir clareza e concisão:** os desenvolvedores devem utilizar uma linguagem simples, que todos possam entender. Eles devem explicar o que o código faz, como ele funciona e por que ele foi escrito dessa forma de maneira clara;
- **Codificação:** os desenvolvedores devem ter em mente diretrizes de formatação e estilo de código. Devem utilizar estilos de documentação diferentes, como comentários, documentação em linha e documentação externa. É essencial seguir as convenções definida pela organização, como comentários, nomenclatura e práticas de codificação. Os desenvolvedores devem ter atenção para dar nomes a variáveis, tipos, funções e outras entidades no código-fonte e na documentação. Devem evitar usar letras, abreviações pouco conhecidas, nomes engraçados ou pular sublinhados. Isso pode incluir o uso de guias de estilo populares, como o Google's Style Guide (<https://google.github.io/styleguide/>);

- Documentação: uma boa documentação deve incluir informações sobre como o software funciona, como usá-lo e como modificá-lo. Isso é benéfico para desenvolvedores que são novos no projeto ou precisam fazer alterações no código.

As contribuições desta pesquisa incluem algoritmos de agrupamento para distribuição de classes em pacotes baseada em informações léxicas, criação de um *corpus* específico para esta tarefa e a análise de diferentes estratégias para modularização de software.

As questões de pesquisa propostas no primeiro capítulo desta monografia podem ser respondidas conforme abaixo:

QP1: Uma abordagem de agrupamento baseada em termos léxicos pode levar a resultados melhores do que os resultados obtidos usando abordagens estruturais?

Sim. Os experimentos realizados mostraram que as abordagens léxicas, especificamente utilizando modelos de incorporação de palavras, geram distribuições de classes em pacotes mais próximas daquelas propostas pelos desenvolvedores quando comparadas com abordagens baseadas nas dependências entre as classes do projeto.

QP2: É possível melhorar os resultados das abordagens de agrupamento baseadas em termos léxicos construindo um corpus específico para o problema de agrupamento?

Os melhores resultados foram obtidos com a interseção dos modelos gerados a partir de um *corpus* específico com um modelo de incorporação de palavras pré-treinado com base em informações diversas.

Com base nestes estudos, é possível afirmar que usar modelos de incorporação de palavras oferece uma boa eficácia para reduzir as distâncias entre as distribuições de classes em pacotes geradas pelos algoritmos de agrupamento e as distribuições propostas pelos desenvolvedores com um modelo híbrido construído com projetos selecionados e o modelo de incorporação de palavras pré-treinado *Wiki-en*, utilizando a configuração DOC-REMV-UNIQ com o algoritmo *Word2Vec* na modalidade M4.

5.3 Trabalhos futuros

Embora este trabalho tenha abordado as questões de pesquisa propostas, ele também levantou diversas oportunidades para investigações futuras.

Em primeiro lugar, novos experimentos podem ser realizados considerando diferentes tipos de termos extraídos a partir da análise de *Part-of-Speech Tagging*. O Capítulo 3 apresentou resultados baseados exclusivamente em substantivos, mas outras classes gramaticais podem ser exploradas em experimentos futuros.

Além disso, os estudos conduzidos focaram principalmente em softwares da *Apache Software Foundation* e demonstraram que modelos de incorporação de palavras se beneficiam de um corpus extenso. Inicialmente, a pesquisa pretendia analisar diferentes grupos de software para avaliar se um modelo desenvolvido para uma categoria poderia ser aplicado com sucesso a outra. No entanto, a análise ficou restrita aos projetos da categoria “Library” da *Apache Software Foundation*. Dessa forma, a aplicação da abordagem proposta a diferentes categorias de software e a sistemas desenvolvidos em diversas linguagens de programação são temas promissores para futuras pesquisas.

Outro aspecto a ser aprimorado diz respeito à definição dos parâmetros utilizados na vetorização. Alguns foram escolhidos com base em experimentação, sem o suporte de uma heurística sistemática. Trabalhos futuros podem se beneficiar de um método mais rigoroso para a definição desses parâmetros. Além disso, os experimentos realizados adotaram um único valor para a dimensionalidade dos vetores. Recomenda-se investigar o impacto de diferentes valores para esse parâmetro nos resultados obtidos. Para uma avaliação mais aprofundada dos resultados, futuras pesquisas também podem propor e definir casos qualitativos que ajudem a interpretar os achados do estudo.

A Tabela 5.1 apresenta os dois melhores e os dois piores valores de *MoJoFM* para alguns dos projetos analisados. Observa-se que um maior número de classes e pacotes tende a resultar em piores valores, enquanto quantidades menores indicam melhores resultados para o *MoJoFM*. Estudos futuros poderiam explorar e explicar essas diferenças de forma mais detalhada.

