Vetores de Parágrafo Aplicados à Localização de Características e Bugs de Software

Allysson Costa e Silva

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2020

Allysson Costa e Silva

Vetores de Parágrafo Aplicados à Localização de Características e Bugs de Software

Tese de doutorado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Marcelo de Almeida Maia

Uberlândia
2020

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

S586
2020

Silva, Allysson Costa e, 1979-
Vetores de Parágrafo Aplicados à Localização de Características e Bugs de Software [recurso eletrônico] / Allysson Costa e Silva. - 2020.

Orientador: Marcelo de Almeida Maia.
Tese (Doutorado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.te.2020.404
Inclui bibliografia.

1. Computação. I. Maia, Marcelo de Almeida,1969-, (Orient.). II. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:
Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Ciência da Computação
Av. João Naves de Ávila, nº 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Tese, 04/2020, PPGCO				
Data:	06 de fevereiro de 2020	Hora de início:	**08hr15min**	Hora de encerramento:	**12hr20min**
Matrícula do Discente:	11423CCP002				
Nome do Discente:	Allysson Costa e Silva				
Título do Trabalho:	Vetores de Parágrafo Aplicados à Localização de Características e Bugs de Software				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Engenharia de Software				
Projeto de Pesquisa de vinculação:	-				

Reuniu-se na sala 1B132, Bloco 1B, Campus Santa Mônica, da Universidade Federal de Uberlândia, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Bruno Augusto Nassif Travençolo - FACOM/UFU, Fabiano Azevedo Dorça - FACOM/UFU, Eduardo Magno Lages Figueiredo - DCC/UFMG, Maurílio José Inácio - FACIT/FEMEC e Marcelo de Almeida Maia - FACOM/UFU, orientador do candidato.

Ressalta-se que o Prof. Dr. Eduardo Magno Lages Figueiredo, participou da defesa por meio de videoconferência desde a cidade de Belo Horizonte - MG e o Prof. Dr. Maurílio José Inácio da cidade de Montes Claros - MG . Os outros membros da banca e o aluno participaram *in loco*.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Marcelo de Almeida Maia, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado

Esta defesa faz parte dos requisitos necessários à obtenção do título de Doutor.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Bruno Augusto Nassif Travençolo**, **Professor(a) do Magistério Superior**, em 07/02/2020, às 10:24, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Fabiano Azevedo Dorça**, **Professor(a) do Magistério Superior**, em 07/02/2020, às 10:24, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Marcelo de Almeida Maia**, **Professor(a) do Magistério Superior**, em 07/02/2020, às 10:40, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Maurílio José Inácio**, **Usuário Externo**, em 09/02/2020, às 20:02, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Eduardo Magno Lages Figueiredo**, **Usuário Externo**, em 14/02/2020, às 18:13, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1849371** e o código CRC **44D8B1DE**.

Referência: Processo nº 23117.007421/2020-10 SEI nº 1849371

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da tese intitulada **"Vetores de Parágrafo Aplicados à Localização de Características e Bugs de Software"** por **Allysson Costa e Silva** como parte dos requisitos exigidos para a obtenção do título de **Doutor em Ciência da Computação.**

Uberlândia, ___ de _________ de _____

Orientador: ____
Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia (UFU)

Banca Examinadora:

____
Prof. Dr. Fabiano Azevedo Dorça
Universidade Federal de Uberlândia (UFU)

____
Prof. Dr. Bruno Augusto Nassif Travençolo
Universidade Federal de Uberlândia (UFU)

____
Prof. Dr. Eduardo Figueiredo
Universidade Federal de Minas Gerais (UFMG)

____
Prof. Dr. Maurílio José Inácio
Universidade Estadual de Montes Claros (Unimontes)

Aos meus pais, minha esposa e filha.

Agradecimentos

A Deus,
à minha família,
ao Professor Marcelo Maia,
aos meus colegas de trabalho,
e todos aqueles que me ajudaram direta ou indiretamente neste trabalho,
meus sinceros agradecimentos.

Resumo

Durante a fase de manutenção de software, os processos de localização de características e *bugs* de *software* exercem um relevante papel na recuperação da rastreabilidade entre descrições expressas em linguagem natural e trechos de código fonte. Entretanto, ferramentas automatizadas propostas para esta finalidade não tem apresentado acurácia que possa ser considerada como definitiva. Neste sentido, o objetivo principal deste trabalho foi o aprimoramento de vetores de parágrafos, produzidos por uma rede neural artificial empregada no algoritmo *Doc2vec*, e aplicados ao processo de localização de características e de *bugs* de software. Assim, melhor definidos, estes vetores puderam ser usados no cálculo de similaridades entre descrições de características e métodos do código fonte. Da mesma forma, a similaridade entre *bugs* e classes do código fonte pôde ser aprimorada. Para atingir este objetivo, foi aplicada uma taxa de aprendizado cíclica além da alteração da função de custo da rede neural que estavam associados ao algoritmo *Doc2vec*. Outras abordagens utilizadas reforçaram os ganhos em acurácia para o algoritmo *Doc2vec* a partir da combinação de rankings obtidos em ferramentas que expressam o estado da arte em Engenharia de Software. Um conjunto de abordagens que não apresentaram resultados satisfatórios também foram experimentadas e são descritas neste trabalho. Dentre estas podem ser citadas a melhoria da qualidade dos vetores representativos de métodos e classes do código fonte a partir de outros sistemas escritos na linguagem de programação Java; uso de pesos sintáticos diferenciados para termos que participam da composição de vetores relacionados às classes e métodos; e uso do algoritmo *fastText* para geração de vetores. Para cada abordagem proposta, ponderações foram feitas no sentido de se avaliar sua efetividade de uso em contextos de localização de *bugs* e características. Em resumo, com uma combinação das melhorias propostas para a rede neural artificial, a acurácia do estado da arte para a tarefa de localização de características pôde ser superada. Além disso, também foi possível melhorar a localização de *bugs* realizada a partir do uso do algoritmo *Doc2vec* bem como determinar quais hiperparâmetros eram mais influentes para melhoria da acurácia.

Palavras-chave: localização de características, *bugs*, recuperação de informação, Doc2vec.

Abstract

Throughout a software maintenance phase, the processes of feature and bug location play an important role in the retrieval of traceability between natural language and source code. However, existing automated tools, which were designed to perform these tasks, have not yet presented the desirable accuracy expected as a definitive outcome. In this sense, this work main objective seeks to attend to the improvement of paragraph vectors, produced by artificial neural network in the Doc2vec algorithm, and applied into the processes of feature and software bug location. By improving the vector quality, these were used in the calculation of similarities found between feature descriptions and source code methods. Also, improvements in identifying bugs similarity and source code classes were observed. In order to reach this result, a cyclic learning rate was applied to the task in addition to customizing neural network loss function, which were associated with the Doc2vec algorithm. Furthermore, other approaches used have also demonstrated the gains in accuracy for the Doc2vec algorithm, provided by the combination of rankings obtained in tools that express the state of the art in Software Engineering. Yet, a set of other minor approaches were applied to this work, such as: improvement in the quality of representative vectors from source code methods, departing from other systems' source code; usage of word syntactic influence within paragraphs extracted from source code; and usage of fastText algorithm to generate paragraph vectors. For each proposed approach, considerations were made in order to evaluate its effectiveness in feature and bug location tasks. In sum, the improvements proposed by this work to artificial neural network allowed to improve state of the art work in feature location tasks. Moreover, improvements in bug location were made by using the Doc2vec algorithm, and most influential hyper parameters were defined to improve accuracy.

Keywords: Feature Location, Bug Location, Information Retrieval, Doc2vec.

Lista de ilustrações

Figura 1 – LCS usando *Doc2vec* 37
Figura 2 – LCS usando LSI 38
Figura 3 – Localização de *Bugs* de Software usando a técnica LDA 40
Figura 4 – RNA do tipo rasa 41
Figura 5 – RNA do tipo profunda 42
Figura 6 – Processos de (a)*overfitting* e (b)*underfitting* em Redes Neurais 42
Figura 7 – Vetor de palavra para o termo "*return*"gerada pelo *Doc2vec* sobre o corpus do *software Eclipse Framerwork Communication* 44
Figura 8 – Exemplo utilizando o treinamento no algoritmo *Word2vec* 45
Figura 9 – Arquiteturas CBOW(a) e Skip-gram(b) 47
Figura 10 – Arquiteturas PV-DM(a) e PV-DBOW(b) 48
Figura 11 – Exemplo de palavras positivas e negativas usadas na modalidade *Negative Sampling* 49
Figura 12 – O processo de operação da função de custo *Negative Sampling* 50
Figura 13 – Variação de taxa de aprendizado Triangular 62
Figura 14 – MRR médio para o experimento com RNA 65
Figura 15 – Melhores e piores resultados para o teste de Massa 76
Figura 16 – Correlação entre abordagens e hiperparâmetros no teste de 720 combinações 77
Figura 17 – Abordagem FED aplicada à Localização de *Bugs de Sofware* 84
Figura 18 – Abordagem ESL aplicada à Localização de *Bugs de Sofware* 87
Figura 19 – Polissemia entre diferentes domínios de problema 95
Figura 20 – Enriquecimento de *vetores de parágrafos* para LCS 130

Lista de tabelas

Tabela 1 – Exemplificando o uso de uma janela de tamanho 2 no *Word2vec* 45
Tabela 2 – Ferramentas Utilizadas 60
Tabela 3 – Hardware utilizado nos experimentos 60
Tabela 4 – *Software* analisados 61
Tabela 5 – Hiperparâmetros testados para localização de *bugs* de *software* 71
Tabela 6 – Ferramentas Utilizadas 74
Tabela 7 – *software* analisados 74
Tabela 8 – *M*elhores e piores resultados de MRR para cada *software* 75
Tabela 9 – Comparativo entre LBS usando métodos e classes 79
Tabela 10 – *R*esultados de MRR para cada *software* - Abordagem FED 84
Tabela 11 – *R*esultados de MRR para cada *software* - Abordagem ESL 87
Tabela 12 – Comparação do MRR entre abordagens DV*, DV**+LRT+NGS50 e FST 89
Tabela 13 – *A*bordagens testadas sem sucesso 89
Tabela 14 – MRR médio para as abordagens DV**+LRT e DV**+NGS 120
Tabela 15 – Configurações discretas de Hiper-parâmetros para pré-testes 121
Tabela 16 – MRR para as abordagens DV**, BZZ, LRT, NGS50, WDS50, HSM . 122
Tabela 17 – *Software* utilizado na abordagem 131
Tabela 18 – Abordagem de enriquecimento semântico: Média do MRR para 10 execuções 131

Lista de siglas

API *Application Programming Interface*

BZZ - abordagem que se utiliza do ranking provido pela ferramenta Blizzard para melhorar a acurácia do algoritmo Doc2vec.

CLR *Cyclical Learning Rates*

CBOW *Continuous bag-of-words*

DV* - Estudo de Referência

DV** - Reprodução do Estudo de referência

DM - hiperparâmetro que define o modo de treinamento do algoritmo Doc2vec: PV-DM ou PV-DBOW;

FED - Abordagem para enriquecimento de *word embeddings* e *document vectors*

HSM - hiperparâmetro que define a função de custo Hierarquical Sampling da rede neural aplicada ao algoritmo Doc2vec;

IDE - *Integrated Development Environment*

JDT *Java Development Tools*

LCS - Localização de características de Software

LBS - Localização de bugs de Software

LSI - *Latent Semantic Indexing*

LDA - *Latent Dirichlet Allocation*

LDA-GA - Abordagem de utilização de algoritmos genéticos para calibrar a técnica LDA

LRT - *Customização da Política de adaptação da taxa de aprendizado denominada Cyclical Learning Rate*

MAP - *Mean Average Precision*

MLP - *Multilayer Perceptron*

MRR - *Mean Reciprocal Rank*

NGS - *Função de custo Negative Sampling*

PLN - Processamento de linguagem natural

pLSI *Probabilistic Latent Semantic Indexing*

PV *Paragraph Vector*

PV-DBOW *Paragraph Vector Bag of Words model*

PV-DM *Paragraph Vector Distributed Memory*

RI *Recuperação de Informação*

RNA *Rede Neural Artificial*

RNAs *Redes Neurais Artificiais*

SVD - *Singular Value Decomposition*

SPR - *Scenario based Probabilistic Ranking*

SWT *Standard Widget Toolkit*

WDS - hiperparâmetro relativo ao tamanho da janela escolhido.

VSM - *Vector Space Model*

XML - *Extensible Markup Language*

Sumário

1 INTRODUÇÃO **25**
1.1 Objetivos de Pesquisa e Contribuições **28**
1.2 Organização do Texto **29**

2 FUNDAMENTAÇÃO TEÓRICA **31**
2.1 Manutenção de *Software* **31**
2.1.1 Compreensão de Programas 32
2.1.2 Análise de impacto de mudanças 34
2.1.3 Características e *Bugs* de *Software* 35
2.1.4 Localização de Características e *bugs* de *Software* 35
2.2 Técnicas para LCS e LBS baseadas em frequência de termos . **38**
2.2.1 *Latent Semantic Indexing* (LSI) 38
2.2.2 *Latent Dirichlet Allocation* (LDA) 39
2.3 Técnicas baseadas em Rede Neural Artificial (RNA) **40**
2.3.1 Vetores de Palavras 43
2.3.2 Vetores de Parágrafos 47
2.3.3 Algoritmos usados pelo algoritmo *Doc2vec* 49
2.4 Medidas de Avaliação da Acurácia em localização de Características e *Bugs de Software* **51**
2.4.1 Medidas para cálculo de similaridade entre vetores 51
2.4.2 *Mean Reciprocal Rank* - MRR 51
2.4.3 Mean Average Precision - MAP 52
2.5 Resumo do Capítulo **53**

3 UM ESTUDO EMPÍRICO PARA LOCALIZAÇÃO DE CARACTERÍSTICAS DE SOFTWARE USANDO VETORES DE PARÁGRAFO APRIMORADOS **55**
3.1 Referencial Teórico **55**

3.2 **Planejamento Experimental** **57**
3.3 **Aspectos Técnicos do Experimento** **57**
3.4 **Questões de Pesquisa** **58**
3.5 **Materiais e Métodos** **58**
3.5.1 Estudo de Referência 59
3.5.2 *Software*, ferramentas e hardware utilizados 60
3.5.3 Caracterização dos projetos de *software* 60
3.5.4 Abordagem utilizando a taxa de aprendizado cíclica 61
3.5.5 Abordagem utilizando a função de custo da Rede Neural Artificial . . . 63
3.5.6 Combinação de abordagens 63
3.6 **Resultados Experimentais** **63**
3.7 **Discussão dos Resultados** **66**
3.8 **Resumo do Capítulo** **67**

4 **APRIMORAMENTO DO USO DE *DOC2VEC* NA LOCALIZAÇÃO DE *BUGS* DE *SOFTWARE*** **69**
4.1 **Referencial Teórico** **69**
4.2 **Planejamento Experimental e Metodologia utilizada** **70**
4.3 **Aspectos técnicos do experimento** **72**
4.4 **Questões de Pesquisa** **72**
4.5 **Materiais utilizados** **72**
4.5.1 Estudo de Referência 72
4.5.2 *software* e ferramentas utilizados 74
4.6 **Discussão dos Resultados** **74**
4.6.1 Combinação de hiperparâmetros do *Doc2vec* e Abordagens 75
4.6.2 Usando métodos para Localização de *Bugs* 78
4.7 **Resumo do Capítulo** **79**

5 **RESULTADOS NEGATIVOS E IMPLICAÇÕES** **81**
5.1 **Estudo 1 - O uso de Enriquecimento Semântico (FED)** **82**
5.1.1 Materiais e Métodos 82
5.1.2 Resultados e Discussão 83
5.2 **Estudo 2 - Atribuição de pesos distintos aos elementos sintáticos do parágrafo (PDP)** **85**
5.2.1 Materiais e Métodos 85
5.2.2 Resultados e Discussão 86
5.3 **Estudo 3 - Uso do *fastText* (FST)** **88**
5.3.1 Materiais e Métodos 88
5.3.2 Resultados e Discussão 88
5.4 **Resumo do Capítulo** **89**

6 TRABALHOS RELACIONADOS 91
6.1 Localização de Características usando o *grep* 91
6.2 Localização de Características usando *Latent Semantic Indexing* e *PROMESIR* 92
6.3 Localização de Características usando *Latent Dirichlet Allocation* e Algoritmos Genéticos 93
6.4 Melhoria de vetores com enriquecimento semântico 93
6.5 O uso de vetores de Palavras e Parágrafos sobre o código fonte 94
6.6 Resumo do Capítulo 96

7 CONCLUSÃO 97
7.1 Ameaças à Validade 98
7.2 Trabalhos futuros 99
7.3 Considerações finais 101

REFERÊNCIAS 103

APÊNDICES 117

APÊNDICE A – VALORES NUMÉRICOS PARA MRR COM DESVIO PADRÃO 119

APÊNDICE B – PRÉ-TESTES PARA DEFINIÇÃO DO TAMANHO DO VETOR DE PARÁGRAFOS 121

APÊNDICE C – UM EXEMPLO UTILIZANDO O ALGORITMO DOC2VEC DO FRAMEWORK GENSIM . . . 123

APÊNDICE D – ENRIQUECIMENTO SEMÂNTICO COM SOFTWARE DO MESMO DOMÍNIO DE PROBLEMA 129
D.0.1 Resultados Preliminares 131

Capítulo 1

Introdução

Existe um crescente interesse em se criar ferramentas que apoiem os desenvolvedores durante o ciclo de vida de um sistema de *software*. Uma das fases deste ciclo diz respeito à manutenção de *software* que corresponde ao conjunto de atividades executadas sobre um sistema no período posterior à sua implantação. Estas atividades promovem modificações de componentes do sistema com o objetivo de corrigir erros de especificação, codificação ou projeto. Além disso, esta fase também engloba a adição de novos componentes ao sistema [1]. Penny G. e Takang[2] reportam casos em que os custos de manutenção de sistemas de *software* variam entre 49 e 75% em relação ao custo total dispendido durante o ciclo de vida do *software*. Segundo Nishizono et al.[3], grande parte do custo associado à manutenção de *software* deve-se principalmente à dificuldade de se compreender o código fonte existente, notoriamente, separar aquelas partes do código que necessitam ser alteradas.