Além disso, os modelos desenvolvidos podem servir como diretrizes para os desenvolvedores, funcionando como um diagnóstico. Melhorando a escolha das palavras

Tabela 5.1: Dois projetos de software com melhores e piores valores do MoJoFM.

Melhores	classes	pacotes	MoJoFM
org.apache.commons.cli	51	2	70,19
org.apache.yetus	11	3	88,47
Piores	classes	pacotes	MoJoFM
org.apache.lucene	9874	309	23,52
org.apache.poi	3645	209	23,03

e a documentação do código, esses modelos podem auxiliar na sugestão de uma remodelarização mais eficiente dos softwares.

Esta pesquisa utilizou algoritmos de incorporação de palavras estáticos. Como continuidade, estudos futuros podem explorar abordagens dinâmicas, como *ELMO* [94], *BERT* [32] e *GPT* [98], para avaliar seu impacto na análise realizada. Por fim, as técnicas de Inteligência Artificial Generativa podem melhorar o agrupamento de módulos de software ao identificar relações e características comuns entre eles

A aplicação de Redes Adversariais Generativas (*Generative Adversarial Networks* - *GANs*) [45] à remodelarização de software é uma abordagem nova e que pode trazer *insights* poderosos para a tarefa, especialmente quando as técnicas tradicionais de detecção de modularidade são limitadas. As *GANs* podem ajudar em cenários onde as abordagens tradicionais falham, especialmente ao lidar com sistemas grandes e complexos, onde é difícil encontrar limites ideais manualmente.

Para usar *GANs* para remodelarização, provavelmente adaptariamos a estrutura de uma *GAN* padrão para gerar propostas de modularização (ou seja, como dividir ou reorganizar um sistema de software) e avaliariamos essas propostas com base em como elas melhoram as métricas de modularidade (por exemplo, minimizando o acoplamento, melhorando a coesão).

- **Gerador:** O gerador proporia novas modularizações do sistema existente. Ele poderia aprender a criar novas partições do sistema de software em módulos com base em vários recursos, como dependências, chamadas de função, métricas de código, etc.

- **Discriminador:** O discriminador avaliaria as modularizações propostas com base em sua eficácia, ou seja, quão bem as mudanças propostas melhoram a modularidade do sistema. Ele poderia contar com métricas predefinidas como acoplamento (medida de interdependências entre módulos), coesão e métricas de complexidade.

Alguns desafios serão enfrentados, como por exemplo a) As GANs normalmente exigem uma grande quantidade de dados, mas a remodelarização de sistemas de software muitas vezes carece de exemplos suficientes de “boas” modularizações. Se não houver um conjunto de dados rotulado, você terá que confiar na avaliação orientada por métricas, que pode nem sempre estar alinhada com a intuição humana; b) GANs são conhecidas por sua instabilidade de treinamento. Para remodelarização, você pode encontrar problemas com convergência, especialmente se os sinais de feedback (avaliação do discriminador) não forem fortes ou claros o suficiente para guiar o gerador.

O uso de GANs para remodelarização de software envolve enquadrar a tarefa como uma em que o gerador propõe novos esquemas de modularização (ou seja, como agrupar componentes de software), e o discriminador avalia essas propostas com base em como elas melhoram métricas de modularidade, como acoplamento, coesão e complexidade. Com o tempo, o gerador melhora ao receber feedback do discriminador, resultando potencialmente em melhores decisões de remodelarização.

Essa abordagem pode ser particularmente útil quando os métodos tradicionais não conseguem encontrar a modularização ideal ou quando você está trabalhando com um sistema muito grande e complexo, onde decisões tomadas por humanos seriam muito demoradas ou inviáveis.

APÊNDICE A PROJETOS SELECIONADOS PARA GERAÇÃO DE MODELOS DE INCORPORAÇÃO DE PALAVRAS

Tabela A.1: Lista dos 183 projetos selecionados na Seção 4.3, com suas respectivas categorias, número de classes escritas na linguagem de programação Java e número de termos linguísticos identificados em cada projeto.

Projeto	Categorias	Classes	Termos
Accumulo	big-data	2.276	555.349
ActiveMQ	network-client, network-server	4.325	905.955
AGE	database	21	3.387
Airavata	big-data, cloud, database, network-client, network-server	1.077	211.687
Ambari	big-data	4.296	879.282
Anakia	library	12	2.783
Ant	build-management	1.226	294.779
Ant dotnet Library	build-management	22	7.034
Ant VSS Library	build-management	19	3.884
AntUnit	build-management	37	7.807
Any23	content	439	82.339
Archiva	build-management	690	269.804
Arrow	library	1.026	222.768
Avro	big-data, library	703	131.044
Axiom	library, xml	2.083	322.356
Axis2	http, network-client, xml, network-server	3.030	581.297
Batik	xml	1.655	370.837
Beam	big-data	5.506	1.358.195
Bigtop	big-data	205	31.127
BookKeeper	big-data	2.262	376.964
Brooklyn	cloud	317	815.645
BVal	library	319	56.403
Calcite	big-data	2.249	555.097
Camel	cloud, library	21.539	4.052.348
CarbonData	big-data	1.142	245.208
Cassandra	database	37	966.920
Cayenne	database, library, network-client, network-server, web-fwk, xml	4.166	1.397.662
Chainsaw	network-client	186	28.373

Tabela A.1 - Continuação da página anterior

Projeto	Categorias	Classes	Termos
CloudStack	cloud	5.963	482.599
Cocoon	database, http, network-client, network-server, web-framework, xml	689	154.648
Commons BCEL	library	607	103.235
Commons BeanUtils	library	258	57.703
Commons BSF	library	59	11.430
Commons Chain	library	95	19.173
Commons CLI	library	52	10.821
Commons Codec	library	143	45.214
Commons Collections	library	543	118.023
Commons Compress	library	554	126.671
Commons Configuration	library	462	119.143
Commons Crypto	library	101	19.845
Commons CSV	library	47	11.391
Commons Daemon	library	18	2.375
Commons DBCP	library	135	34.741
Commons DbUtils	library	92	16.025
Commons Digester	library	308	55.281
Commons Email	library	49	11.380
Commons Exec	library	49	10.806
Commons FileUpload	library	59	12.182
Commons Geometry	library	414	92.155
Commons Imaging	library	582	96.412
Commons IO	library	364	76.667
Commons JCI	library	66	10.789
Commons JCS	library	479	98.686
Commons Jelly	library	536	96.490
Commons JEXL	library	177	43.077
Commons XPath	library	213	40.586
Commons Lang	library	401	93.203
Commons Logging	library	66	17.259
Commons Math	library	1.154	252.549
Commons Net	library	277	66.406
Commons Numbers	library	192	43.946
Commons Pool	library	97	24.112
Commons Proxy	library	80	11.414
Commons RDF	library	165	27.952
Commons RNG	library	418	105.337
Commons SCXML	library	144	29.634
Commons Statistics	library	127	29.929

Tabela A.1 - Continuação da página anterior

Projeto	Categorias	Classes	Termos
Commons Text	library	194	42.325
Commons Validator	library	150	38.189
Commons VFS	library	506	98.620
Commons Weaver	library	115	16.975
Compress Ant Library	build-management	97	15.796
cTAKES	content	1.404	308.502
Curator	database	694	119.219
CXF	library, network-client, xml, network-server	7.239	1.313.294
Daffodil	big-data, library, xml	48	3.664
DataFu	big-data	261	51.117
Derby	database	2.861	976.892
Directory Server	network-server	1.909	286.218
Directory Studio	network-client	1.850	394.737
Drill	big-data	4.517	1.045.960
Empire-db	database	348	84.806
Flex	web-framework	2.818	490.100
Flink	big-data	13.985	3.116.435
Flume	big-data	722	158.124
Fluo	big-data	287	55.067
Fluo Recipes	big-data	96	18.113
FOP	xml	2.062	458.523
Fortress	library	325	89.830
FreeMarker	content	868	177.034
FtpServer	network-server	333	59.215
Geode	database	8.555	2.111.773
Geronimo	network-server, web-framework, http	12	620.683
Giraph	big-data	1.510	268.090
Gora	database	492	77.966
Groovy	library	1.746	46.729
Guacamole	network-client, network-server	707	127.888
HBase	database	4.649	1.317.511
Helix	big-data, cloud	1.414	333.526
Hive	database	1.414	274.838
Hop	big-data	3.849	895.425
HttpComponents Client	http, library, network-client	715	147.869
HttpComponents Core	http, library, network-client, network-server	937	170.898
Hudi	library	1.959	482.167
Ignite	big-data, cloud, database, network-client, network-server	11.715	2.576.781
IoTDB	database	4.135	826.621