Desenvolvedores gastam mais da metade de seu tempo tentando compreender o sistema para que possam então modificá-lo corretamente [4]. Durante o processo de compreensão, o desenvolvedor comumente recebe a descrição de um *bug*, ou a solicitação de incremento ou ainda a alteração de uma funcionalidade do *software*. Esta demanda geralmente possui trechos de linguagem natural, *snippets* de código ou uma combinação de ambos [5] e precisam ser mapeados para as partes do código fonte. Para sistemas orientados a objeto, as formas tradicionais de modularização de código fonte, que são baseadas em métodos, classes e pacotes, ajudam na organização do mesmo. Contudo, para uma efetiva compreensão do código fonte a ser alterado em uma tarefa de manutenção de *software*, novas visões e organizações do código se tornaram necessárias. Isto se deve ao fato de muitas funcionalidades terem suas implementações espalhadas pelo código fonte, conforme apontam Dit et al.[6], em um quadro de baixa coesão que ultrapassa os limites dos módulos citados [7]. Neste sentido, Sobreira e Maia[8], desenvolveram uma ferramenta para analisar o espalhamento de características de software sobre o código fonte. Nesta ferramenta é possível fazer um análise visual dos rastros de execução de um software seccionando características com o uso de diferentes cores em mapas visuais.

O desenvolvimento de ferramentas que permitam localizar quais partes do código fonte

são acionadas durante a execução de uma característica ou ocorrência de um *bug* de *software* tornou-se objeto de estudo de diversos trabalhos científicos que utilizaram técnicas de Recuperação de Informação (RI) baseadas em frequência de termos [9, 10] e mais recentemente em modelos de aprendizado de máquina [11, 12]. Estes modelos utilizam um conjunto de parâmetros, denominados comumente como hiperparâmetros[1], que geralmente são definidos antes de se treinar o modelo de aprendizado. A maioria dos modelos de Aprendizado de Máquina trazem um conjunto de hiperparâmetros que controlam o funcionamento interno do algoritmo. Para Mikolov et al.[13], a escolha do algoritmo de treinamento e valores para seus hiperparâmetros é uma decisão específica que deve ser tomada levando em conta que diferentes problemas possuem distintas calibrações ótimas para seus hiperparâmetros. No caso de modelos baseados em redes neurais, podem ser citados como exemplo de hiperparâmetros, o número épocas para treinamento ou o número de neurônios das camadas internas da rede. Posteriormente, novos trabalhos foram propostos no sentido de se encontrar a calibração mais adequada para estes hiperparâmetros de forma a maximizar a acurácia dos processos de localização de características de software (LCS) e localização de *bug* de software (LBS) [14, 15, 16]). A correta calibragem destes parâmetros impacta diretamente a capacidade de generalização do modelo e na acurácia de predição para novos dados [17]. As ferramentas assim criadas permitem que desenvolvedores que não possuam um conhecimento profundo da estrutura e código fonte de um *software* possam localizar os artefatos relacionados à sua tarefa de manutenção de maneira rápida e precisa.

Como discutido anteriormente, apesar de existirem tentativas no sentido de evoluir técnicas e ferramentas que viabilizem uma melhor recomendação ao desenvolvedor no que se refere às atividades de LCS e LBS, o maior problema associado a estas atividades continua sendo a baixa acurácia obtida para algumas consultas no processo de recuperação de informação. O resultado das ferramentas é uma lista ranqueada de artefatos a serem analisados para efetivamente localizar a característica ou o *bug*. O cenário ideal é quando o artefato se encontra na primeira posição da lista. Neste contexto, Lukins, Kraft e Etzkorn[10] analisaram um conjunto de 216 *bugs* do Eclipse[2], coletados a partir de 13 versões diferentes do *software* e 106 *bugs* do *software* Rhino[3] em 12 versões. Para ranqueamento, eles utilizam a técnica Latent Dirichlet Allocation (LDA), descrita no Capítulo 2. Para o Eclipse, levando-se em conta a melhor calibração da técnica, para 28% dos *bugs* no estudo, o resultado de localização demanda a análise de pelos menos 1000 métodos, para cada *bug*, antes que seja possível encontrar um método relevante do código fonte relacionado à descrição do *bug*. Para o Rhino, levando-se em conta que este *software* é pequeno, os autores conseguiram um resultado em que 12 dos 106 *bugs* necessitam a análise de mais de 1000 métodos. Agrava esta situação o fato que mais de 26 dentre os 106 *bugs* analisa-

[1] https://docs.microsoft.com/pt-br/dotnet/machine-learning/resources/glossary
[2] https://www.eclipse.org/
[3] https://discourse.mcneel.com/t/rhino-web-browser/5504

dos demandem a inspeção de mais de 176 métodos para consecução da tarefa. Trata-se de uma limitação severa para o desenvolvedor. Neste ponto, observa-se dois pontos de evolução: o aprimoramento das técnicas voltadas a este fim ou a melhoria dos artefatos envolvidos no *pipeline* de execução de uma consulta ao código fonte como reformulação de consultas e tratamento do código fonte.

Recentemente, novas abordagens para recuperação de rastreabilidade entre artefatos de *software* tem utilizado técnicas baseadas em redes neurais [11, 18]. Corley, Damevski e Kraft[11] propuseram uma abordagem de LCS utilizando o algoritmo *Doc2vec* em seis diferentes *software* de código aberto, usando uma métrica denominada Mean Reciprocal Ranking (MRR) para avaliação dos resultados. *Doc2vec* é um algoritmo que permite representar parágrafos sob a forma de vetores numéricos de baixa dimensionalidade. Em relação ao problema em questão, se forem observados a média dos melhores valores obtidos para a medida MRR, pode-se observar que esta não ultrapassa 0,0772. Isto equivale a dizer que, em média, em um conjunto de 91 consultas, um desenvolvedor necessita inspecionar 13 métodos do código fonte para cada consulta submetida à ferramenta de localização. Mas, apesar deste aparente sucesso, este quadro não é tão eficiente assim. O caso em questão refere-se a um experimento real utilizando o *software* ArgoUML[4]. A análise individual das consultas submetidas nos indica que em alguns casos o desenvolvedor demandará a análise de mais de 11000 métodos antes de encontrar algum relevante para iniciar o seu trabalho de manutenção.

Uma outra possibilidade para melhoria da acurácia das tarefas de LCS e LBS seria o uso de técnicas de processamento de linguagem natural (PLN). Hindle et al.[19] defendem que a utilização de modelos estatísticos aplicados a linguagem natural pode ser aplicada em linguagens de programação uma vez que o código fonte dos *software* também possuem natureza simples e repetitiva. De maneira explícita, os autores afirmam que as atividades de sumarização e busca de código fonte são similares àquelas relacionadas à tradução de textos em linguagens naturais. Chen e Monperrus[20] reafirmam esta visão. Para eles, assim como as linguagens naturais possuem caracteres, palavras e parágrafos, as linguagens de programação possuem diferentes níveis de granularidade como variáveis, expressões, comandos e métodos. Entretanto, o modo adequado de como se utilizar estas técnicas para tratamento de código fonte permanece como uma questão de pesquisa em aberto que precisa ser elucidada.

Nesta perspectiva, o foco desta trabalho é melhorar o processo de LCS e LBS, avaliando e aprimorando o estado da arte, o qual se sustenta em técnicas baseadas no paradigma de Redes Neurais. Mais especificamente, foram estabelecidas abordagens cujo objetivo fosse aumentar a acurácia no processo de localização de métodos e classes que correspondam à implementação de características e *bugs* no código fonte. Desta forma, as seguintes vertentes foram propostas para serem exploradas: melhoria da rede neural sob o qual

[4] argouml.tigris.org

o algoritmo *Doc2vec* está baseado; enriquecimento semântico da base de código fonte e descrições de características e *bugs* utilizadas; priorização de métodos e classes do código fonte com base na combinação de *rankings* de técnicas diferentes.

1.1 Objetivos de Pesquisa e Contribuições

Apesar da criação de ferramentas automatizadas, a acurácia calculada após o processo de LCS e LBS deve ser melhorada em função do grande número de artefatos que o desenvolvedor ainda precisa analisar em tarefas de manutenção de *software*. Desta forma, o objetivo deste trabalho é apresentar um conjunto de abordagens que sejam capazes de aumentar a acurácia nestes processos. De maneira específica, trata-se de reduzir o número de métodos e classes do código fonte que necessitam ser analisados até que o desenvolvedor encontre aqueles que são relevantes em relação a uma descrição de característica de *software*, *bug* ou solicitação de mudança. As principais contribuições do presente trabalho são:

1. Combinações de abordagens, baseadas na utilização do algoritmo *Doc2vec*, que permitam individualmente a melhoria da acurácia do processo de LCS e LBS, que pode ser mensurada a partir de métricas de RI como *Mean Reciprocal Rank* (MRR).

2. Evolução do processo de localização de *bugs* baseado na combinação de técnicas aplicadas em ferramentas que utilizam o algoritmo *Doc2vec* que espelhem o estado da arte nesta tarefa de manutenção.

3. Uma ferramenta que suporte a LCS com desempenho superior ao estado da arte.

Enunciado da Tese

A acurácia do processo de localização de características e *bugs* de *software* pode ser melhorada a partir da descoberta e calibração adequada da técnica *Doc2vec*, tais como função de custo *Negative Sampling* e o modo de treinamento de vetores de parágrafos *Paragraph Vectors Distributed Memory* (PV-DM). Além disso, outros fatores como a utilização de uma taxa de treinamento cíclica durante o treinamento da rede permite a melhoria da qualidade dos vetores representativos de elementos do código fonte como classes e métodos com consequente aumento de acurácia do processo de busca. Por fim, a combinação dos *rankings* providos por técnicas que expressam o estado da arte propicia a melhoria da localização de *bugs* e características de *software* para tarefas que se utilizam do algoritmo *Doc2vec*.

1.2 Organização do Texto

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica e técnicas que têm sido comumente aplicadas à localização de características e *bugs* de *software*; o Capítulo 3 descreve um experimento em que a localização de característica é facilitada a partir de melhorias em Redes Neurais Artificiais providas pela comunidade científica, além de delinear calibrações de hiperparâmetros para o algoritmo *Doc2vec*; o Capítulo 4 descreve uma abordagem voltada à localização de *bugs* de *software* em que uma combinação de técnicas permite ao *Doc2vec* melhorar sua acurácia; O Capítulo 5 retrata um conjunto de abordagens que não obteve bons resultados permitindo ao leitor uma visão do que empreender nesta área; o Capítulo 6 descreve os trabalhos relacionados; e por fim, o Capítulo 7 traz a conclusão do trabalho.

Capítulo 2

Fundamentação Teórica

Neste capítulo foram apresentados os principais conceitos utilizados para o desenvolvimento do presente trabalho. Assim, são abordados conceitos relativos às principais técnicas de RI que tem sido aplicadas à LCS dentro do contexto de manutenção de *software*. Modelos de aprendizado de máquina baseados em Redes Neurais Artificiais (RNA) também foram utilizados neste trabalho. Portanto, deu-se mais ênfase em pontos relevantes da teoria de RNAs. Isto permitirá ao leitor, sempre que necessário, recorrer a este capítulo para embasar seu entendimento. Adicionalmente, separou-se, em cada capítulo do trabalho, uma seção para tratar conceitos mais específicos relativos ao experimento realizado.

2.1 Manutenção de *Software*

O padrão IEEE 1219 [21][1] define manutenção de *software* como:

> A modificação de um produto de *software* após sua implantação para correção de falhas, melhoria de performance ou outros atributos ou adaptação do produto para um novo ambiente.

Segundo Binkley e Lawrie[22], a fase de manutenção inicia-se após o *software* entrar em produção, isto é, após sua entrega e colocação em execução da sua primeira versão funcional. Durante a fase de manutenção diversos tipos de atividades são realizadas pelo desenvolvedor. Conforme o tipo de atividade desenvolvida, Bennett e Rajlich[23] expõe as seguintes classificações para as atividades de manutenção de *software*:

1. Adaptativa - Este tipo de manutenção é necessário quando ocorrem mudanças no ambiente operacional do *software*.

2. Perfectiva - Necessária quando novos requisitos do usuário são introduzidos.

[1] https://ieeexplore.ieee.org/document/257623

3. Corretiva - Destinada à correção de erros.

4. Preventiva - Manutenção realizada com o intuito de prevenção de problemas futuros.

Independente do tipo de manutenção a ser realizado, o objetivo final é sempre manter o sistema funcionando e atendendo às necessidades dos usuários. Entretanto, existem altos custos associados à manutenção de *software*. Banker, Davis e Slaughter[24] destacam que nas últimas décadas o custo da manutenção de *software* permanece elevado e estimam que seu valor esteja entre 50 a 80% dos gastos do orçamento do sistema. Assim, Robson et al.[25] ressaltam como dificuldade o fato do desenvolvedor lidar com código fonte com pouca ou nenhuma documentação. Junte-se a isso a questão do desenvolvedor muitas vezes não ser o projetista original do sistema, ou ter esquecido detalhes de projeto ou código após certo tempo. Desta forma, Banker, Davis e Slaughter[24] destacam que a maior parte do tempo disponível para manutenção é gasto pelo desenvolvedor na tentativa de compreender o código fonte da funcionalidade do *software* a ser modificada. Este trabalho propõe melhorar a tarefa de manutenção de *software* a partir da melhoria da acurácia do processo de localização de características e *bugs* de *software*.

2.1.1 Compreensão de Programas

A compreensão de *software* é requerida durante as tarefas de manutenção, reuso, migração, reengenharia ou incremento de funcionalidades de um sistema [26]. Diariamente, desenvolvedores lidam com milhares de linhas de código e antes de realizar quaisquer alterações necessitam localizar e compreender determinadas partes do *software*. Um exemplo de tal complexidade são os códigos fonte dos *software* Eclipse e Openoffice.org que são formados por centenas de arquivos do tipo XML, C++, Java dentre outros. Desta forma, a compreensão de programas é um pré-requisito fundamental para manutenção de sistemas de informação atuais e legados.

Durante o processo de compreensão de programas, o desenvolvedor pode fazer uso de ferramentas automatizadas que o auxilie no entendimento de um código fonte desconhecido. Tais ferramentas analisam o código fonte e demais artefatos do sistema e extraem informações específicas de interesse do desenvolvedor. Rajlich[27] classifica as ferramentas de compreensão de programas nas seguintes categorias:

1. Navegadores - Ajudam o programador a navegar através do *software* explorando suas dependências, isto é, os relacionamentos entre a definição de variáveis, procedimentos e tipos.

2. Ferramentas baseadas no casamento de padrões - Analisam o texto do código fonte em busca por padrões informados pelo programador.

3. Ferramentas de Segmentação de Código - Reduzem a quantidade de código a ser inspecionada durante uma tarefa de manutenção.

4. Ferramentas de Documentação - Suportam a criação e navegação de uma documentação associada ao *software*.

As ferramentas citadas anteriormente quase sempre envolvem a utilização de Processamento de Linguagem Natural (PLN), que é um campo da Ciência da Computação derivado da Linguística e Inteligência Artificial. Técnicas baseadas em PLN baseiam-se no desenvolvimento de modelos computacionais para a realização de tarefas que dependem de informações expressas em alguma língua natural como o Inglês, Português e outros [19]. Dentre as principais tarefas destacam-se: tradução e interpretação de textos, sumarização, reconhecimento de discurso, análise de sentimento, criação de *chat bots*, busca de informações em documentos etc.

Neste trabalho, com o objetivo de apoiar tarefas de Engenharia de *Software*, utilizou-se os seguintes recursos de PLN: stemização[2], remoção de *stop words*, atribuição de *labels* em documentos e *tags* em termos, classificação textual, indexação de documentos. Muitas tarefas de PLN partem de um *corpus*. Um *corpus* (no plural *corpora*) é definido como uma coleção de documentos em linguagem natural construída com alguma finalidade específica [28]. Neste trabalho foi utilizado dois tipos de *corpus*: um formado por uma coleção de métodos; e outro formado por classes do código fonte.

Em Kim et al.[29], os autores buscam melhorar o problema da detecção de intenção em sentenças do discurso oral usando técnicas de PLN. No trabalho ressalta-se que o enriquecimento tem se restringido à granularidade de palavras, em tarefas como busca por similaridade e analogia de termos, sem atingir o nível de sentenças ou parágrafos. Outro ponto citado é que dicionários desempenham o papel de repositórios de contextos para o enriquecimento. Os autores utilizam então um conjunto de vetores de termos pré-treinados com o algoritmo *Glove* e dimensionalidade 200 para os vetores gerados. Por fim, uma técnica para análise de sentenças é proposta dentro do contexto do reconhecimento de discurso oral. Em outro trabalho, Bojanowski et al.[30] propõem a utilização de *n-grams* , isto é, fragmentos de palavras para enriquecer o espaço vetorial gerado. Sua aplicação foi testada nas atividades de busca de similaridade e analogia entre termos do texto. A abordagem é baseada no modelo *skip-gram* de Mikolov et al.[31]. Cada fragmento foi representado por um vetor, sendo possível a descoberta de novos termos que estavam implícitos ao se utilizar uma abordagem tradicional.

Por fim, Ye et al.[32] demonstram a viabilidade de se aprimorar a localização de *bugs* a partir da utilização de fontes de informações diversas além do próprio código fonte que está relacionado ao *bug*.

Geralmente, as partes do código fonte que implementam uma determinada funcionalidade não podem ser encontradas de maneira óbvia. O quadro que se apresenta na maioria dos sistemas de *software* é caracterizado pela presença de uma documentação desatualizada, quando ela sequer existe. Outro complicador é a ausência dos desenvolve-

[2] Stemização http://www.nilc.icmc.usp.br/nilc/download/lematizacao$_versus_steming.pdf$

dores originais que não estão mais presentes ou a visão obsoleta de um sistema que evoluiu durante o tempo. Diante desta situação, segundo Eisenbarth, Koschke e Simon[33], o processo de manutenção de *software* pode introduzir mudanças inconsistentes no sistema que provoquem uma degradação geral de sua estrutura. Assim, conclui-se que a compreensão do sistema é altamente dependente da estrutura do sistema que se está analisando. Por exemplo, classes e métodos muito longos e que apresentem baixa coesão podem dificultar a busca de uma implementação onde esteja um *bug* específico. Isto se dá porque o desenvolvedor geralmente parte de uma descrição de *bug* e tenta encontrar o método ou classe equivalente no código fonte. Uma das tarefas realizadas neste trabalho foi justamente a localização de *bugs* no código fonte a partir da análise de classes de sistemas orientados a objetos.

Muitos trabalhos científicos recentes na área de Engenharia de *Software* apostam na junção de várias fontes de documentação para melhoria do processo de compreensão de *software*. Buscando-se atingir esta meta, a recuperação da rastreabilidade entre artefatos tem sido amplamente estudada para apoiar tarefas de compreensão de programas e consequentemente o processo de manutenção de *software*. Neste sentido, são exemplos de tarefas comumente estudadas pela comunidade científica: localização de *bugs* [10], localização de características, mapeamento de requisitos no código fonte de classes [15], dentre outros. Portanto, a aproximação proporcionada pela análise semântica de artefatos de *software* permite atacar o problema de compreensão de *software* enfrentado pela maioria dos desenvolvedores que necessitam dar manutenção no código fonte de um *software* para incrementar novas características ou corrigir aquelas já existentes.

2.1.2 Análise de impacto de mudanças

A análise de impacto compreende à identificação dos componentes que são impactados por uma solicitação de mudança (*change request*) [34]. Segundo Tripathy e Naik[34], a análise de impacto é realizada pelas seguintes razões:

1. Para estimar o custo de execução de uma mudança no sistema.

2. Para avaliar se seções críticas do sistema serão afetadas pelas mudanças pretendidas.

3. Para documentar as mudanças realizadas com vistas a favorecer em mudanças futuras.

4. Para entender como os itens a serem mudados impactam a estrutura do *software*.

5. Para determinar quais parte do sistemas estarão sujeitas a realização de testes de regressão depois que as mudanças forem implementadas.

Para Dit et al.[6] a atividade de análise de impacto de mudanças inicializa-se após o processo de LCS. Desta forma, a partir do trecho de código fonte que foi identificado, todo

o restante de código fonte afetado por uma solicitação de mudança no *software* pode ser localizado e mantido. Este trabalho não avança sobre a etapa de análise de impacto de um sistema de *software*, limitando-se a melhorar o processo de localização de características e *bugs*.

2.1.3 Características e *Bugs* de *Software*

Durante a atividade de manutenção e evolução de um sistema, os desenvolvedores realizam a adição de novas funcionalidades, melhoria das já existentes, bem como remoção de *bugs*. Uma funcionalidade é frequentemente descrita através de conceitos oriundos do domínio do problema do *software* [35]. Contudo, nem todas as funcionalidades do *software* podem ser acessadas diretamente pelos usuários e desenvolvedores. Neste sentido, uma característica de *software* descreve uma funcionalidade de sistema observável em tempo de execução pelos usuários e desenvolvedores [9, 6, 33] e tem sido alvo de diversos estudos. Bohnet, Voigt e Doellner[36] afirmam que para realizar uma solicitação de mudança em uma característica os desenvolvedores necessitam identificar quais artefatos a implementam e como estes últimos colaboram para produzir a funcionalidade.

Por outro lado, o conceito de *bug* precisa ser melhor explicado. No cotidiano de um desenvolvedor, *bug* é sinônimo de defeito, erro ou falha [37]. Para Zeller[38], cada item citado pode ser assim definido:

1. Defeito é definido como um código de programa que possua incorreções (*"a bug in the code"*).

2. Erro é definido como a entrada do programa em um estado inconsistente ou incorreto (*"a bug in the state"*).

3. Falha é um comportamento incorreto do programa que seja visível (*"a bug in the behavior"*).

Outra proposta deste trabalho se dedica a localizar segmentos de código com incorreções (defeitos), utilizando e aprimorando técnicas baseadas em no modelo de PLN denominado *Paragraph Vector*.

2.1.4 Localização de Características e *bugs* de *Software*

Características de *software* de um sistema estão diretamente relacionadas à descrição e arquitetura das funcionalidades de um sistema. Contudo, o espalhamento da implementação de uma característica por vários módulos do código fonte [39] bem como a não documentação da rastreabilidade entre os artefatos de *software* durante o processo de desenvolvimento [40] dificultam a tarefa de manutenção. Um processo característico de LCS acontece quando o desenvolvedor, partindo de uma descrição de característica,

necessita recuperar quais partes do código fonte precisam ser alteradas para sua manutenção. Assim, Dit et al.[6] definem a localização de uma característica como a atividade de identificar a posição inicial no código fonte onde determinada funcionalidade está implementada.

Algumas abordagens para LCS baseiam-se em aprendizado de máquina a partir do uso de RNAs [11]. De acordo com a necessidade do desenvolvedor, o elemento a ser mapeado no código fonte pode ser de diversas naturezas como como classes, métodos, pacotes ou mesmo *snippets* de código. Um processo de LCS foi ilustrado na Figura 1, em que métodos do código fonte de um *software* devem ser localizados em resposta à uma descrição de característica. Deseja-se encontrar aqueles métodos mais susceptíveis de conterem a implementação da característica desejada. Neste processo, baseado em um abordagem estática do código fonte, os métodos e as consultas são convertidos para vetores numéricos após processamento pelo algoritmo *Doc2vec* cujo funcionamento é detalhando na Seção 2.3.2. Inicialmente o código fonte passa por um pré-processamento que compreende as seguintes etapas:

1. Separação dos métodos do código fonte com formação de uma coleção de documentos distintos (*corpus*);

2. Remoção de *stop words*, isto é, palavras sem valor semântico agregado como artigos, conjunções, dentre outros. Além disso são removidos números, caracteres especiais, entre outros elementos;

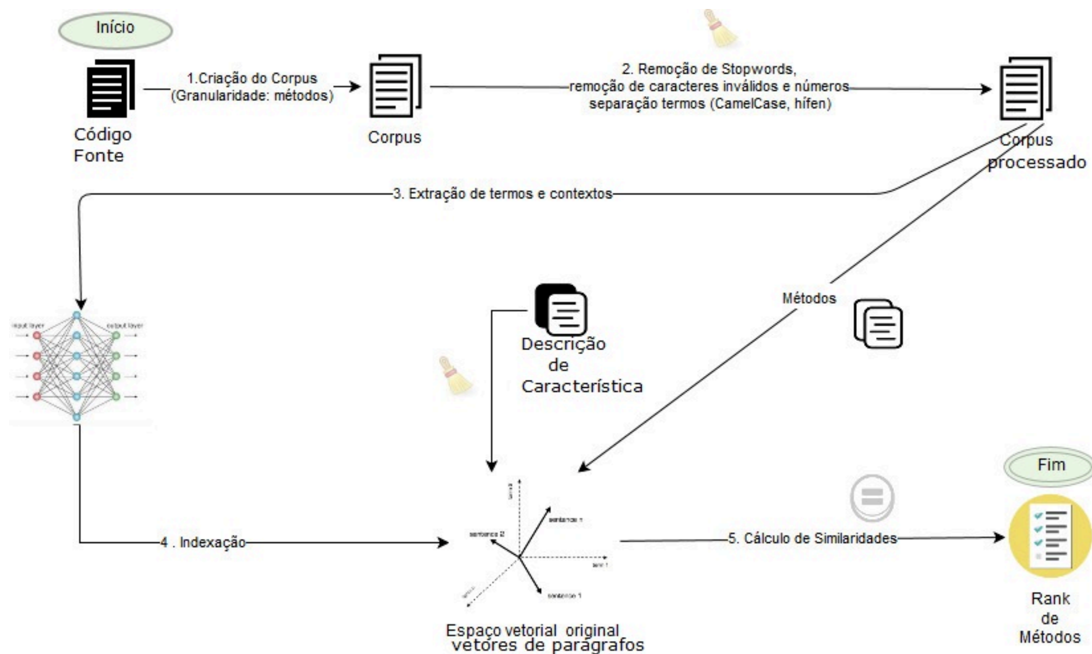
3. Remoção de palavras curtas, conforme critério definido pelo pesquisador e,

4. Separação de termos no padrão *CamelCase*.

Finalizado este processo, passa-se à fase de treinamento do modelo utilizando uma RNA. Este processo resulta na produção de vetores numéricos representativos de cada método em um espaço embarcado (*embedded space or embedded layer*) [41]. Posteriormente, é possível inferir um vetor para cada consulta a partir do modelo gerado. Cada vetor de consulta é comparado com cada vetor de método do código fonte utilizando uma medida de similaridade como por exemplo, o cosseno. O processo finaliza-se com a geração de uma lista ordenada de métodos em função da maior similaridade semântica (maior valor de cosseno) [42].

As técnicas de LCS podem ser classificadas em estáticas, dinâmicas ou híbridas, de acordo com a forma com que a informação é extraída do *software* para seu processamento [43]. Abordagens estáticas baseiam-se somente em informações textuais dos artefatos do *textitsoftware* e suas dependências, principalmente o código fonte, sem que o mesmo seja executado. Binkley e Lawrie[22] exemplificam este processo com base em técnicas de RI que tem sido usadas para explorar elementos de linguagem natural como comentários e identificadores dentro do contexto de LCS. Por outro lado, a forma dinâmica baseia-se na

Figura 1 – LCS usando *Doc2vec*



análise dos rastros de execução do *software* e seus casos de teste para identificação dos itens do código fonte que estejam relacionados às características de interesse do desenvolvedor. Poshyvanyk et al.[14] concordam que a integração das duas abordagens é benéfica para melhorar a performance geral do processo de LCS, definindo a modalidade híbrida.

Por outro lado, em um processo de LBS, os desenvolvedores devem localizar o *bug* no código fonte de um programa procurando identificar as classes (e suas sub-partes) responsáveis pela comportamento não desejável do sistema [44]. A utilização de técnicas automatizadas para esta tarefa tipicamente analisa um relatório de *bug* e o código fonte de um *software* e reportam uma lista de entidades ordenadas por similaridade com o *bug*. De maneira similar à LCS, as técnicas de LBS recebem a mesma classificação [45]. É importante ratificar que a modalidade estática possui a vantagem de não necessitar da execução do código fonte para serem utilizadas e, portanto, foi definida para ser utilizada neste trabalho. Assim, sua utilização torna escalável a atividade de explorar grandes volumes de código fonte provenientes de repositórios de *software* sem ter que se criar cenários de execução bem definidos para o *software*.

O objetivo final da aplicação de técnicas de LCS é identificar que partes do código fonte implementam determinada funcionalidade com o objetivo de melhorar a compreensão do sistema tornando mais segura a inserção de novas mudanças. De maneira similar, técnicas de LBS buscam localizar que partes precisam ser alteradas para correção de *bugs*. Como foi visto, a acurácia destas técnicas tem se mantido baixa requerendo grande esforço e tempo por parte dos desenvolvedores. Nas subseções seguintes as principais técnicas para LCS e LBS serão apresentadas.

2.2 Técnicas para LCS e LBS baseadas em frequência de termos

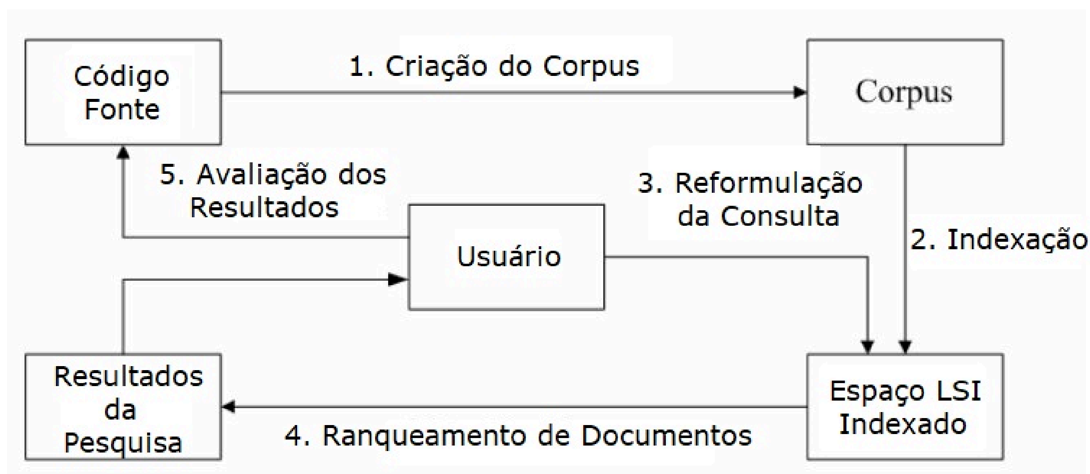
Pesquisas recentes para recuperação da rastreabilidade entre artefatos de *software* tem sido apoiadas por diversas técnicas de RI como *Latent Semantic Indexing* (LSI), de Marcus et al.[9] e *Latent Dirichlet Allocation* (LDA) [10, 15].

2.2.1 *Latent Semantic Indexing* (LSI)

Muitos modelos baseados em Álgebra Linear foram propostos com o intuito de representar e recuperar documentos. O modelo mais tradicional, denominado de Modelo de Espaço Vetorial (do Inglês, Vector Space Model), busca representar documentos de texto como vetores em um espaço algébrico, permitindo computar a similaridade entre os mesmos. No entanto, representar documentos de maneira compacta tornou-se uma necessidade, dada a quantidade de documentos que são produzidos atualmente e o custo computacional de se processar tamanha quantidade de documentos. Desta forma, a redução da dimensionalidade do vetor representativo de um documento representou um avanço na tentativa de se criar um espaço de busca de informações mais compacto. Exemplos de técnicas que implementam a redução de dimensionalidade inclui a Indexação por Semântica Latente (LSI) e sua vertente probabilística (pLSI).

Um processo típico de LCS envolve o *parsing* dos documentos (métodos, classes, etc) do código fonte, a indexação e o ranqueamento dos mesmos em relação à uma consulta (descrição de característica). Este processo foi ilustrado na Figura 2 para a técnica LSI.

Figura 2 – LCS usando LSI



Fonte: Traduzido de Poshyvanyk et al.[14]

A técnica LSI foi proposta por Deerwest em 1990 [46], e tinha como objetivo melhorar a localização de documentos relevantes em um conjunto de textos com base em

termos oriundos de consultas fornecidas pelo usuário. A técnica utiliza o algoritmo "*Singular Value Decomposition*" (SVD). Este algoritmo recebe como entrada uma matriz de termos-documentos e produz como saída outra matriz equivalente, embora com dimensões reduzidas. A matriz de entrada é produzida com base na coocorrência de termos dentro dos documentos. Documentos são assim representados em um espaço n-dimensional, em que n representa um hiperparâmetro fornecido no momento da execução do algoritmo SVD, com valores sugeridos pela literatura variando entre 100 e 300 dimensões. Por outro lado, as consultas nesta técnica são representadas como pseudo-documentos. Estas são inseridas no espaço semântico vetorial, com vistas à obtenção de similaridade com outros documentos já indexados. A similaridade é obtida a partir do cálculo do cosseno entre os vetores representativos de documentos e consulta. Diversos trabalhos utilizaram LSI de maneira efetiva para recuperação de rastreabilidade entre artefatos de *software* ([40]; [9]).

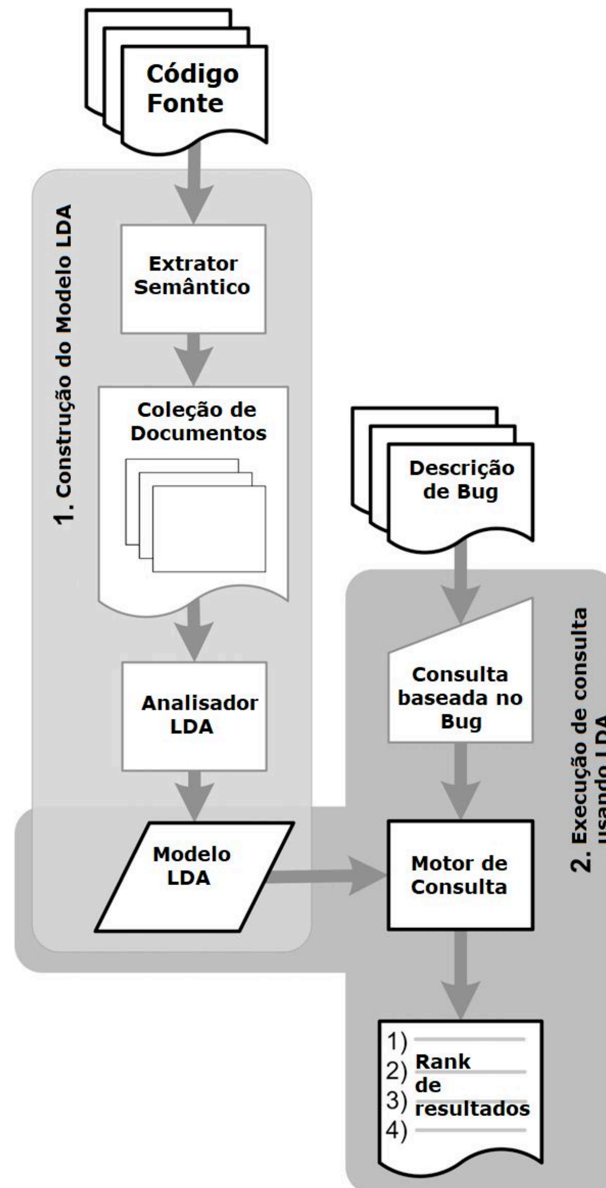
2.2.2 *Latent Dirichlet Allocation* (LDA)

A técnica LDA é um modelo generativo de tópicos baseado em probabilidade utilizado para extrair tópicos que estejam latentes ou ocultos em uma coleção de documentos ([47]; [48]). Esta técnica está baseada em um modelo de aprendizado de máquina não supervisionado. Esta técnica permite modelar cada documento como uma mistura finita sobre o conjunto de tópicos. Cada tópico neste conjunto é a uma distribuição de probabilidade sobre o conjunto de termos que constituem o vocabulário da coleção de documentos. Especificamente, assume-se que K tópicos estão associados com uma coleção e que cada documento expressa estes tópicos em diferentes proporções, isto é, cada documento é uma variada combinação de tópicos.

LDA tem sido usado em um conjunto de experimentos científicos dentro da área de Engenharia de *Software* como localização de *bugs* [10], localização de características [15], categorização de *software* [49], recuperação de rastreabilidade entre artefatos de *software* [50], dentre outros. A Figura 3 ilustra o processo de LBS usando a técnica LDA. O Modelo LDA é criado a partir de uma coleção de documentos, no caso específico são métodos do código fonte. Cada documento/método passa a ser representado como uma distribuição de tópicos presentes no modelo. A consulta, oriunda de uma descrição de *bug* é passada ao mecanismo de pesquisa que gera uma lista de métodos ordenados por similaridade semântica.