Tabela A.1 - Continuação da página anterior

Projeto	Categorias	Classes	Termos
Ivy	build-management	664	132.858
IvyDE	build-management	194	36.654
Jackrabbit	database, library, xml, network-server	3.133	674.964
James	network-server	5.500	971.978
jclouds	cloud, library	5.874	1.104.476
Jena	library	6.318	1.168.891
JSPWiki	content	749	167.372
Kafka	big-data	3.502	813.857
Karaf	network-server	1.595	274.116
Kerby	library	1.043	171.797
Knox	big-data	1.115	196.405
Kudu	big-data	269	78.126
Linkis	big-data	1.822	262.213
Log4j 2	library	2.447	46.333
Lucene Core	database, library	5.328	1.243.255
Mahout	library	1.223	255.672
ManifoldCF	content	1.360	318.346
Maven	build-management	980	166.542
Maven Doxia	build-management	135	15.713
Mesos	cloud	24	3.826
MINA	network-client, network-server	634	122.977
MyFaces	web-framework	1.889	293.428
Nutch	web-framework	494	92.222
OFBiz	database, web-framework, content, xml	1.085	306.521
Olingo	library	1.155	216.483
ODDT	big-data	1.275	245.833
Oozie	big-data	1.388	245.833
OpenJPA	database, library	4.703	825.718
OpenMeetings	network-client, network-server	619	127.285
OpenNLP	library	1.009	176.227
OpenOffice	content	3.476	174.527
ORC	big-data, database, library	391	90.310
Parquet	big-data	780	166.286
PDFBox	content	1.381	259.771
Phoenix	big-data, database	2.051	506.206
Pig	database	1.766	436.125
Pivot	library	845	155.283
PLC4X	library, network-client	1.183	199.249
POI	content, library	3.688	843.802

Tabela A.1 - Continuação da página anterior

Projeto	Categorias	Classes	Termos
PyLucene	library	5.302	6.004
Rat	build-management	173	27.694
Roller	content	612	126.289
Sandesha2	xml	227	51.878
Santuario	library, xml	601	108.794
Scout	xml	179	33.931
ServiceMix	network-client, network-server, xml	107	16.374
Shiro	library, web-framework	726	135.342
Solr	network-server, web-framework	4.188	1.063.674
Spark	big-data	1.058	189.778
Spatial Information System	library	2.844	948.326
SSHD	network-server	1.320	296.803
Storm	big-data	212	42.300
Struts	web-framework	2.087	278.735
Subversion	build-management	174	24.295
Synapse	http, network-client, xml, network-server	1.283	242.468
Tapestry	web-framework	3.331	462.301
Tentacles	build-management	32	4.217
Tez	big-data	1.162	289.656
Thrift	http, library, network-client, network-server	283	48.951
Tika	content, library	1.649	366.479
Tomcat	http, network-server	2.569	192.488
TomEE	network-server	6.213	87.263
Torque	database	1.122	162.044
Traffic Control	cloud, content, network-server, http	260	52.102
Turbine	web-framework	296	65.888
Velocity DVSL	library	20	3.644
Vysper	network-server	581	106.566
Whisker	build-management	107	15.582
Wicket	web-framework	3.322	597.833
Xalan for Java XSLT Processor	xml	975	175.094
Xerces for Java XML Parser	xml	760	179.239
XML Commons External	xml	478	137
XML Commons Resolver	xml	31	8.765
XMLBeans	xml	1.279	129.545
Yetus	build-management, library	14	2.031
Zeppelin	big-data	955	220.464
ZooKeeper	database	900	219.584

REFERÊNCIAS

- [1] Gunjan Agicha. Word embeddings in NLP, May 2019.
- [2] Adem Akdogan. Word Embedding Techniques: Word2Vec and TF-IDF Explained, 2021.
- [3] P. Andritsos and V. Tzerpos. Information-theoretic software clustering. *IEEE Transactions on Software Engineering*, 31(2):150–165, February 2005.
- [4] A Anikin, A Katyshev, M Denisov, V Smirnov, and D Litovkin. Using online update of distributional semantics models for decision-making support for concepts extraction in the domain ontology learning task. *IOP Conference Series: Materials Science and Engineering*, 483:012073, March 2019.
- [5] N. Anquetil and T.C. Lethbridge. Experiments with clustering as a software remodularization method. In *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, pages 235–255, Atlanta, GA, USA, 1999. IEEE Comput. Soc.
- [6] Nicolas Anquetil and Timothy C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11(3):201–221, June 1999.
- [7] Amine Aoullay. Spectral Clustering for beginners, May 2018.
- [8] Rabeh Ayari. NLP: Word Embedding Techniques Demystified, March 2020.
- [9] Ashish Bansal. *Advanced Natural Language Processing with TensorFlow 2: Build effective real-world NLP applications using NER, RNNs, seq2seq models, Transformers, and more*. Packt Publishing, Birmingham Mumbai, 2021.
- [10] Saumya Bansal. Text Classification using BERT and TensorFlow, December 2021.
- [11] Marcio de Oliveira Barros. An analysis of the effects of composite objectives in multiobjective software module clustering. In *Proceedings of the fourteenth*