Apesar de ter sido muito utilizado em tarefas de manutenção de *software*, a técnica LDA apresenta problemas com performance em grandes *datasets* [31] e heterogeneidade nos tópicos gerados a cada vez que é executado, precisando ser calibrado com os valores adequados para seus respectivos hiperparâmetros [51]. Por fim, durante experimentos envolvendo a LCS, mostrou acurácia inferior em muitos casos [11]. Outros trabalhos também reforçam a hipótese, defendida neste trabalho, de que modelos de aprendizado

Figura 3 – Localização de *Bugs* de Software usando a técnica LDA



Traduzido de Lukins, Kraft e Etzkorn[10]

de máquina, neste caso RNA, representam o estado da arte ([32];[52];[53]) e podem ser evoluídos para obtenção de maior acurácia no processo de LCS.

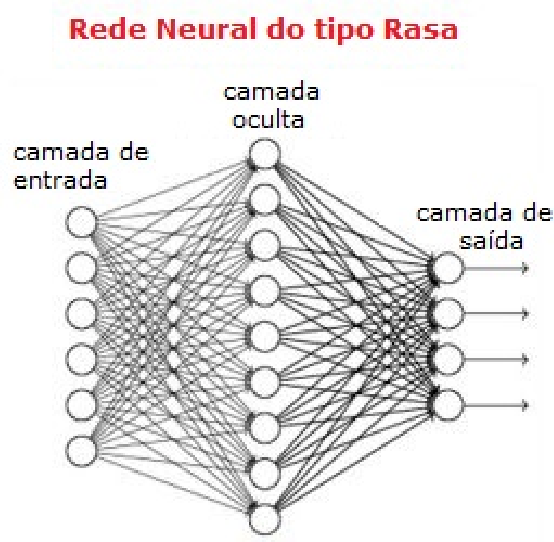
2.3 Técnicas baseadas em Rede Neural Artificial (RNA)

RNAs são modelos de aprendizado de máquina inspirados no sistema nervoso dos seres vivos [54]. O termo "rede neural" surgiu no intuito de se encontrar uma representação matemática para a metáfora relativa ao processamento de informação em sistemas biológicos [55]. Trata-se de um modelo computacional que durante seu treinamento aprende uma função não linear, a partir de dados, para realização de diversas tarefas como classificação,

predição, tomada de decisão, dentre outros.

Ainda do ponto de vista arquitetural pode-se dividir o modelo de RNA em duas categorias: redes profundas (*deep*) e rasas (*shallow*). O modelo raso é composto de duas ou três camadas e qualquer quantidade a mais é considerada profunda [56]. Tal modelo é ilustrado na Figura 4. Para aprender funções complexas estas redes fazem uso de muitos neurônios artificiais na camada oculta. Esta arquitetura é utilizada pelos algoritmos *Word2vec* e *Doc2vec* durante a geração de vetores de palavras e parágrafos respectivamente.

Figura 4 – RNA do tipo rasa

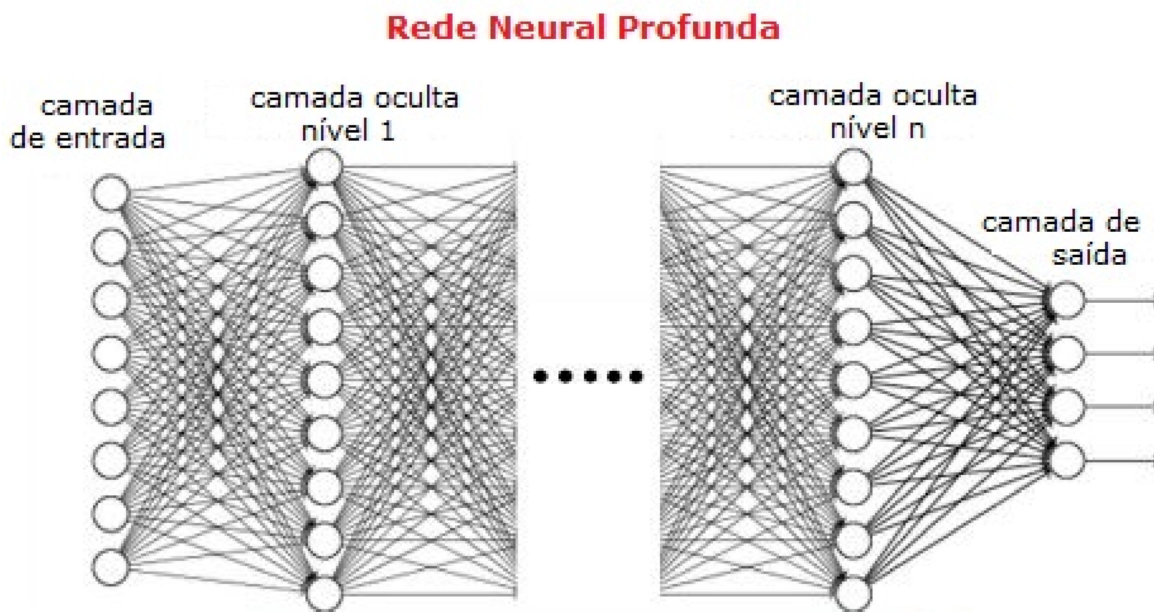


Fonte: Adaptado e traduzido de Lee, Shin e Realff[57].

Como visto, uma RNA profunda possui em sua arquitetura várias camadas de processamento, conforme ilustrado na Figura 5. Elas permitem uma maior capacidade de aprendizagem e generalização [58] e são relevantes em muitos problemas da área de PLN, reconhecimento de discurso, visão computacional, sistemas de recomendação *on-line*, etc [59].

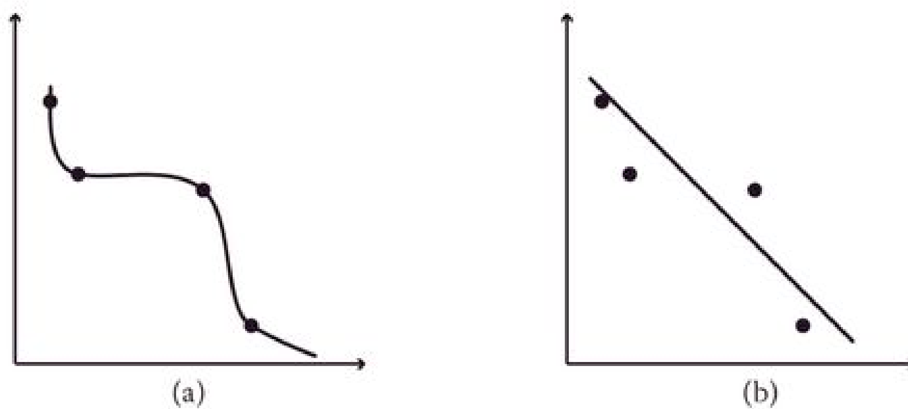
Dois problemas muito comuns em RNA são o *underfitting* e o *overfitting*. Durante o treinamento de uma RNA ocorrerá o mapeamento de dados de entrada para dados de saída. Então, a saída da rede é reiteradas vezes comparadas com os valores ideais providos para treinamento da rede. Quanto mais próximas as respostas da rede, em relação àquelas esperadas para as saídas, sobre os dados de treinamento, maiores as chances de ocorrer o fenômeno de *overfitting*. Isto caracteriza uma perda da capacidade de generalização do modelo. Esta situação é aquela em que modelo descreve exatamente os dados de treinamento conforme ilustrado na Figura 6(a). Por outro lado, quando o modelo possui poucos dados para descrever o conjunto de treinamento aparece o fenômeno denominado *underfitting*.

Figura 5 – RNA do tipo profunda



Fonte: Adaptado e traduzido de Lee, Shin e Realff[57].

Figura 6 – Processos de (a)*overfitting* e (b)*underfitting* em Redes Neurais



Fonte: Lamm e Unger[60].

Recentemente, RNAs têm sido utilizadas para geração da representação de palavras e parágrafos sob a forma de vetores de números reais densos e de baixa dimensionalidade [61]. Para referência a vetores representativos de palavras foi cunhado o termo *Word embedding* e para parágrafos, documentos ou sentenças *Paragraph Vectors*. O modelo *Paragraph Vectors* foi desenvolvido por Le e Mikolov[62]. Uma implementação para este modelo pode ser encontrada no algoritmo *Doc2vec*[3] presente no *framework Gensim*.

Doc2vec é uma extensão do algoritmo *Word2vec*[4], sendo que estes algoritmos serão discutidos nas próximas seções deste capítulo. O algoritmo *Doc2vec* foi aplicado à LCS

[3] https://radimrehurek.com/gensim/models/doc2vec.html
[4] https://radimrehurek.com/gensim/models/word2vec.html

realizada por Corley, Damevski e Kraft[11]. No Capítulo 3, uma extensão deste trabalho é apresentada no sentido de se delinear quais modificações em RNAs, propostas pela comunidade científica, podem ser aplicadas à rede rasa do Algoritmo Doc2vec para melhoria do processo de LCS. Outros modelos relevantes, utilizados na geração de vetores de palavras, estão implementados nos algoritmos fastText[5] e Glove[6]. Trata-se de um modelo desenvolvido por pesquisadores de Stanford. O modelo baseado em Glove considera uma matriz não esparsa de co-ocorrência entre palavras, isto é, sem elementos com valores 0, para produzir um espaço semântico fazendo uso dos recursos de janelas (contexto de palavras), bem como fatorização global de matrizes [63]. Por outro lado, o mecanismo de criação de vetores de palavras no fastText foi desenvolvido por pesquisadores do Facebook. Este considera utilizar n-gramas menores que as palavras em uma tentativa de aproximar vetores de palavras não contidas no dicionário. Um n-grama é uma sequência contígua de n itens de uma determinada amostra de texto ou fala. Os itens podem ser fonemas, sílabas, letras, palavras ou pares de bases de acordo com a aplicação. Por exemplo, supondo que em um código fonte exista a palavra *java*, então, são consideradas sub-palavras e terão seus vetores também gerados "j", "ja", "av", "va", "jav" e "ava" [30]. Os vetores produzidos a partir destes algoritmos prestam-se ao tratamento de texto em um contexto computacional. A escolha do algoritmo *Doc2vec* para este trabalho está baseada na proeminente acurácia aferida em um conjunto de estudos científicos de PLN [64, 65].

2.3.1 Vetores de Palavras

Vetores de palavras (do inglês *Word embeddings*) são representações de palavras sob a forma de vetores numéricos com baixa dimensionalidade. Como foi visto anteriormente, é possível gerar vetores numéricos representativos de palavras com base em técnicas baseadas em contagem e frequência de palavras. Na literatura diversas técnicas estão baseadas em modelos de extração de vetores de palavras ([66];[67]; [68]; [30]). Nesta seção o interesse se volta àquelas técnicas cujos vetores são gerados a partir da utilização de modelos baseados em RNAs, como exposto por [31].

Como visto, o algoritmo *Word2vec* permite a geração de vetores de palavras tendo como entrada documentos de texto. Vetores de palavras representam a base para diversas atividades no campo do PLN. Um destes modelos, desenvolvido e implementado por Mikolov et al.[31] como um algoritmo *multithread* escrito em C++, foi denominado *Word2vec*[7]. A Figura 7 ilustra um vetor numérico obtido a partir de um modelo gerado pelo algoritmo *Doc2vec* neste trabalho que é derivado do *Word2vec.* O vetor em questão foi gerado para representar a palavra "return" que estava presente no código fonte do software *Eclipse Framerwork Communication.* A partir da análise do contexto das outras

[5] https://fasttext.cc/docs/en/supervised-tutorial.html
[6] https://nlp.stanford.edu/projects/glove
[7] https://code.google.com/p/word2vec/

palavras que circundavam a palavra "return" e da utilização de uma rede neural do tipo rasa foi gerada uma representação numérica para cada palavra do código fonte. A partir destes vetores podemos realizar várias tarefas como analogias entre palavras, clusterização de termos e geração de vetores representativos de documentos.

Figura 7 – Vetor de palavra para o termo "*return*" gerada pelo *Doc2vec* sobre o corpus do *software Eclipse Framerwork Communication*

```
[ 5.02063613e-03 -3.53393541e-03 -5.06016659e-04 -3.66158341e-03
 -7.94113148e-03  7.62236165e-03 -7.32919248e-03 -6.03418611e-03
  8.93782079e-03 -3.13158263e-03 -4.63420479e-03  6.70957525e-05
 -9.78110358e-03  2.40295543e-03 -2.62816413e-03  8.64844397e-03
  9.84590407e-03  5.68793248e-03 -1.78309891e-03  5.97400358e-03
 -6.14077377e-04 -2.35955440e-03 -4.39928751e-03  3.75728094e-04
  5.06142434e-03 -5.40464325e-03  6.06373884e-03  4.71173692e-03
 -8.45219195e-03 -4.21121437e-03 -9.06209275e-03  8.35464709e-03
 -2.21151611e-04  9.40386951e-03 -4.78478195e-03 -5.84691297e-03
 -3.75199452e-04  5.57582779e-03  1.87078526e-03  9.62991209e-04
  1.04630024e-04 -7.32536567e-03  1.47068128e-03 -6.96029374e-03
 -7.35334959e-03  7.06776744e-03 -4.18115035e-03 -5.51255839e-03
 -8.39496148e-04 -3.09504196e-03]
```

Neste trabalho, os documentos utilizados para alimentar o algoritmo são os métodos e classes de sistemas orientados a objetos. Para se compreender o funcionamento do algoritmo *Word2vec* torna-se necessário a compreensão do conceito de janela textual. Dada uma palavra alvo, isto é, aquela que se deseja representar vetorialmente, definiu-se como contexto as palavras que a circundam dentro de uma janela nos documentos providos para aprendizado.

Por outro lado, pode-se também partir das palavras do contexto e tentar mapeá-las para a palavra alvo. Estas duas abordagens serão abordadas mais à frente. Desta forma utilizou-se estes mapeamentos, no algoritmo *Word2vec* que usa uma rede neural do tipo rasa (*shallow network*), para geração de vetores. Para entender melhor o conceito de janela, tome-se como exemplo um comentário encontrado dentro de uma classe do *software* Eclipse Framework Communication: "return the remote service reference". Como a palavra "the" é uma *stop word*, então, a frase ficará *"return remote service reference"*.

A Tabela 1 ilustra quais outras palavras fazem parte da janela de cada palavra alvo se for definida uma janela de tamanho 2. Assim, caso seja definida uma janela de tamanho três, deve-se observar as três palavras à esquerda e direita ao redor da palavra alvo para compor o contexto. De forma semelhante outros tamanhos de janela podem ser definidos e processados.

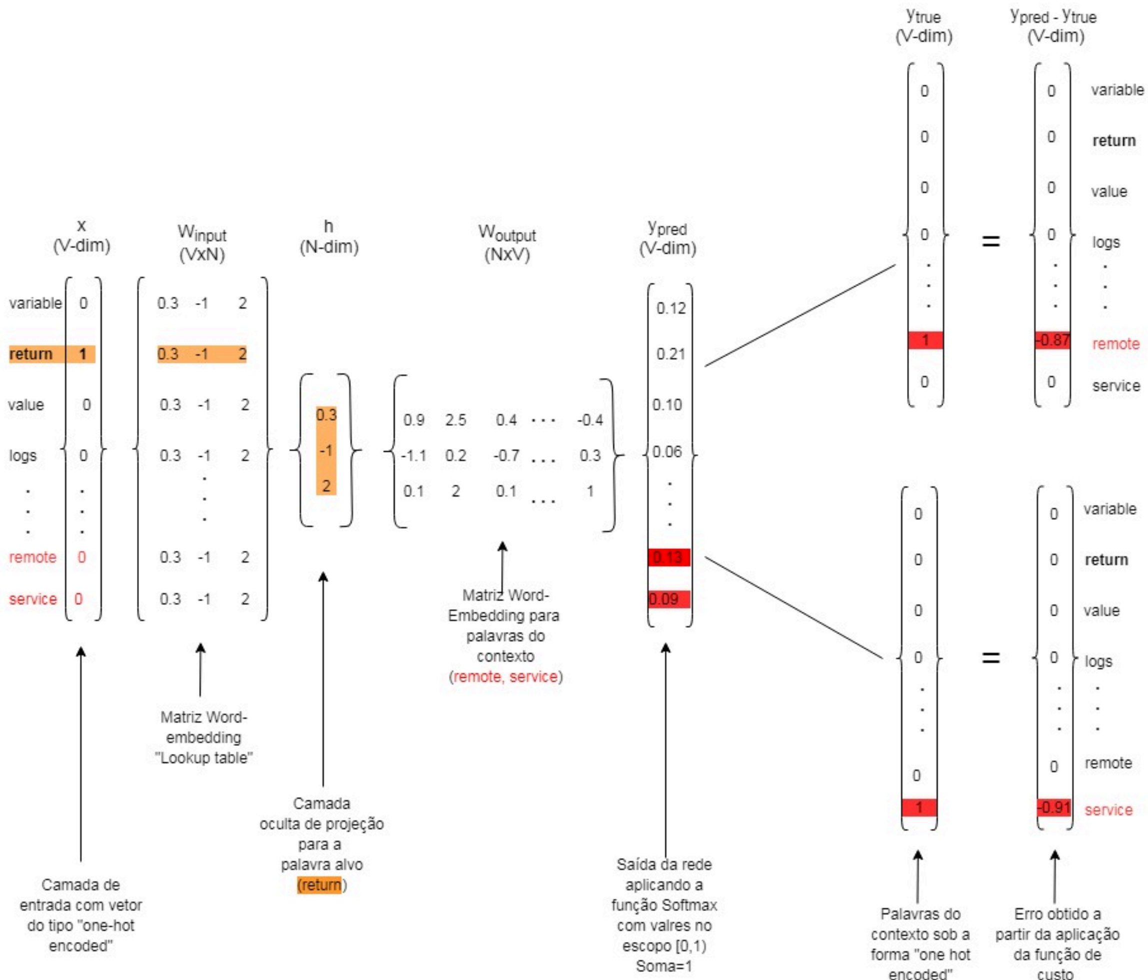
O tamanho da janela é um hiperparâmetro do algoritmo *Word2vec* que pode ser calibrado para diferentes *datasets*, e, portanto, é variável podendo englobar 1, 2, 3 ou mais palavras ao redor da palavra alvo.

Tabela 1 – Exemplificando o uso de uma janela de tamanho 2 no *Word2vec*

Janela Esquerda (Contexto)	Palavra alvo	Janela Direita (Contexto)
-	return	remote service
return	remote	service reference
return remote	service	reference
remote service	reference	-

Ao contrário de técnicas baseadas em frequência de termos, *Word2vec* funciona com base em predição. Assim deseja-se predizer uma palavra alvo dado um contexto, isto é, $\Pr(Walvo \| Wcontexto)$. Tem-se também a opção de predizer um contexto (palavras que circundam a palavra alvo) dado uma palavra alvo, isto é, $\Pr(Wcontexto \mid Walvo)$.

Figura 8 – Exemplo utilizando o treinamento no algoritmo *Word2vec*



Fonte: Adaptação de Eric[69]

A Figura 8 ilustra bem o processo de geração de vetores. Neste caso, tomando-se como referencial a palavra alvo, deseja-se gerar os vetores das palavras que fazem parte do contexto desta referência. Como exemplo a palavra alvo "return" que tem como contexto as palavras "remote" e "service". Para que sejam gerados os vetores representativos das

palavras do contexto deve-se seguir os seguintes passos:

1. Representar a palavra *"return"* sobre a forma de *"one hot encoded"*[8]. De maneira simples, gera-se um vetor esparso em que cada posição corresponde a um item do vocabulário do *corpus* analisado. Para a geração de um vetor *"one hot encoded"*, uma dada palavra do vocabulário terá uma posição do vetor associada e receberá o valor 1. Neste caso a posição referente à palavra *"return"* recebeu o valor 1. As demais palavras receberão valor 0.

2. Treinar a RNA que é do tipo rasa. Observe que só existe uma camada oculta. No caso específico, o tamanho de vetor, escolhido empiricamente, para representação das palavras é 3, vide o número de neurônios da cada oculta. A literatura relata que vetores com 100, 200 ou 300 dimensões são suficientes para obter-se bons resultados.

3. Comparar a saída da rede presente na camada $\mathrm{y_{pred}}$ com os vetores *"one hot encoded"* das palavras alvo.

4. Aplicar a função de custo e aplicar o algoritmo *backpropagation* para atualizar os pesos da rede.

5. Atualizar os pesos das matrizes $\mathrm{W_{input}}$ e $\mathrm{W_{output}}$.

6. Obter os vetores das palavras do contexto ao fim do treinamento a partir da matriz $\mathrm{W_{output}}$.

7. Utilizar o modelo treinado para fazer inferência de palavras e parágrafos.

Word2vec é um dos principais algoritmos utilizados para geração de vetores de palavras e seu funcionamento é similar a um *autoencoder*. Assim, cada palavra é codificada como um vetor, mas ao invés de treinar objetivando obter uma palavra idêntica à palavra passada para o algoritmo, como é feito por exemplo em máquinas restritas de Boltzmann[9], o algoritmo *Word2vec* treina palavras contra outras palavras (o contexto) que estão em uma janela de distância determinada experimentalmente. A entrada da rede é uma palavra alvo e a saída esperada são as palavras do contexto como explicado na discussão da Figura 8. As janelas podem ser obtidas a partir da observação do *corpus* sob o qual o algoritmo irá gerar um modelo de predição.

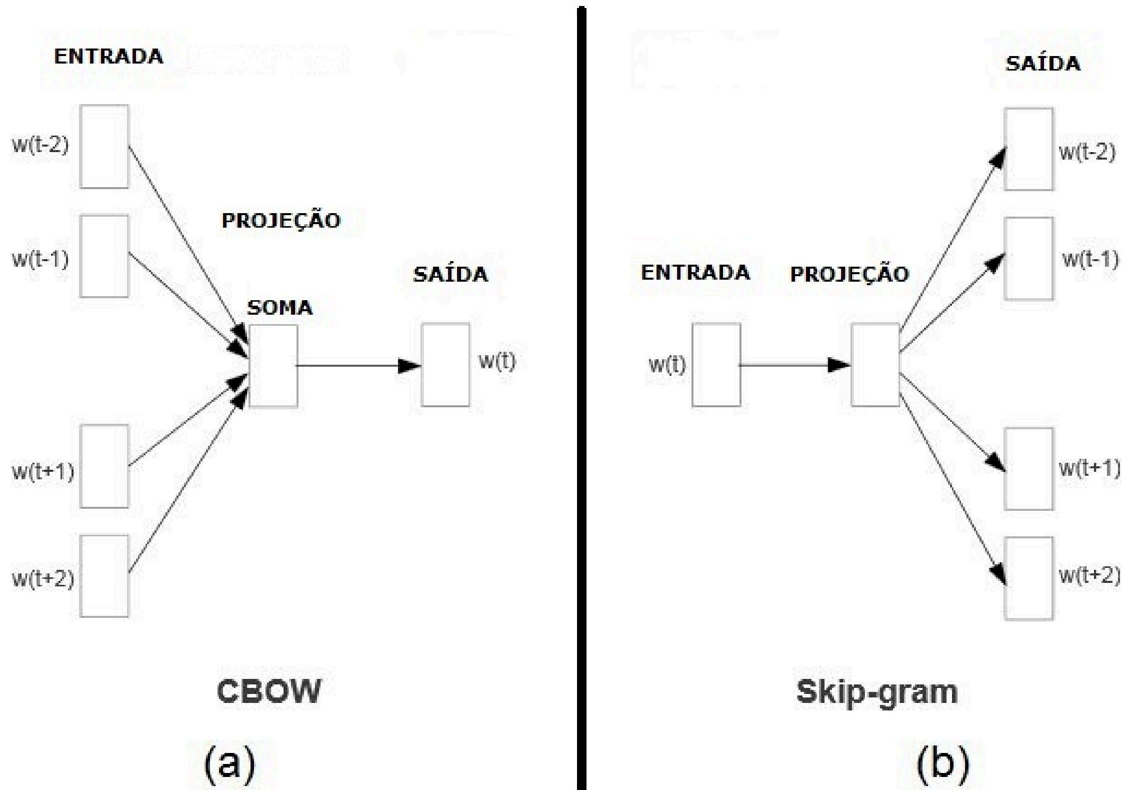
Dois modelos foram desenvolvidos para geração dos vetores de palavras: *Continuous bag-of-words (CBOW)* e *Skip-gram*. O modelo CBOW assemelha-se a uma RNA *feedforwad* em que a camada oculta não linear é removida e a camada de projeção é compartilhada por todas as palavras [31]. Observa-se pela Figura 9(a) que no modelo CBOW, as palavras do contexto são usadas para predição do vetor da palavra alvo enquanto na

[8] https://towardsdatascience.com/categorical-encoding-using-label-encoding-and-one-hot-encoder-911ef77fb5bd

[9] https://matheusfacure.github.io/2017/07/30/RBM/

Figura 9(b), que retrata a arquitetura *skip-gram*, as palavras do contexto tem seus vetores numéricos gerados a partir da palavra alvo.

Figura 9 – Arquiteturas CBOW(a) e Skip-gram(b)



Fonte: Traduzido de Mikolov et al.[31].

Desta forma, este trabalho utiliza vetores gerados a partir de palavras e seus contextos obtidos a partir da informação textual presente em código fonte de *software*. Dentro do contexto de LCS, consultas formuladas pelo desenvolvedor são convertidas para vetores numéricos a partir da concatenação de vetores de palavras individuais. Já para representação de métodos do código fonte foi utilizado o algoritmo *Doc2vec* proveniente do Modelo *Paragraph Vectors*, assunto da próxima subseção. Mais detalhes sobre a geração e utilização de vetores de palavras e parágrafos podem ser encontrados no Capítulo 3.

2.3.2 Vetores de Parágrafos

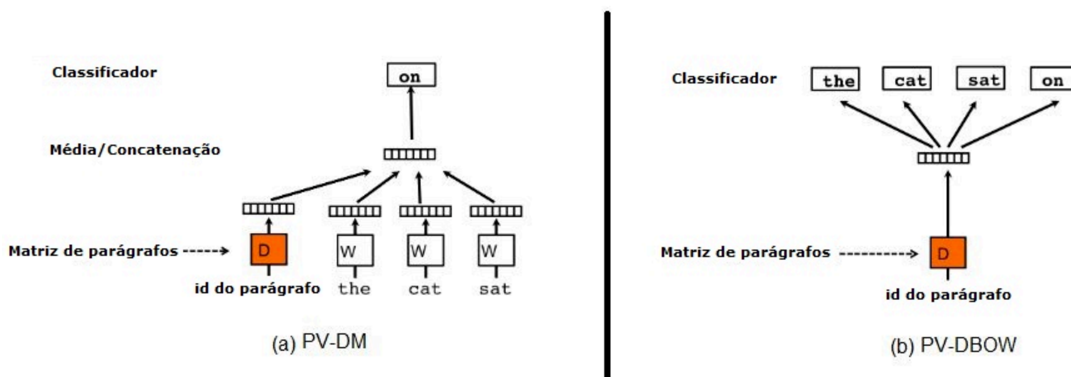
Com base no algoritmo *Word2vec*, Mikolov et al.[13] introduziram em 2014 um novo modelo denominado *Vetores de Paragráfo* (PV[10]) para aprendizado de vetores representativos de parágrafos, documentos ou sentenças. Trata-se de uma técnica inspirada no aprendizado da representação de parágrafos sob a forma de vetores numéricos gerados a partir do uso de RNAs [70]. Neste modelo os vetores são gerados a partir de um processo de treinamento supervisionado em uma RNA do tipo rasa. Duas vantagens são apontadas

[10] Do inglês, *Paragraph Vectors*

por Mikolov et al.[13] para este modelo em relação às técnicas *bag of word*: preservação da semântica e ordem das palavras. Parágrafos de diferentes tamanhos podem ser transformados pela RNA em vetores de tamanho fixo [71].

Segundo Dong e Liu[72], PV foi apresentado sob a forma de dois modelos: *Paragraph Vector Distributed Memory* (PV-DM) e *Paragraph Vector Bag of Words model* (PV-DBOW). O modelo PV-DM é uma derivação do modelo CBOW e cada parágrafo recebe um identificador único que é adicionado durante o treinamento do modelo ao contexto das palavras como se fosse uma outra palavra, conforme ilustrado na Figura 10(a). Portanto, ao se treinar os vetores representativos de palavras, o vetor de parágrafo (paragraph_id) também será treinado. Ao fim do treinamento este vetor armazenará uma representação numérica do parágrafo, também chamada de memória distribuída. Trata-se da inclusão de uma pseudo palavra na janela ou contexto de cada palavra constituinte do parágrafo. O pseudo vetor **p**, gerado desta forma, representará o parágrafo. Por outro lado, o modelo PV-DBOW baseia-se na adaptação do modelo Skip-Gram, conforme ilustrado na Figura 10(b). Neste caso, o vetor **p** representativo do parágrafo será a entrada da RNA. Espera-se como saída obter os vetores de um conjunto de palavras, escolhidas randomicamente dentre aquelas pertencentes ao parágrafo. Representações vetoriais obtidas através de PVs tem se mostrado superiores, em um conjunto de tarefas, àquelas obtidas através de abordagens do tipo *bag of word* [72].

Figura 10 – Arquiteturas PV-DM(a) e PV-DBOW(b)



Fonte: Traduzido de Mikolov et al.[13].

Conforme dito anteriormente, este trabalho utiliza como base o modelo baseado em vetores de parágrafo para representar, sob a forma de vetores numéricos, os métodos do código fonte dos *software* orientado a objetos utilizados. Umas da principais implementações deste modelo é o algoritmo *Doc2vec*[11], encontrado no *framework* Gensim, escrito na linguagem de programação Python. Um exemplo básico de utilização do algoritmo *Doc2vec* na linguagem Python pode ser encontrado no Apêndice C.

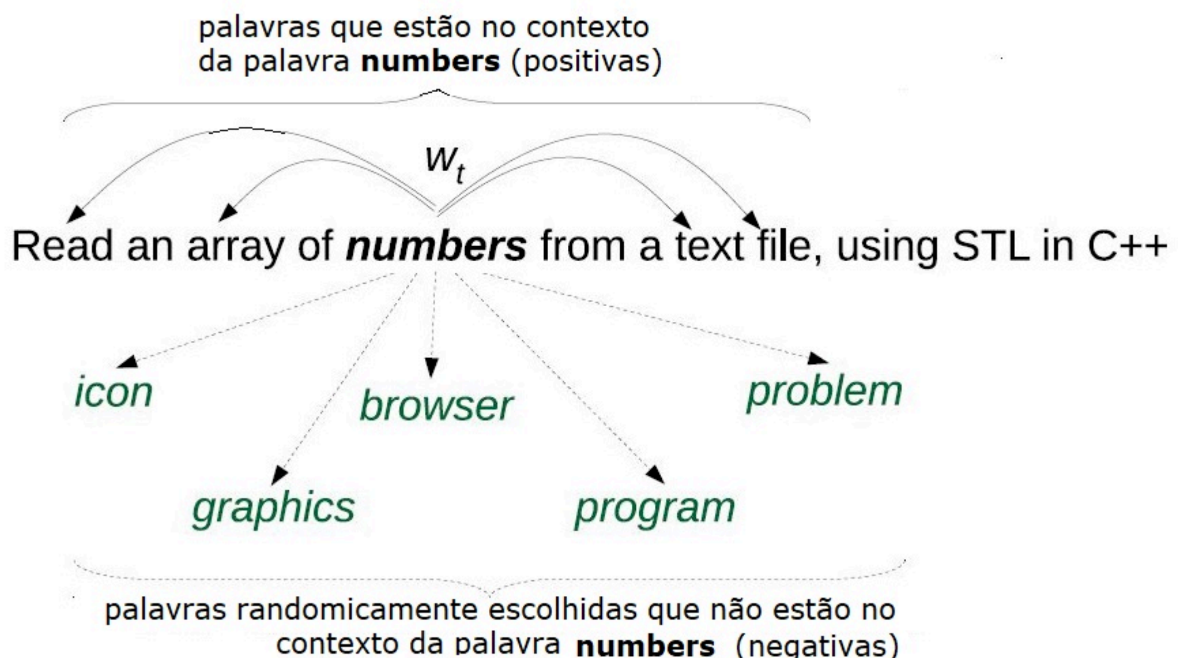
[11] https://radimrehurek.com/gensim/models/doc2vec.html

2.3.3 Algoritmos usados pelo algoritmo *Doc2vec*

Uma rede neural é operacionalizada a partir de um conjunto principal de funções ou algoritmos, dentre os quais se destacam: a função de ativação, a função de custo e o algoritmo de otimização da rede [73]. Os algoritmos *Word2vec* e *Doc2vec* tipicamente usam redes neurais rasas em que a função estocástica gradiente descente e o método *backpropagation* são usados para o aprendizado [74]. As funções não lineares de saída do modelo possuem, dentre suas representantes mais importantes, a função *hierarquical softmax* e *negative sampling* [62, 13]. Em todos os capítulos deste trabalho *hierarquical softmax* e *negative sampling* são referenciados respectivamente como HSM e NGS.

Negative sampling é uma aproximação da função *Noise Constrastive Estimation* (NCE). Segundo este modelo dados de boa qualidade devem ser diferenciados de ruídos a partir de regressão logística. O objetivo de se encontrar representações de alta qualidade para palavras se reduz agora em diferenciar dados válidos (conjunto de palavras que realmente ocorrem dentro de um contexto) de ruídos (conjunto de pares de palavras que não representa pares válidos encontrados em um contexto).

Figura 11 – Exemplo de palavras positivas e negativas usadas na modalidade *Negative Sampling*



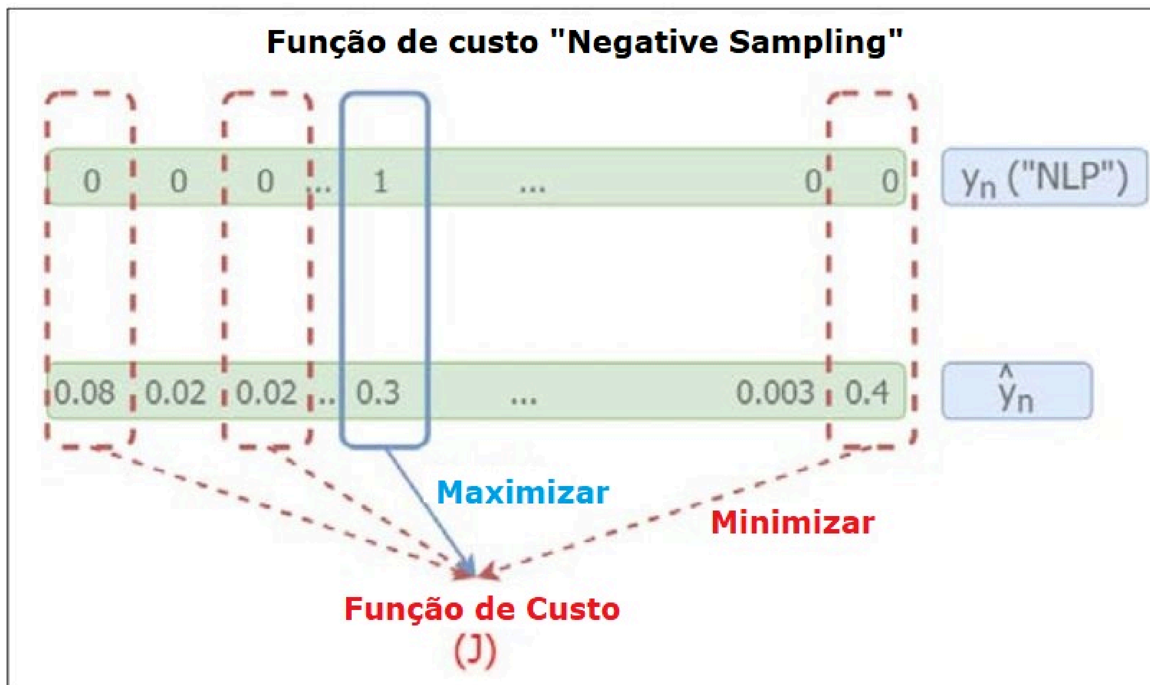
Fonte: Traduzido de Ye et al.[12].

A Figura 11 traz um exemplo de palavras positivas e negativas utilizadas durante o treinamento e aplicação da função de custo *Negative Sampling*. Tendo como referencial a palavra *numbers*, figuram como palavras do contexto ou positivas aquelas que a circundam dentro de um intervalo de palavras previamente definido. Palavras negativas são aquelas que não figuram neste contexto e seu número é definido empiricamente e objeto de estudo

deste trabalho quanto à influência sobre a acurácia na localização de características e *bugs*. Durante o treinamento usando *negative sampling* tanto as palavras negativas quanto as positivas tem seu peso atualizado na matriz de saída, isto é, a matriz de pesos que fica entre a camada oculta e a camada de saída da rede. As outras palavras do vocabulário não tem seus pesos atualizados o que otimiza o treinamento da rede e mesmo assim mantém boa acurácia ao fim do processo.