- international conference on Genetic and evolutionary computation conference - GECCO '12*, page 1205, Philadelphia, Pennsylvania, USA, 2012. ACM Press.
- [12] Márcio de Oliveira Barros, Fábio de Almeida Farzat, and Guilherme Horta Travassos. Learning from optimization: A case study with Apache Ant. *Information and Software Technology*, 57:684–704, January 2015.
 - [13] Abba Chouni Benabdellah, Asmaa Benghabrit, and Imane Bouhaddou. A survey of clustering algorithms for an industrial context. *Procedia Computer Science*, 148:291–302, 2019.
 - [14] Vishwas Bhanawat. The Architecture of Word2Vec, June 2019.
 - [15] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python*. O'Reilly, Beijing ; Cambridge [Mass.], 1st ed edition, 2009.
 - [16] Roberto Almeida Bittencourt and Dalton Dario Serey Guerrero. Comparison of Graph Clustering Algorithms for Recovering Software Architecture Module Views. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 251–254, Kaiserslautern, Germany, 2009. IEEE.
 - [17] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics*, 5:135–146, December 2017.
 - [18] Alokesh Bora. ML \textbackslashtextbar BIRCH Clustering, 2022.
 - [19] Jason Brownlee. A Gentle Introduction to Transfer Learning for Deep Learning, December 2017.
 - [20] Jason Brownlee. A Gentle Introduction to k-fold Cross-Validation, 2020.
 - [21] Oswald Campesato. *NLP fundamentals for developers*. Mercury Learning and Information, Duxbury, 2021.
 - [22] Alexandre Xavier Ywata Carvalho, Pedro Henrique Melo Albuquerque, Gilberto Rezende Almeida Junior, Rafael Dantas Guimarães, and Camilo Rey

Laureto. *Clusterização Hierárquica Espacial com Atributos Binários*. Instituto de Pesquisa Econômica Aplicada – ipea, 2009.

- [23] Marina Chatterjee. *Introduction to Spectral Clustering*, 2020.
- [24] Anshuman Chhabra, Vidushi Vashishth, and Prasant Mohapatra. Fair Algorithms for Hierarchical Agglomerative Clustering, June 2021.
- [25] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. Embedding Java Classes with code2vec: Improvements from Variable Obfuscation. 2020.
- [26] A Corazza, S Di Martino, and G Scanniello. A Probabilistic Based Approach towards Software System Clustering. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 88–96, Madrid, March 2010. IEEE.
- [27] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. Investigating the use of lexical information for software system clustering. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 35–44, Oldenburg, March 2011. IEEE.
- [28] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. Weighing lexical information for software clustering in the context of architecture recovery. *Empirical Software Engineering*, 21(1):72–103, February 2016.
- [29] Pratap Dangeti. *Statistics for machine learning: techniques for exploring supervised, unsupervised, and reinforcement learning models with Python and R*. Packt Publishing, Birmingham, UK, 2017.
- [30] Márcio De Oliveira Barros and Fábio De Almeida Farzat. What Can a Big Program Teach Us about Optimization? In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Günther Ruhe, and Yuanyuan Zhang, editors, *Search Based Software Engineering*, volume 8084, pages 275–281. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

- [31] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, September 1990.
- [32] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv, May 2019.
- [33] Devopedia. Text Corpus for NLP.
- [34] Niklas Donges. What Is Transfer Learning? Exploring the Popular Deep Learning Approach, June 2019.
- [35] D. Doval, S. Mancoridis, and B.S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *STEP '99. Proceedings Ninth International Workshop Software Technology and Engineering Practice*, pages 73–81, Pittsburgh, PA, USA, 1999. IEEE Comput. Soc.
- [36] Susan T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, & Computers*, 23(2):229–236, June 1991.
- [37] Vasiliki Efstathiou, Christos Chatzilenas, and Diomidis Spinellis. Word embeddings for the software engineering domain. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 38–41, Gothenburg Sweden, May 2018. ACM.
- [38] Alexandre F. Pinto, Adriana C. F. Alvim, and Marcio de O. Barros. ILS for the Software Module Clustering Problem. *XLVI Simpósio Brasileiro de Pesquisa Operacional, Pesquisa Operacional na Gestão da Segurança Pública*, September 2014.
- [39] William Fleshman. Spectral Clustering - Foundation and Application, 2019.
- [40] Richard Fuchshuber and Marcio de O. Barros. An Enhanced Iterated Local Search Algorithm for the Next Release Problem. *Intl. Trans. in Op. Res. xx (2021) 1–25*, pages 1–25, 2021.