A Figura 12 exemplifica a forma de funcionamento da função de custo *Negative Sampling*. Assim, os dados verdadeiros foram assinalados com a linha azul contínua. Em vermelho e pontilhado aparece o ruído, isto é, conjunto de pares de palavras que não ocorrem no mesmo contexto. O objetivo desta estratégia é maximizar a função de custo para os dados verdadeiros e minimizar para aqueles que não acontecem na realidade, isto é, onde as palavras não apareçam no mesmo contexto [41]. Pode-se ainda ver que para os dados "negativos" a saída esperada é 0 (y_n=0), enquanto para as palavras que aparecem no contexto deseja-se que esta seja igual a 1 (y_n=1).

Figura 12 – O processo de operação da função de custo *Negative Sampling*



Fonte: Traduzido de Ganegedara[41].

A função de custo *hierarchical softmax* tem o mesmo objetivo da função *negative sampling* que é aproximar o *softmax* sem ter que calcular a função de ativação para todas as palavras do vocabulário. A diferença consiste no fato que *hierarchical softmax* só considera dados válidos, isto é, conjunto de pares de palavras que coocorrem no mesmo contexto. Além disso possui seu funcionamento baseado em uma árvore binária.

No algoritmo *backpropagation*, utilizado pelo *Doc2vec* [75], os pesos sinápticos são

atualizados seguindo a seguinte equação:

$$\theta^t = \theta^{t-1} - \epsilon_t \frac{\partial L}{\partial \theta} \tag{1}$$

onde θ representa os pesos da rede neural, L é a função de custo, ϵ_t é a taxa de aprendizado da rede e t é o instante de tempo [72].

A taxa de aprendizado determina o tamanho dos ajustes dos pesos feitos em cada época (iteração) e, portanto, influencia a taxa de convergência da rede neural.

2.4 Medidas de Avaliação da Acurácia em localização de Características e *Bugs de Software*

Nesta seção são apresentadas as métricas que comumente são utilizadas para avaliação de resultados em experimentos de LCS e LBS.

2.4.1 Medidas para cálculo de similaridade entre vetores

Várias medidas foram propostas pelos pesquisadores para para calcular a distância entre vetores em contextos relacionados a RI. Dentre as principais podem ser citadas o cosseno, a correlação de *Pearson* e a correlação de *Spearman* [76]. O cosseno é uma medida comumente utilizada para cálculo de similaridade entre consultas e documentos em experimentos de RI e aprendizado de máquina [77, 78]. Em Engenharia de *Software* também tem sido utilizado com frequência [79, 11].

Dadas duas palavras w_t e w_u, matematicamente, o cosseno entre os vetores numéricos (vetores de palavras) pode ser expresso da seguinte forma [32]:

$$sim(w_t, w_u) = cos(w_t, w_u) = \frac{{w_t}^T w_u}{|| \; w_t \; || \;\; || \; w_u \; ||} \tag{2}$$

O cálculo para similaridade entre documentos é idêntico, ressaltando que neste caso deve se usar o vetor representativo de cada documento. Desta forma, optou-se neste trabalho por estratégia semelhante ao se utilizar o cosseno como medida de similaridade para ranqueamento dos métodos do código fonte em relação às consultas submetidas ao motor de busca construído com base no algoritmo *Doc2vec*. O uso do cosseno como medida de similaridade é observado no trabalho apresentado por Mikolov et al.[13] onde o algoritmo *Doc2vec* foi definido.

2.4.2 *Mean Reciprocal Rank* - MRR

Neste trabalho, a principal métrica utilizada para avaliação da performance das abordagens propostas foi a MRR. Tal decisão se justifica em função do processo de localização

de características se basear na identificação da posição inicial do código fonte que implementa determinada funcionalidade de *software* como enfatiza [6]. Esta métrica foi utilizada em relevantes experimentos de LCS [11, 14]. Assim, dado um conjunto de consultas e seus respectivos resultados, a métrica MRR calcula a média harmônica dos *rankings* obtidos em para um conjunto de consultas. Em uma situação real pode-se aplicar tal métrica para se ter uma visão da performance geral de um mecanismo de RI que processa um dado conjunto de consultas. O cálculo da métrica MRR representa a média da somatória de um conjunto de consultas cujos operandos são formados pelo inverso da posição ordinal do primeiro elemento relevante em cada consulta. Matematicamente, a métrica MRR pode ser expressa conforme apresentado na Equação 3.

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{e_i} \tag{3}$$

Onde Q é o número de consultas e e_i é a posição ordinal do primeiro elemento relevante recuperado para cada consulta Q_i.

Outras medidas de performance de técnicas de localização de características são expostas nas subseções seguintes e são aplicadas neste trabalho conforme o interesse do estudo.

2.4.3 Mean Average Precision - MAP

A métrica MAP é calculada para um conjunto de consultas como a média obtida a partir da precisão média de cada consulta [80]. Segundo Saha et al.[81], esta métrica leva em consideração todos os arquivos relevantes a serem localizados com suas respectivas posições no *rank*. É uma medida que enfatiza a revocação sobre a precisão. Por esta razão torna-se relevante em situações em que os usuários do sistema de busca desejam analisar uma grande quantidade de itens retornados pelo motor de busca. Matematicamente, a métrica MAP pode ser expressa conforme apresentado na Equação 4.

$$MAP = \sum_{i=1}^{|Q|} \frac{AveP(q_i)}{|Q|} \tag{4}$$

Onde Q é o número de consultas e $AveP(q_i)$ é a precisão média em cada consulta Q_i.

No estudo retratado por Poshyvanyk et al.[14], os autores não utilizam medidas tradicionais de RI como precisão e revocação para a tarefa de LCS. Eles justificam que as técnicas de LCS poderão associar *scores* para todos os métodos de um *software* e por consequência sempre obter uma revocação igual a 1.0. Isto é problemático nas situações em que não existe um limite de eficiência estabelecido. Os autores acabam por utilizar o *rank* do primeiro método relevante como medida de avaliação do processo de LCS, remetendo-nos ao MRR que foi citado anteriormente. Assim calcular a precisão para todos os itens

relevantes pode não ser tão efetivo no trabalho de um desenvolvedor que necessite realizar uma tarefa de LCS. Como definido anteriormente, para Dit et al.[6] a atividade de LCS compreende o esforço de se encontrar a posição inicial onde determinada funcionalidade está localizada no código fonte. Ainda segundo os autores, o ciclo completo de uma atividade de alteração de *software* é fechado com a Análise de Impacto, que foi definida anteriormente neste capítulo.

2.5 Resumo do Capítulo

Neste capítulo foram apresentados os elementos teóricos necessários à compreensão das técnicas e modelos utilizados neste trabalho. Desta forma foram apresentadas as técnicas que comumente tem sido utilizadas na localização de características e *bugs* de *software*. As principais técnicas apresentadas foram:

1. LSI - *Latent Semantic Indexing*;
2. LDA - *Latent Dirichlet Allocation*;
3. Word2vec;
4. Doc2Vec .

As técnicas Word2vec e Doc2vec representam o estado da arte para a LCS e LBS e serão abordados em experimentos empíricos nos próximos capítulos.

Capítulo 3

Um estudo empírico para localização de Características de Software usando Vetores de Parágrafo aprimorados

Com o objetivo de melhorar a acurácia em processos de localização de características, diversos trabalhos empregam novas técnicas ou propõem a evolução das já existentes. O estado da arte para a tarefa de localizar características implementadas no código fonte apoia-se na utilização de "*Paragraph Vectors*" [11]. Trata-se de um modelo cuja implementação permite a produção de vetores numéricos capazes de representar parágrafos que são usados em tarefas de Processamento de Linguagem Natural (PLN) e Recuperação de Informação (RI). Na prática, o algoritmo *Doc2vec* representa a implementação mais utilizado do modelo *"Paragraph Vectors"*. Assim, o estudo empírico descrito neste capítulo propôs a melhoria da qualidade dos vetores numéricos gerados, a partir da utilização do algoritmo *Doc2vec*, para representação de métodos do código fonte (parágrafos) e consultas (descrições de características). O contexto abordado refere-se à localização de características de *software* (LCS) a partir da utilização de Redes neurais artificiais (RNAs). Desejava-se com isso um incremento na acurácia do processo de LCS.

3.1 Referencial Teórico

Esta seção se dedica a relatar conceitos específicos não abordados no Capítulo 2, mas que se mostraram relevantes para o desenvolvimento deste capítulo.

Muitas alterações tem sido propostas para melhoria da performance em RNAs. Estratégias clássicas como a escolha das características de entrada da RNA [82] bem como a parada precoce do processo de treinamento *(early stopping)* [83] podem ser citadas. Neste sentido, a partir da análise da performance de classificadores, Villiers e Barnard[84] afirmam que a correta escolha das características que compõem a entrada da rede, a metodologia de treinamento usada e a topologia escolhida são fatores que concorrem para a

descoberta do melhor projeto de uso deste modelo de aprendizado de máquina. Han et al.[85] discutem o fato de redes neurais estarem se tornando o estado da arte em diversas áreas do conhecimento como o campo da visão computacional, o processamento de linguagem natural e reconhecimento de discurso. Entretanto, os autores deixam claro que a operacionalização de tais tarefas em uma rede neural, notoriamente redes profundas, demanda considerável quantidade de armazenamento, alta velocidade do barramento de memória e demais recursos computacionais. Para contornar esta situação eles apresentam um método em que são realizadas podas de arestas da rede com posterior afinamento do pesos sinápticos da rede, visando manter a acurácia do modelo. Em tarefas de PLN, o considerável custo computacional apresentado para treinamento de uma RNA profunda pode ser amenizado a partir da utilização do algoritmo *Doc2vec*. Este algoritmo representa uma alternativa em função de utilizar uma RNA rasa para cumprir sua tarefa.

O algoritmo *Doc2vec* é uma extensão do algoritmo *Word2vec* e, como tal, pode se beneficiar de melhorias promovidas neste último. *Word2vec* depende de uma série de hiperparâmetros que podem ser melhor calibrados para utilização em tarefas de PLN como busca da similaridade entre palavras e detecção de analogias [86]. Consequentemente, *Doc2vec* também mantém esta dependência dos hiperparâmetros. Caselles-Dupré, Lesaint e Royo-Letelier[86] levantam a questão que tais parâmetros merecem ser melhor discutidos para efetiva utilização do *Word2vec* em atividades de recomendação.

Os parâmetros levantados por eles estão ligados aos seguintes aspectos: a função de custo da RNA denominada (*Negative Sampling*); o número de épocas de treinamento; e o tamanho da janela usado para captura dos contextos das palavras. Os autores apontam que o conjunto de valores ótimos para tais hiperparâmetros revela estreita relação com os dados e o tipo de tarefa a ser executada. Neste estudo foi desenvolvida uma abordagem denominada NGS para avaliar a calibração adequada da função de custo *Negative Sampling* no intuito de aumentar a acurácia do processo de LCS.

Apesar de existirem discussões sobre a melhor forma de se utilizar o algoritmo *Word2vec* em tarefas de PLN, esta mesma constatação não está clara para o algoritmo *Doc2vec* em tarefas de manutenção de *software*. Desta forma, permanece aberta a questão de pesquisa que visa delinear qual a melhor calibração para os hiperparâmetros ou alterações estruturais da RNA utilizada pelo algoritmo *Doc2vec*. Espera-se que as modificações sugeridas neste estudo propiciem ganhos reais na acurácia em processos de localização de características de *software*.

Este capítulo apresenta uma abordagem baseada no uso de uma taxa de aprendizado cíclica (LRT). Durante o treinamento da RNA utilizada pelo algoritmo *Doc2vec* aplica-se esta estratégia. Assim, em intervalos regulares, entre épocas de treinamento, chamou-se uma *callback* de alteração da taxa de aprendizado. Uma *callback* equivale a um conjunto de funções aplicada em certos estágios do procedimento de treinamento com a finalidade de visualizar estatísticas e estados internos do modelo em treinamento [87]. Neste caso,

seu objetivo foi alterar a taxa de treinamento da RNA.

3.2 Planejamento Experimental

O objetivo do estudo experimental descrito neste capítulo é descobrir empiricamente quais pontos de evolução da teoria e arquitetura de RNAs podem ser aplicados à rede do tipo rasa (*shallow*) usada pelo algoritmo *Doc2vec*. Em um primeiro momento, propõem-se a avaliação de valores discretos que melhor calibrem a função de custo *Negative Sampling* que é utilizada como padrão pelo algoritmo *Doc2vec*. Posteriormente, o uso de uma política para adaptação da taxa de aprendizado da RNA é explorada. Deseja-se saber quais configurações dentre as sugeridas impactam positivamente a aplicação do algoritmo *Doc2vec* em tarefas de localização de características. Por fim uma combinação das abordagens sugeridas foi exposta avaliando-se uma utilização híbrida das abordagens propostas. As próximas seções detalham o arcabouço experimental utilizado neste estudo.

3.3 Aspectos Técnicos do Experimento

Neste experimento, os métodos do código fonte representam os parágrafos que serão representados em forma de vetores numéricos. Estes vetores são obtidos a partir da extração dos valores dos pesos sinápticos de uma RNA após seu treinamento utilizando o algoritmo *Doc2vec*. Após a geração dos vetores de métodos e consultas, os mesmos podem ser comparados para obtenção das afinidades entre os mesmos. Assim, uma lista de métodos ranqueados é gerada a partir de um processo de recuperação de informação, baseado em comparação de vetores numéricos. A granularidade dos itens retornados, em resposta a uma descrição de característica, são métodos oriundos de classes de sistemas orientado a objetos em forma de uma lista ordenada. Um "arquivo verdade", aqui denominado *goldset* é então checado para validar os resultados e calcular a acurácia para cada consulta.

Neste estudo experimental, avaliou-se o ganho de acurácia durante a aplicação de duas abordagens voltadas às mudanças estruturais da RNA utilizada pelo algoritmo *Doc2vec*. Estas mudanças são a modificação da taxa de aprendizado e a calibração da função de custo (*loss function*). Apesar do estudo se restringir à avaliação destas duas mudanças, existem potenciais alterações como, por exemplo, a mudança da arquitetura da rede para um modelo baseado em *deep learning*, troca e calibração de outras funções de custo da rede, dentre outros.

Em trabalhos anteriores, que expressam o estado da arte para a localização de características [11], valores padrões do algoritmo *Doc2vec*, para os quesitos levantados, foram aplicados. Por exemplo, no *framework* Gensim [88], que possui uma implementação para o algoritmo *Doc2vec*, a taxa de aprendizado da RNA utilizada é mantida constante e igual a 0,025. Já a função de custo da RNA adotada por padrão é denominada *Negative*

Sampling que tem por padrão seu valor igual a 5. A utilização *ad hoc* desta configuração pode acarretar a obtenção de valores sub-ótimos para a acurácia como constatado em Corley, Damevski e Kraft[11], em que o algoritmo *Doc2vec* não ultrapassa a técnica LDA em 23,33 % de todos os casos analisados. Por fim, defende-se neste trabalho que somente uma combinação correta de configurações no entorno do algoritmo *Doc2vec* poderia elevar a acurácia do processo de localização de características de maneira generalizada em relação a outras técnicas como LSI, pLSI ou LDA.

Em função do aspecto aleatório da geração inicial dos pesos sinápticos da rede usada pelo algoritmo *Doc2vec*, especificamente neste experimento, optou-se por repetir dez vezes cada teste referente a cada combinação (*software* versus K). K é um hiperparâmetro da RNA, passível de ser calibrado, e representa o tamanho dos vetores numéricos que se deseja obter. Uma amostra de testes com valores muito baixos ou muito altos para o número de épocas de treinamento resultou em baixa acurácia. Assim, fixou-se empiricamente o número de épocas de treinamento em 50. Desta forma, tem-se uma visão mais embasada dos valores de MRR calculados e expostos na Seção 3.6. Para averiguar a variabilidade da amostra com cada conjunto de 10 valores de MRRs gerados, o desvio padrão foi calculado e pode ser consultado no Apêndice A.

3.4 Questões de Pesquisa

As seguintes questões de pesquisa foram propostas para este estudo empírico:

1. Q1: As abordagens individuais, baseadas em uma taxa de aprendizado cíclica ou calibração da função de custo "*Negative Sampling*", podem melhorar a acurácia do processo de localização de características de *software* em seu estado da arte?

2. Q2: Uma possível combinação das abordagens citadas, neste trabalho denominada abordagem híbrida, pode superar a acurácia em relação a cada abordagem individual?

Estas questões devem ser respondidas durante a discussão dos resultados neste mesmo capítulo.

3.5 Materiais e Métodos

Nesta seção é apresentado todo o arcabouço experimental que compreende desde o trabalho tomado como referência até ferramentas, *software*, *datasets* e metodologias utilizados.

3.5.1 Estudo de Referência

O *baseline* estabelecido para este estudo empírico, aqui denominado Estudo de Referência, está baseado no trabalho desenvolvido por Corley, Damevski e Kraft[11]. Trata-se de um estudo comparativo entre técnicas utilizadas para LCS. As técnicas citadas no estudo são LDA e *Doc2vec*. Os autores destacam a viabilidade da utilização de *"Deep Learning"* para tarefas de LCS. Apesar de não se tratar realmente de uma rede profunda, como afirmam os autores, os vetores representativos de métodos do código fonte podem ser aferidos a partir dos pesos sinápticos obtidos após o treinamento. Neste trabalho, o Estudo de Referência foi referenciado como DV* na seção de resultados. A reprodução deste mesmo estudo, a partir de novas versões das ferramentas, foi denominada DV**.

Com o objetivo de comparar as abordagens e apontar os pontos de evolução em relação ao Estudo de Referência, podem ser enumeradas as seguintes semelhanças entre os estudos:

1. A granularidade dos resultados retornados como resposta a uma consulta foi expressa em nível de métodos do código fonte;

2. A forma de geração do vetor numérico representativo da consulta deu-se a partir da somatória dos vetores de palavras individuais [30, 89];

3. A forma de geração dos vetores de métodos do código fonte utilizou o algoritmo *Doc2vec*;

4. O tamanho de janela (contexto das palavras) teve tamanho igual a 5 e,

5. A função de custo aplicada à RNA foi *Negative Sampling.*

Para Hamed e Bruce[90], a geração do vetor numérico representativo da consulta, a partir da média ou soma dos vetores de palavras que a compõe, tem sido a estratégia adotada em um série de estudos. Desta forma neste estudo optou-se pela geração do vetor da consulta através da somatória dos vetores representativos das palavras da consulta. Esta estratégia também foi adotada no Estudo de Referência.