- [41] Marcin Gabryel, Robertas Damaševičius, and Krzysztof Przybyszewski. Application of the Bag-of-Words Algorithm in Classification the Quality of Sales Leads. In Leszek Rutkowski, Rafał Scherer, Marcin Korytkowski, Witold Pedrycz, Ryszard Tadeusiewicz, and Jacek M. Zurada, editors, *Artificial Intelligence and Soft Computing*, volume 10841, pages 615–622. Springer International Publishing, Cham, 2018.
- [42] Rafael Gallardo García, Beatriz Beltrán, Darnes Vilariño, Claudia Zepeda, and Rodolfo Martínez. Comparison of Clustering Algorithms in Text Clustering Tasks. *Computación y Sistemas*, 24(2), June 2020.
- [43] Julian Gilyadov. Word2Vec Explained, March 2017.
- [44] Fred Glover and Gary A Kochenberger. *Handbook of Metaheuristics*. 2003.
- [45] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks, June 2014. arXiv:1406.2661 [stat].
- [46] Alind Gupta. ML \textbackslashtextbar Spectral Clustering, 2019.
- [47] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, December 2001.
- [48] Geoffrey E. Hinton. Training Products of Experts by Minimizing Contrastive Divergence. *Neural Computation*, 14(8):1771–1800, August 2002.
- [49] Alex Housley. Transfer Learning for Machine Learning, June 2021.
- [50] Purva Huilgol. Quick Introduction to Bag-of-Words (BoW) and TF-IDF for Creating Features from Text, February 2020.
- [51] Anil K. Jain. Data clustering: 50 years beyond K-means. *Pattern Recognition Letters*, 31(8):651–666, June 2010.
- [52] R. Janani and S. Vijayarani. Text document clustering using Spectral Clustering algorithm with Particle Swarm Optimization. *Expert Systems with Applications*, 134:192–200, November 2019.

- [53] javatpoint. Hierarchical Clustering in Machine Learning, 2020.
- [54] Dan Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition*. Prentice Hall, Upper Saddle River, NJ, 2. ed. [nachdr.] edition, 2009.
- [55] Aniruddha Kalbande. Parts of Speech Tagging Using NLP, August 2020.
- [56] Oscar Karnalim. TF-IDF Inspired Detection for Cross-Language Source Code Plagiarism and Collusion. *Computer Science*, 21(1), January 2020.
- [57] Ilia Karpov and A. Goroslavskiy. Application of BIRCH to text clustering. *CEUR Workshop Proceedings*, 934:102–105, January 2012.
- [58] Andrew P. King and Robert J. Eckersley. *Statistics for biomedical engineers and scientists: how to visualize and analyze data*. Academic Press, an imprint of Elsevier, London, United Kingdom, 2019.
- [59] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*, pages 201–210, Limerick, Ireland, 2000. IEEE Comput. Soc.
- [60] Oliver Kramer. K-Nearest Neighbors. In *Dimensionality Reduction with Unsupervised Nearest Neighbors*, pages 13–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
- [61] Luciana Kuchenbecker Araújo. Sentença, 2024. Publication Title: Sentença Type: Education.
- [62] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, March 2007.
- [63] Quoc V. Le and Tomas Mikolov. Distributed Representations of Sentences and Documents. arXiv, May 2014.

- [64] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology*, 58(7):1019–1031, May 2007.
- [65] "The Challenge of Building Corpus for NLP Libraries". The Challenge of Building Corpus for NLP Libraries, July 2020.
- [66] Ana Carolina Lorena and André C. P. L. F. de Carvalho. Uma Introdução às Support Vector Machines. *Revista de Informática Teórica e Aplicada*, 14(2):43–67, December 2007.
- [67] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. Iterated Local Search: Framework and Applications. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, pages 363–397. Springer US, Boston, MA, 2010.
- [68] Rudi Lutz. Evolving good hierarchical decompositions of complex systems. *Journal of Systems Architecture*, 47(7):613–634, July 2001.
- [69] Laurens van der Maaten and Geoffrey E. Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.
- [70] J. MacQueen. Some methods for classification and analysis of multivariate observations. *Berkeley Symposium on Mathematical Statistics and Probability*, 5.1:281–297, 1967.
- [71] K. Mahdavi, M. Harman, and R.M. Hierons. A multiple hill climbing approach to software module clustering. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 315–324, Amsterdam, Netherlands, 2003. IEEE Comput. Soc.
- [72] J.I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 103–112, Toronto, Ont., Canada, 2001. IEEE Comput. Soc.

- [73] S. Mancoridis, B.S. Mitchell, Y. Chen, and E.R. Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 50–59, Oxford, UK, 1999. IEEE.
- [74] S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, pages 45–52, Ischia, Italy, 1998. IEEE Comput. Soc.
- [75] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge University Press, New York, 2008.
- [76] Shie Mannor, Xin Jin, Jiawei Han, Xin Jin, Jiawei Han, Xin Jin, Jiawei Han, and Xinhua Zhang. K-Medoids Clustering. In Claude Sammut and Geoffrey I. Webb, editors, *Encyclopedia of Machine Learning*, pages 564–565. Springer US, Boston, MA, 2011.
- [77] O. Maqbool and H.A. Babri. The weighted combined algorithm: a linkage algorithm for software clustering. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 15–24, Tampere, Finland, 2004. IEEE.
- [78] Onaiza Maqbool and Haroon Babri. Hierarchical Clustering for Software Architecture Recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, November 2007.
- [79] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. An information retrieval approach to concept location in source code. In *11th Working Conference on Reverse Engineering*, pages 214–223, Delft, Netherlands, 2004. IEEE Comput. Soc.
- [80] Eduarda Marques. Aprenda de uma vez por todas qual é a diferença entre frase, oração e período, February 2023. Publication Title: Aprenda de uma vez por todas qual é a diferença entre frase, oração e período Type: Education.

- [81] Cláudia A. Martins, Maria Carolina Monard, and Edson Takashi Matsubara. Reducing the Dimensionality of Bag-of-Words Text Representation Used by Learning Algorithms. In *Proceedings AIA 2003*, EEUU, 2003. Acta Press.
- [82] Matthew Mayo. Building a Wikipedia Text Corpus for Natural Language Processing, November 2017.
- [83] Pierre Michaud. Clustering techniques. *Future Generation Computer Systems*, 13(2-3):135–147, November 1997.
- [84] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. 2013.
- [85] B.S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 744–753, Florence, Italy, 2001. IEEE Comput. Soc.
- [86] B.S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, March 2006.
- [87] Marlon C. Monçores, Adriana C.F. Alvim, and Márcio O. Barros. Large Neighborhood Search applied to the Software Module Clustering problem. *Computers & Operations Research*, 91:92–111, March 2018.
- [88] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse-engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.
- [89] Chris Nicholson. A Beginner’s Guide to Word2Vec and Neural Word Embeddings, 2020.
- [90] JOSE OROZCO. *CONVOLUTIONAL NEURAL NETWORKS IN TEXT CLASSIFICATION OF CONSPIRATORIAL CONTENT*. PhD Thesis, Tilburg University, School of Humanities and Digital Sciences, May 2021.

- [91] Vatsal Parsana. Explaining the Intuition of Word2Vec & Implementing it in Python, July 2021.
- [92] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.
- [93] Francisco Câmara Pereira and Stanislav S. Borysov. Machine Learning Fundamentals. In *Mobility Patterns, Big Data and Transport Analytics*, pages 9–29. Elsevier, 2019.
- [94] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations, 2018. Version Number: 2.
- [95] Samo Plibersek. Natural language processing with Python – POS tagging, dependency parsing, named entity recognition, topic modelling and text classification, 2020.
- [96] Kata Praditwong, Mark Harman, and Xin Yao. Software Module Clustering as a Multi-Objective Search Problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, March 2011.
- [97] Mudda Prince. Categorizing and POS Tagging with NLTK Python, September 2019.
- [98] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019.
- [99] Sanjana Reddy. GloVe and fastText — Two Popular Word Vector Models in NLP, July 2019.
- [100] Michele Risi, Giuseppe Scanniello, and Genoveffa Tortora. Architecture Recovery Using Latent Semantic Indexing and K-Means: An Empirical Evaluation. In *2010 8th IEEE International Conference on Software Engineering and Formal Methods*, pages 103–112, Pisa, Italy, September 2010. IEEE.

- [101] Gustavo S. Semaan, André L. V. Botelho, and Luiz S. Ochi. HEURÍSTICA BASEADA EM BUSCA LOCAL ITERADA PARA A RESOLUÇÃO DO PROBLEMA DE AGRUPAMENTO DE SISTEMAS ORIENTADOS A OBJETOS. *Simpósio Brasileiro de Pesquisa Operacional*, pages 1735–1744, 2011.
- [102] A. L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3(3):210–229, July 1959.
- [103] Amit Saxena, Mukesh Prasad, Akshansh Gupta, Neha Bharill, Om Prakash Patel, Aruna Tiwari, Meng Joo Er, Weiping Ding, and Chin-Teng Lin. A review of clustering techniques and developments. *Neurocomputing*, 267:664–681, December 2017.
- [104] Alexander Sboev, Ivan Moloshnikov, Dmitry Gudovskikh, Anton Selivanov, Roman Rybka, and Tatiana Litvinova. Deep Learning neural nets versus traditional machine learning in gender identification of authors of RusProfiling texts. *Procedia Computer Science*, 123:424–431, 2018.
- [105] Giuseppe Scanniello, Anna D’Amico, Carmela D’Amico, and Teodora D’Amico. Using the Kleinberg Algorithm and Vector Space Model for Software System Clustering. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 180–189, Braga, Portugal, June 2010. IEEE.
- [106] School of Software, Xinjiang University, Urumqi 830008, China, Rahim Khan, Yurong Qian, and Sajid Naeem. Extractive based Text Summarization Using KMeans and TF-IDF. *International Journal of Information Engineering and Electronic Business*, 11(3):33–44, May 2019.
- [107] Natasha Sharma. K-Means Clustering Explained, 2022.
- [108] Paul Smolensky. Information processing in dynamical systems: Foundations of harmony theory. In *Parallel Distributed Processing: Exploration in the Microstructure of Cognition*, volume Volume I: Foundations, page 56. MIT Press/Bradford Books, 1 edition, February 1986.