Apesar de ambos os estudos basearem-se no uso da função de custo "*Negative Sampling*", o estudo empírico, descrito neste trabalho, foi mais abrangente e aplicou diferentes configurações a esta função com o objetivo de descobrir quais valores eram mais propensos a promover ganhos quanto à acurácia do processo de LCS.

Todo o processo utilizado pelo autor, com detalhes específicos da metodologia empregada no Estudo Referência, foi detalhado no Capítulo 6, referente aos trabalhos relacionados.

3.5.2 *Software*, ferramentas e hardware utilizados

Neste estudo foi utilizado um conjunto de ferramentas computacionais direcionadas ao PLN e RI tais como: linguagens de programação, bibliotecas e *frameworks* para ciência de dados, ferramentas de análise estatística e *A*mbiente de Desenvolvimento Integrado (IDE), como consta na Tabela 2.

Tabela 2 – Ferramentas Utilizadas

Software	Versão	Finalidade
Python	3.6	Construção do motor de busca.
Python	2.7	Geração de *datasets* no formato especificado em [11].
Java	diversas	Composição do código fonte dos *software* analisados.
Framework Gensim	3.4	Fornecimento de modelos *Word2vec* e *Doc2vec*.
Framework Gensim	0.10	Usado no experimento de referência.
IDE Pycharm	Community 2017.2	Ambiente de Desenvolvimento Integrado .
R	3.5.1	*Software* estatístico - Gráficos e análises. estatísticas

A Tabela 2 lista todas as linguagens de programação e ferramentas de *software* que foram utilizadas com suas respectivas versões e finalidades. O experimento iniciou-se com a produção do *dataset* a ser analisado. O *dataset* é gerado a partir da versão original do Estudo de Referência [11]. O experimento original foi executado no sistema opecional GNU/Linux, utilizando a versão do Python 2.7 e o *framework* Gensim 0.10. Após a geração do *dataset*, o experimento foi movido para o sistema operacional *Microsoft Windows* em que é utilizada a versão do Python 3.6 e o *framework* Gensim em sua versão 3.4. Para codificação e construção do motor de busca foi utilizada a IDE Pycharm. Também foram utilizados os algoritmos *Word2vec* e *Doc2vec* presentes no *framework* Gensim.

Na Tabela 3 foram enumerados os equipamentos utilizados durante o desenvolvimento do trabalho neste e em todos os outros capítulos.

Tabela 3 – Hardware utilizado nos experimentos

Id	Processador	Ram	Disco	Função
1	Intel i7 6700HQ, 4 núcleos, 2.6GHz	64GB	2TB	Servidor - testes e armazenamento
2	AMD Opteron 6272, 16 núcleos, 2.3GHz	8GB	500GB	Testes
3	AMD A10 6800K, 4 núcleos, 4.1GHz	16GB	620GB	Desenvolvimento e testes

3.5.3 Caracterização dos projetos de *software*

Neste estudo foi utilizado o *dataset* proposto por Dit et al.[91] que engloba um conjunto de seis *software* de código aberto escritos na linguagem de programação Java. Obseve que cada versão do ArgoUML é contada como um software diferente. O *dataset* foi recentemente utilizado em trabalhos voltados à LCS [92], além do Estudo de Referência. Este *dataset* é composto pelo código fonte dos *software*, um conjunto de descrições de características e o *goldset* que relaciona quais métodos são relevantes em relação às

características. Importante ressaltar que para cada característica existe pelo menos um método relevante que deva ser analisado durante a manutenção da mesma. Os *software* estudados compreendem: o ArgoUML[1] que é um diagramador *Unified Modeling Language* (UML); o JEdit[2] que é um editor de texto para programação; o JabRef[3] que é um gerenciador de referências bibliográficas; e o muCommander[4] que é um gerenciador de arquivos multiplataforma.

A Tabela 4 lista cada *software* com respectivo número de versão, número de características e número total de métodos analisados no *corpus*.

Tabela 4 – *Software* analisados

Software	Versão	Nº Características	Nº Métodos
ArgoUML	0.22	91	12353
ArgoUML	0.24	52	13064
ArgoUML	0.26.2	209	16880
JabRef	2.6	39	5357
jEdit	4.3	150	7305
muCommander	0.8.5	92	8799

O *dataset* continha um total de 633 descrições de características adotando um nível de granularidade de métodos. Os dados estão disponíveis para reprodução do estudo[5]. Em relação à escolha dos software, os critérios para escolha foram os seguintes: ser um software escrito na linguagem Java; usar o SVN[6] como repositório bem como armazenar os *logs* de alteração do código fonte; manter um rastro das solicitações de mudanças e evolução do software. A escolha das características de software obedeceu o critério de disponibilidade de informações relativas à identificação da característica no arquivo de *log* de alteração da mesma. Este dataset foi utilizado em um número considerável de trabalhos relativos à área de Engenharia de Software [11, 93, 94].

3.5.4 Abordagem utilizando a taxa de aprendizado cíclica

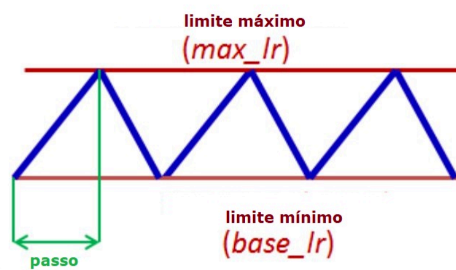
Como foi visto no Capítulo 2, ao final do treinamento de uma RNA, os pesos sinápticos atribuídos às arestas retém o conhecimento aprendido pela RNA. Assim, controlar a forma como tais pesos são gerados pode proporcionar obtenção de vetores numéricos de melhor qualidade ao se utilizar o algoritmo *Doc2vec*. Se as palavras e parágrafos forem representados em vetores numéricos, então, tarefas correlatas como a LCS e LBS podem se beneficiar de tal representação.

[1] <http://argouml.tigris.org>
[2] <http://www.jedit.org>
[3] <http://jabref.sourceforge.net>
[4] <http://www.mucommander.com>
[5] http://www.cs.wm.edu/semeru/data/msr13/
[6] Apache Subversion - SVN: https://subversion.apache.org/

Um dos parâmetros mais influentes nesta situação é a taxa de aprendizado da RNA [95]. A implementação do algoritmo *Doc2vec* do *framework* Gensim adota um valor fixo para a taxa de aprendizado cujo valor é 0,025. A taxa de aprendizado da RNA é representado pelo parâmetro *alpha* em um dado momento do treinamento. O valor 0,025 foi obtido experimentalmente a partir da aferição do parâmetro *alpha* durante a replicação do estudo sobre LCS reportado em Corley, Damevski e Kraft[11]. Smith[95] demonstrou que a variação da taxa de aprendizado da RNA durante o treinamento, a partir de uma nova política denominada *cyclical learning rates*, foi capaz de aumentar a acurácia durante tarefas de classificação. Em seu estudo original, a aplicação da nova técnica envolve o contexto de RNAs profundas e levanta a questão se sua eficiência pode ser verificada em RNAs rasas que são a base do algoritmo *Doc2vec*.

O modelo de adaptação da taxa de aprendizado *cyclical learning rates* é equivalente ao chamado modelo Triangular de Beigi[96] que é ilustrado na Figura 13. Neste modelo, faz-se a taxa de aprendizado variar dentro de um escopo de valores ao invés de adotar um valor fixo ou fazê-la decrescer exponencialmente. Assim, evidenciou-se um valor mínimo (base_lr) e outro máximo (max_lr), além do parâmetro *stepsize* que é numericamente igual à metade do número de iterações em cada época de treinamento.

Figura 13 – Variação de taxa de aprendizado Triangular



Fonte: Traduzido de Smith[95].

Observa-se que apareceram outras implementações relativas à política de taxa de aprendizado variável como descrita nesta nesta subseção. Isto pode ser evidenciado entre os principais *frameworks* ligados ao aprendizado de máquina como *Tensorflow*[7] e Keras[8]. Apesar disto, é importante destacar que sua eficácia ainda não foi verificada para RNAs rasas como aquelas usadas pelo algoritmo *Doc2vec*. Além disto, destaca-se o fato destas RNAs estarem sendo usadas em tarefas de manutenção de *software* como é a intenção deste trabalho. Neste trabalho, a aplicação da política baseada em uma taxa de aprendizado variável foi referenciada como DV**+LRT.

[7] https://github.com/mhmoodlan/cyclic-learning-rate
[8] https://github.com/bckenstler/CLR

3.5.5 Abordagem utilizando a função de custo da Rede Neural Artificial

Como apresentado anteriormente, no Capítulo 2, a função de custo influencia diretamente a política de atualização dos pesos sinápticos que compõe a RNA durante a aplicação do algoritmo *Backpropagation*. Rigutini et al.[97] sugerem a permuta da função de custo para avaliar sua proposta denominada *CmpNN* voltada a melhoria da função de similaridade entre itens retornados em um *rank*.

Neste estudo empírico foram definidas as abordagens DV**+NGS10, DV**+NGS20, DV**+NGS30, DV**+NGS40 e DV**+NGS50 que representam uma tentativa de se obter a melhor calibração para a função de custo *Negative Sampling* que é utilizada pelo algoritmo *Doc2vec*. Os números 10, 20, 30, 40 e 50 representam o número de *Negative samplings* escolhidos para trabalhar com o modelo skip-gram proposto por Mikolov conforme visto no Capítulo 2. Outra função de custo alternativa para geração de vetores de palavra é a função *hierarquical softmax*. Esta função foi utilizada no Capítulo 4 para localização de *bugs*.

3.5.6 Combinação de abordagens

As abordagens individuais propostas neste capítulo não são mutualmente exclusivas. Assim, as abordagens LRT e NGS foram testadas e os resultados reportados nesta seção. Elas podem ser combinadas experimentalmente com o objetivo de se avaliar o aumento da acurácia final do processo de LCS. Desta forma as questões de pesquisa propostas focam-se nesta avaliação conforme descrito na próxima seção.

3.6 Resultados Experimentais

Nesta seção são apresentados e discutidos os resultados do experimento. As repostas às questões de pesquisa, propostas na Seção 3.4, são abordadas sendo então possível conhecer as performances individuais e conjunta das abordagens LRT e NGS. Os resultados foram condensados no Gráfico da Figura 14. Para cada combinação de *software* versus tamanho de vetor (100, 200, 300, 400 e 500), foram expostos os resultados para as abordagens: DV*, DV**,DV**+NGS10, DV**+NGS20, DV**+NGS30, DV**+NGS40, DV**+NGS50 e DV**+LRT+NGS50.

Neste momento é relevante discutir qual critério utilizar para escolher dentre as abordagens que usam NGS qual seja mais representativa e que irá compor a abordagem conjunta. Neste sentido, a abordagem DV**+NGS50 domina outras concorrentes se for levando em consideração os maiores valores absolutos de MRR em todos os *software*, com exceção do JabRef. Existe a possibilidade de se testar todas, mas esta tarefa é computacionalmente cara. Desta forma, a abordagem conjunta foi representada pela abordagem

DV**+LRT+NGS50 que serve como referencial para comparação abordagens individuais e conjuntas.

Os resultados para a abordagem LDA*, provenientes do Estudo de Referência, foram listados no Apêndice A. Observou-se que os valores de MMR obtidos a partir da aplicação desta técnica são muito baixos quando comparados à abordagem DV**+LRT+NGS50 que é superior em 93,33% de todas as combinações (*software versus* K) testadas.

A primeira evidência que colabora com a ideia de que abordagens individuais são mais limitadas quando comparadas com a combinação conjunta (DV**+LRT+NGS50), pode ser verificada a partir da análise do desvio padrão obtido em cada relação "*software* versus K", ilustrada na Tabela 14, presente no Apêndice A. Neste sentido, deduzindo-se o valor do desvio padrão de cada valor de MRR médio para a combinação conjunta, ainda assim tem-se sempre valores superiores em relação a qualquer uma das abordagens individuais.

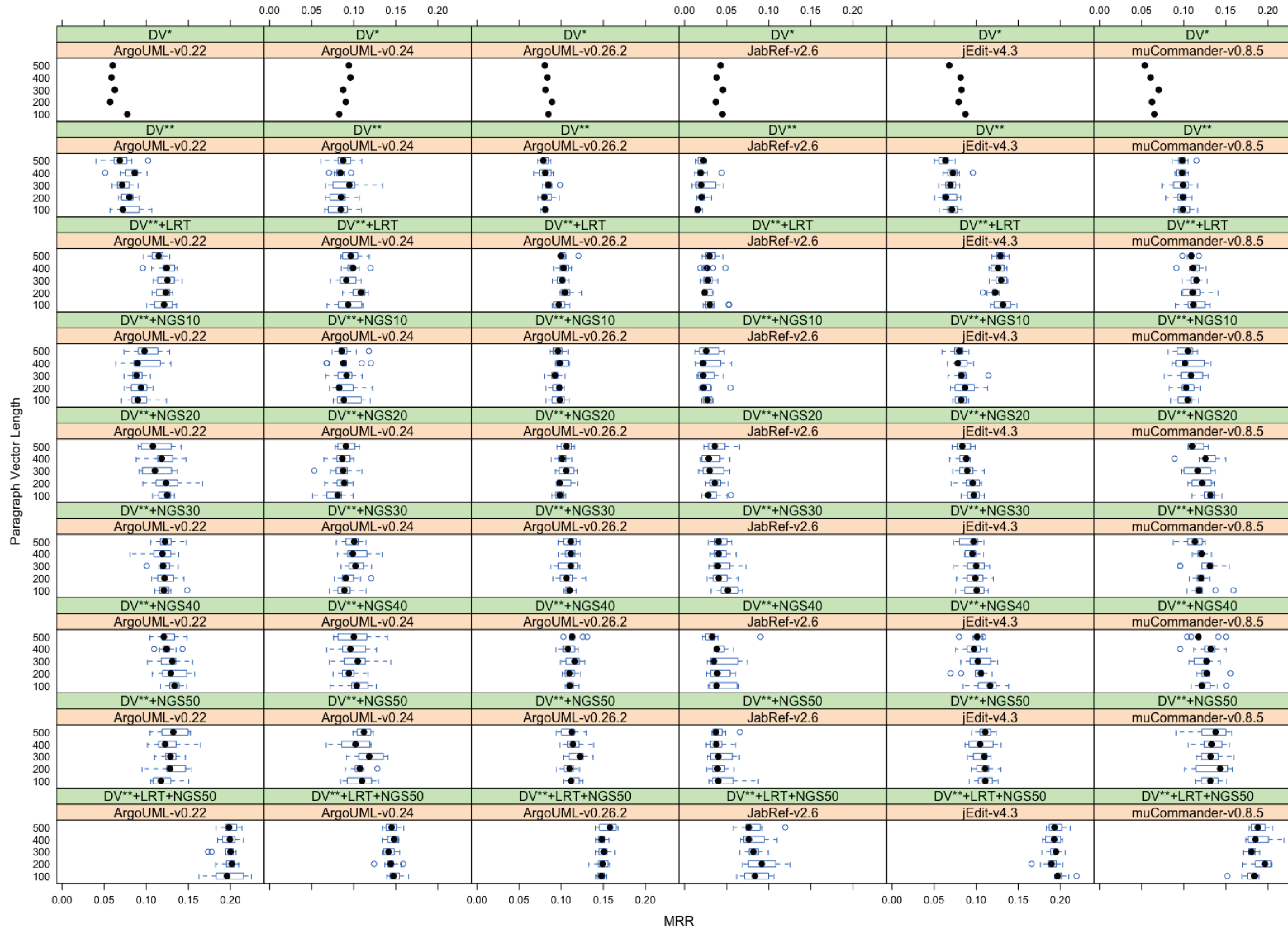
Desta forma, não se pode afirmar que as abordagens individuais sejam melhores que a combinação das mesmas o que concorre para que a 2ª questão de pesquisa Q2 seja verdadeira. Usando o mesmo raciocínio, a abordagem DV**+LRT se mostra superior ao experimento de referência nos *software* ArgoUML v0.22, jEdit v4.3 e muCommander v0.8.5. Para os outros *software* esta situação não se verifica. Portanto, com base na análise do desvio padrão, defende-se aqui que as abordagens individuais não alcançam melhor acurácia no processo de LCS, em resposta à primeira questão de pesquisa Q1.

A hipótese de que a abordagem conjunta seja superior em relação às individuais está relacionada à questão de pesquisa Q2. Assim, o objetivo concentrou-se na tentativa de provar que existe uma diferença significativa entre os valores de MRR obtidos para a combinação DV**+LRT+NGS50 e aqueles obtidos para outras combinações individuais. Para comparar todas as abordagens individuais com a abordagem conjunta foi então utilizado o teste de Kruskall-Wallis [98].

Neste experimento, o nível de significância foi definido para 0,05. Após a análise de cada valor de *p-value* retornado pelo teste concluiu-se que existe um alto grau de diferença entre os grupos de valores de MRR comparados em cada abordagem. Posteriormente, procedeu-se uma análise de *post-hoc* a partir do teste de *Post-hoc Nemenyi*, usando o método *Tukey*[9]. Os resultados apontaram uma diferença significativa entre a abordagem conjunta (DV**+LRT+NGS50) e a reprodução do experimento de referência (DV**). Isto significou mais uma justificativa de a escolha desta vertente para representar a abordagem conjunta.

[9] Post-hoc Nemenyi - https://cran.r-project.org/web/packages/PMCMR/vignettes/PMCMR.pdf

Figura 14 – MRR médio para o experimento com RNA



Por fim, o teste A de *Varga-Delaney*[10] foi utilizado para enfatizar o nível de diferença entre a abordagem referência (DV**) e a abordagem híbrida (DV**+LRT+NGS50). Assim, as duas amostras de MRR a serem comparadas foram provenientes das abordagens citadas. Na aplicação do teste o primeiro parâmetro foi representado pelo conjunto de valores de MRR da abordagem referência (DV**) e o segundo pelos valores provenientes da abordagem híbrida (DV**+LRT+NGS50), necessariamente nesta ordem. Os resultados podem ser assim enumerados:

1. Para todos os *software*, exceto o ArgoUML 0.24, a abordagem híbrida (DV** + LRT + NGS50) é frequentemente superior em todos os casos (valor da medida A=1).