- [109] Ricardo Lemos de Souza, Fabiana Zaffalon Ferreira, and Silvia Silva da Costa Botelho. Proposta para Avaliação de Códigos Fonte com TF-IDF. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação (SBIE 2020)*, pages 112–121, Brasil, November 2020. Sociedade Brasileira de Computação.
- [110] R. Suganthi and S. Manimekalai. Spectral Clustering Based Classification Algorithm for Text Classification. *International Journal of Engineering Science Invention (IJESI)*, pages 36–41, 2018.
- [111] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. 2014.
- [112] Kenneth Sörensen and Fred W. Glover. Metaheuristics. In Saul I. Gass and Michael C. Fu, editors, *Encyclopedia of Operations Research and Management Science*, pages 960–970. Springer US, Boston, MA, 2013.
- [113] Alvin Jian Jia Tan, Chun Yong Chong, and Aldeida Aleti. E-SC4R: Explaining Software Clustering for Remodularisation, October 2021.
- [114] V. Tzerpos and R.C. Holt. MoJo: a distance metric for software clusterings. In *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, pages 187–193, Atlanta, GA, USA, 1999. IEEE Comput. Soc.
- [115] V. Tzerpos and R.C. Holt. ACCD: an algorithm for comprehension-driven clustering. In *Proceedings Seventh Working Conference on Reverse Engineering*, pages 258–267, Brisbane, Qld., Australia, 2000. IEEE Comput. Soc.
- [116] V. Tzerpos and R.C. Holt. On the stability of software clustering algorithms. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*, pages 211–218, Limerick, Ireland, 2000. IEEE Comput. Soc.
- [117] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. arXiv, December 2017.
- [118] P. Vatsal. Word2Vec Explained, July 2021.

- [119] Devraj Verma, Rakshith Yadhav, Sanjana M, and Varsha T P. A ChatBot using Seq2Seq and Bag of Words Model. *International Research Journal of Engineering and Technology (IRJET)*, 07(07):1751–1754, July 2020.
- [120] T.A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33–43, Amsterdam, Netherlands, 1997. IEEE Comput. Soc.
- [121] Niklaus Wirth. A Brief History of Software Engineering. *IEEE Annals of the History of Computing*, 30(3):32–39, July 2008.
- [122] Claes Wohlin. *Experimentation in software engineering*. Springer, New York, 2012.
- [123] Joseph A. Wood. *Improving software designs via the minimum description length principle*. PhD Thesis, University of Sussex, 1998.
- [124] Guohua Wu, Hairong Lin, Ershuai Fu, and Liuyang Wang. An Improved K-means Algorithm for Document Clustering. In *2015 International Conference on Computer Science and Mechanical Automation (CSMA)*, pages 65–69, Hangzhou, China, October 2015. IEEE.
- [125] J. Wu, A.E. Hassan, and R.C. Holt. Comparison of clustering algorithms in the context of software evolution. In *21st IEEE International Conference on Software Maintenance (ICSM’05)*, pages 525–535, Budapest, Hungary, 2005. IEEE.
- [126] Caiquan Xiong, Zhen Hua, Ke Lv, and Xuan Li. An Improved K-means Text Clustering Algorithm by Optimizing Initial Cluster Centers. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 265–268, Macau, China, November 2016. IEEE.
- [127] R. Xu and D. WunschII. Survey of Clustering Algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, May 2005.
- [128] Edward Yourdon and Larry L. Constantine. *Structured design: fundamentals of a discipline of computer program and systems design*. Prentice Hall, Englewood Cliffs, N.J, 1979.

- [129] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: an efficient data clustering method for very large databases. *ACM SIGMOD Record*, 25(2):103–114, June 1996.
- [130] Yuanyuan Zhang, Mark Harman, and S. Afshin Mansouri. The multi-objective next release problem. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation - GECCO '07*, page 1129, London, England, 2007. ACM Press.
- [131] Zhihua Wen and V. Tzerpos. An optimal algorithm for MoJo distance. In *MHS2003. Proceedings of 2003 International Symposium on Micromechatronics and Human Science (IEEE Cat. No.03TH8717)*, pages 227–235, Portland, OR, USA, 2003. IEEE Comput. Soc.
- [132] Zhihua Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pages 194–203, Bari, Italy, 2004. IEEE.