2. Para o ArgoUML 0.24 a abordagem híbrida (DV**+LRT+NGS50) é frequentemente superior em quase todos os casos (valor da medida A=0.9984).

Portanto, são fortes as evidências que a combinação das abordagens DV**, DV**+LRT e DV**+NGS50, como apresentado neste estudo, é mais proeminente na consecução de ganhos de acurácia em LCS do que as abordagens individuais.

3.7 Discussão dos Resultados

Durante o processo de LCS, o desenvolvedor inicia seu trabalho analisando o topo da lista de métodos retornados em ordem semântica e investiga método a método na sequência em que aparecem. Desta forma, quanto menos métodos irrelevantes forem analisados, mais rápido o desenvolvedor encontrará aqueles que necessitam realmente ser analisados em sua tarefa de manutenção da característica.

O caso real de LCS usando uma técnica no estado da arte, reportado neste estudo, apresenta 0,0775 como melhor valor de MMR para o ArgoUML (versão 0.22) com um vetor de parágrafo de tamanho 100. A análise deste valor de MRR implica que em média o desenvolvedor necessita analisar aproximadamente 13 métodos do código fonte, em cada consulta (descrição de característica), antes de encontrar um método relevante, em um total de 91 consultas. Entretanto, em muitas consultas, dentre as 91 analisadas para o ArgoUML, o desenvolvedor terá que analisar mais de 11000 métodos até encontrar o que deseja. Esta baixa acurácia, que foi discutida por Ali et al.[44], acaba criando um cenário desfavorável para a efetiva utilização de ferramentas criadas com o intuito de diminuir os custos de LCS. Comparativamente, o estudo desenvolvido neste capítulo, usando a abordagem híbrida DV**+LRT+NGS50 obteve 0,1762 para o valor médio de MRR nas mesmas condições citadas anteriormente. Desta forma, o número médio de métodos analisados, até encontrar o primeiro relevante em relação a uma consulta, foi reduzido de 13 para 7. Isto demonstra a importância de se calibrar corretamente as técnicas relacionadas à LCS e o êxito obtido em fazê-lo neste trabalho.

[10] Teste A de Varga-Delaney - https://rdrr.io/cran/effsize/man/VD.A.html

Diversas pesquisas sustentam o fato que o ajuste correto das técnicas de RI e PLN pode trazer expressivos ganhos na LCS. Assim, a proposição do LSI probabilístico (pLSI) e o uso de algoritmos genéticos para descoberta da melhor calibração para a técnica LDA (LDA-GA) ilustram esta linha de pensamento. Neste sentido, o estudo proposto neste capítulo expande esta tendência a partir da aplicação do algoritmo *Doc2vec* em conjunto com uma combinação de avanços na área de RNA. Nesta linha, Speer e Chin[99] confirmam a hipótese, defendida neste trabalho, de que a combinação de várias abordagens é necessária para produção de representações de palavras e parágrafos de alta qualidade. Finalmente, a análise dos estudos propostos por Caselles-Dupré, Lesaint e Royo-Letelier[86] e Hutter, Hoos e Leyton-Brown[100] indicam que a calibração adequada dos valores dos hiperparâmetros é fundamental e sua adequada escolha acaba sendo dependente da tarefa e dos dados a serem analisados. Desta forma, a descoberta de novas abordagens para LCS usando RNA e seus avanços mantém-se como uma atividade empírica e instigante.

Alguns fatores concorrem para não utilização da técnica LDA neste estudo. Experimentalmente, o custo computacional de execução de testes utilizando esta técnica foi maior quando comparado às técnicas que utilizam o algoritmo *Doc2vec*. Outro teste realizado tentou utilizar sem sucesso os tópicos gerados para enriquecimento da consulta que seria utilizada no motor *Doc2vec*. Por fim, Agrawal, Fu e Menzies[51] fazem algumas ressalvas quanto à utilização de LDA em experimentos na área de Engenharia de *Software*. Segundo os autores, os hiperparâmetros precisam ser bem calibrados para que não haja instabilidade dos tópicos gerados entre execuções sucessivas da técnica o que dificulta a reprodução do experimento.

3.8 Resumo do Capítulo

Este capítulo teve como objeto de estudo a aplicação de avanços obtidos na área de RNA em LCS. Ficou demonstrado que a correta seleção e calibração dos hiperparâmetros do algoritmo *Doc2vec* obteve ganhos reais na acurácia do processo de LCS.

As abordagens apresentadas para melhoria do processo de LCS foram:

1. DV**+LRT - uma abordagem voltada a explorar uma política de taxa de aprendizado variável aplicada durante o processo de treinamento da RNA responsável por produzir vetores de palavras e parágrafos.

2. DV**+NGS - uma abordagem que visa descobrir empiricamente qual valor melhor calibra a função de custo *"Negative Sampling"* utilizada pelo algoritmo *Doc2vec*.

3. DV**+LRT+NGS50 - abordagem proposta envolvendo a combinação das abordagens anteriores. Esta abordagem mostrou desempenho superior às demais abordagens individuais.

Este capítulo apresentou evidências suficientes para se afirmar que avanços propostos para a teoria de RNAs são capazes melhorar o processo de LCS, tendo como base o algoritmo *Doc2vec*. Por fim, o autor deste trabalho obteve êxito com a publicação do artigo *"Improving feature location accuracy via paragraph vector tuning"* no periódico *Information and Software Technology/Elsevier* [101]. No artigo, aplicou-se as abordagens NGS e LRT ao algoritmo Doc2vec, obtendo-se melhores resultados quanto à acurácia do processo de LCS. Detalhes da publicação podem ser encontrados no Capítulo 7.

Capítulo 4

Aprimoramento do uso de *Doc2vec* na Localização de *Bugs* de *Software*

A localização de *bugs* e características de software são tarefas similares. Ambas partem de uma descrição textual, de um *bug* ou característica, e ordenam entidades do código fonte, como métodos ou classes, gerando ao final do processo uma lista ordenada por similaridade semântica. Neste sentido, este capítulo se dedicou a descrever a tarefa de localização de *bugs* no código fonte de sistemas de *software* (LBS) utilizando o algoritmo *Doc2vec*. Dada uma descrição de *bug*, desejava-se encontrar quais classes foram alteradas para correção do *bug*.

Desta forma, durante o desenvolvimento deste capítulo foram propostos testes com o algoritmo *Doc2vec* utilizando uma quantidade maior de testes e hiperparâmetros do que aqueles utilizados no Capítulo 3. O Estudo de Referência, descrito na Seção 4.5.1, foi reproduzido e teve seus resultados comparados, e posteriormente combinados em uma abordagem denominada BZZ, com aqueles obtidos a partir do uso do algoritmo *Doc2vec*. A abordagem LRT, baseada em uma taxa de aprendizado cíclica para a RNA, também foi adicionada ao experimento. Por fim, as classes foram segmentadas em métodos para avaliar se a mudança na granularidade impacta a acurácia do processo de LBS.

4.1 Referencial Teórico

Esta seção se dedica a tratar de conceitos específicos não abordados no Capítulo 2, mas que se mostram relevantes para o desenvolvimento deste capítulo.

A técnica denominada *Pseudo Feedback Relevance* foi utilizada no estudo de referência para reformulação da descrição de *bugs*. Trata-se de um conjunto de técnicas voltadas à melhoraria da acurácia do sistema de busca [102]. Nesta modalidade, após o sistema retornar os resultados de pesquisa para uma consulta prévia, o usuário faz uma avaliação dos documentos separando-os em dois grupos: relevantes e não relevantes. Então as informações obtidas a partir dos documentos relevantes são extraídas para reformulação

da consulta. Quando o usuário não está disponível ou não aceita fazer o papel de julgador, pode-se assumir que os documento do topo da lista retornada sejam relevantes. Esta última estratégia é denominada "*Pseudo Feedback Relevance*". Hamed e Bruce[90] relata que este mecanismo foi usado para melhoria da qualidade das consultas.

A agregação de *Rankings* também foi utilizada neste trabalho. De maneira resumida, o problema da agregação de *Rankings* corresponde à atividade de computar um *ranking* consensual usando como entrada um conjunto de outros *rankings* obtidos a partir de diferentes técnicas [103]. Neste sentido, o *rank* ótimo caracteriza-se por ter a menor distância média para todos os *ranks* de entrada.

4.2 Planejamento Experimental e Metodologia utilizada

Este experimento se dedicou à melhoria da acurácia do algoritmo *Doc2vec* quando aplicado à LBS no código fonte de sistemas orientados a objeto. Em um primeiro momento elencou-se os principais trabalhos cujo objetivo fosse localizar *bugs* no código fonte do *software*. Neste sentido, buscou-se aqueles que representassem o estado da arte na localização de *bugs*, além de poderem ser reproduzidos em sua íntegra. O Estudo de Referência escolhido foi detalhado na Seção 4.5.1 e pôde ser reproduzido com a cooperação do autor original. Este último usou uma granularidade de classe para expressar seus resultados. Desta forma, inicialmente, esta entidade do código fonte foi utilizada para compor uma lista de resultados para o desenvolvedor, evoluindo posteriormente para utilização de métodos conforme evidências de melhor desempenho encontradas em [104].

Os hiperparâmetros do algoritmo *Doc2vec* que foram testados neste capítulo estão listados na Tabela 5. Assim, optou-se pela avaliação de um número maior de hiperparâmetros[1], que, com seus respectivos valores discretos, podem indicar como a acurácia do processo de LBS pode ser melhorada. Por questões de custo computacional, o número de testes foi limitado fixando-se um tamanho de vetor para as classes do código fonte igual a 100. O critério de escolha deste valor baseia-se no fato de que este tamanho apresentou MRR similar ou superior aos outros tamanhos em testes prévios realizados com as melhores calibrações obtidas no experimento com LCS do Capítulo 3. Esta aferição foi baseada em testes com abordagens discretas listadas na Tabela 15, disponível no Apêndice B. Os resultados destes testes foram disponibilizados na Tabela 16, disponível também no Apêndice B. Definido o tamanho do vetor, a combinação dos outros hiperparâmetros e abordagens, testadas neste trabalho, resultou em um total de 720 combinações. Se este valor for multiplicado pelos 6 *software* a serem testados, então, foram efetuados 4320 testes para cálculo de MRR. Os hiperparâmetros são os seguintes: o tamanho da janela

[1] https://radimrehurek.com/gensim/models/doc2vec.html

(WDS), que varia em valores discretos entre 10 e 50 ; o uso da função de custo NGS com valores discretos variando entre 10 e 50; o uso da função de custo HSM; a modalidade de treinamento de parágrafos usando *distributed memory* (PV-DM) ou *distributed bag of words* (PV-DBOW). As abordagens compreendem o uso de uma taxa de aprendizado cíclica (LRT) e o impulsionamento a partir do resultado prévio da ferramenta Blizzard (BZZ)[2]. O objetivo com este intento foi determinar quais fatores eram preponderantes para aumentar a acurácia do algoritmo *Doc2vec* na localização de *bugs* usando classes. Em seguida aplicou-se a melhor calibração encontrada para métodos durante o experi-

Tabela 5 – Hiperparâmetros testados para localização de *bugs* de *software*

*H*iper-Parâmetro	Descrição Sucinta	Valores testados
VTS	Tamanho do Vetor	100
WDS	Tamanho da Janela	10,20,30,40,50
NGS	Uso de *Negative Sampling*	10,15,20,25,30,35,40,45,50
HSM	Uso de *Hierarquical Softmax*	*Sim, Não*
DM	Uso de PV-DM ou PV-DBOW	0,1

mento de localização de características, isto é, foi testada a calibração DV**+LRT + NGS50+WDS5. Como diferentes *datasets* muitas vezes requerem diferentes calibrações optou-se por testar também calibrações diferenciadas com foco na alteração do tamanho da janela. Por fim, promoveu-se uma combinação dos resultados obtidos no Estudo de Referência com aqueles obtidos a partir da aplicação do algoritmo *Doc2vec*. Esta é uma tendência no meio científico para se obter melhores resultados [105]. Neste intento, impulsionou-se artificialmente os resultados encontrados pelo algoritmo *Doc2vec* com base no ranking provido pela ferramenta *Blizzard*. Aplicou-se então uma agregação de *rankings* em que foram impulsionadas as 10 primeiras classes (Top@10) recomendadas pela Ferramenta *Blizzard*. Testes discretos, com uma configuração baseada nas 100 primeiras classes (Top@100), também foram realizados, notando-se queda na acurácia. Quanto maior o valor de K, mais lento é o teste realizado com o algoritmo *Doc2vec*. Assim, cada valor de similaridade obtido no *Doc2vec* sofreu um acréscimo de 1 unidade para as Top@10 classes. Esta tática foi nomeada na Seção 4.6 com a sigla BZZ. Por fim, testou-se a calibração que envolve a função de custo *Hierarquical Softmax* que representa uma alternativa ao *Negative Sampling*. HSM é a denominação utilizada neste trabalho para nomear o uso desta opção. HSM e NGS foram usadas como mutualmente exclusivas neste trabalho.

Desta forma, os resultados apresentados na Seção 4.6 abrangem um grande número de tentativas com o objetivo de encontrar uma combinação de hiperparâmetros capaz de aumentar a acurácia do processo de LBS. Além disso, novas tentativas com a utilização de um nível de granularidade menor, baseado em métodos do código fonte, foi implementado e seus resultados discutidos.

[2] Blizzard - https://github.com/masud-technope/BLIZZARD-Replication-Package-ESEC-FSE2018

4.3 Aspectos técnicos do experimento

Em função do tamanho do *dataset* o teste e cálculo de MRR para cada *software* foi realizado somente uma vez, diferentemente do estudo realizado com características, relatado no Capítulo 3.

Nos experimentos relatados no presente capítulo, o interesse recaiu em localizar classes no código fonte. Estas classes eram aquelas que sofreram alteração em função da correção do *bug* em questão. Um *goldset* estava disponível para avaliação dos resultados. Por fim, o auxílio do autor original na reprodução do experimento, descrito no Estudo de Referência, foi fundamental para consecução dos resultados.

4.4 Questões de Pesquisa

As seguintes questões de pesquisa foram propostas para este estudo empírico:

1. Q1: Como os hiperparâmetros do algoritmo *Doc2vec*, listados neste estudo, afetam sua performance na localização de *bugs*?

2. Q2: A utilização de uma política de treinamento cíclica (LRT) para a RNA do algoritmo *Doc2vec* e a combinação de ranking (BZZ) impactam positivamente a acurácia na localização de *bugs*?

Estas questões foram respondidas durante a discussão dos resultados realizada neste mesmo capítulo.

4.5 Materiais utilizados

Nesta seção é apresentado todo o arcabouço experimental que compreende desde o trabalho utilizado como referência até ferramentas, *software*, *datasets* e metodologias utilizados.

4.5.1 Estudo de Referência

O Estudo de Referência utilizado como *baseline* para este capítulo foi baseado na LBS proposta e executada por [5]. Neste estudo, os autores propõem uma ferramenta denominada *Blizzard* que reformula a descrição do *bug* removendo ou adicionando termos. O trabalho em questão lista dois pontos negativos que prejudicam a performance das técnicas de recuperação de informação em função do conteúdo da descrição do *bug*: a carência de nomes de entidades (nomes de identificadores, classes, pacotes, métodos relacionados ao *bug*) ou o excesso de informações de *debug* como *stack traces*. Eles argumentam que abordagens tradicionais desenvolvidas até o presente momento estão limitadas à aplicação

de um pré-processamento insuficiente em que somente *stop-words*, elementos de pontuação e dígitos são retirados. Para alcançar os objetivos propostos, os autores desenvolveram a ferramenta Blizzard que possui o código fonte disponibilizado para reprodução[3]. Os dados para reprodução do estudo também foram disponibilizados e compreendem: consultas (descrições de *bugs*), código fonte dos software e *goldsets*[4].

A metodologia empregada pelos autores parte inicialmente de um relatório de *bug*. Cada *bug* é então classificado em uma de três categorias:

1. $\mathrm{BR_{ST}}$ - se o relatório de *bug* possui um ou mais *stack traces* e precisa ser filtrado para remover o ruído. Segundo o autor, relatórios de *bug* com *stack traces* possuem muito ruído. Portanto, usar estes relatórios diretamente, sem tratamento, acarreta baixa acurácia no processo de LBS. No estudo, o autor transforma os rastros em um grafo de rastros e identifica importantes palavras chaves através da aplicação do algoritmo PageRank.

2. $\mathrm{BR_{PE}}$ - se o relatório contém alguns elementos como nome de métodos e classes. Nesta categoria, *bugs* cuja descrição contenham *stack traces* não são considerados. Novamente, os autores apostam na escolha adequada de palavras chaves dentre aquelas palavras disponíveis no relatório de *bug*. Eles exploram a coocorrência de termos e as relações sintáticas entre os mesmos para identificar importantes palavras chave.

3. $\mathrm{BR_{NL}}$ - *Bugs* classificados nesta categoria não possuem elementos de programa e nem *stack traces*. Para os autores, relatórios que apresentam somente texto em linguagem natural apresentam baixa performance e precisam ser complementados. No estudo, este tipo de relatório de *bug* é tratado a partir de um mecanismo de *pseudo feedback relevance* antes de serem submetidos diretamente ao mecanismo de busca. Os autores coletam nome de assinaturas de métodos e campos nos Top-K (K=10) primeiros documentos retornados para extração de palavras chaves que permitirão ao reformulação do relatório de *bug*.

Após reformulação da consulta, utilizou-se um motor de busca baseado na biblioteca Lucene[5]. Foi usado para ranqueamento a função *Okapi BM25*[6].

As melhorias apontadas pelo trabalho podem ser assim enumeradas:

1. Uma nova técnica para reformulação de consultas com base no filtro de ruídos e adição de novos termos relevantes à consulta;

[3] https://github.com/masud-technope/BLIZZARD-Tool-New
[4] https://github.com/masud-technope/BLIZZARD-Replication-Package-ESEC-FSE2018
[5] https://lucene.apache.org/
[6] https://nlp.stanford.edu/IR-book/html/htmledition/okapi-bm25-a-non-binary-model-1.html

2. Uma avaliação de 5139 relatórios de *bug* com incremento em acurácia promovido pela nova técnica em comparação aos trabalhos no estado arte;

3. Um protótipo de *software* funcional com dados experimentais para replicação do estudo.

4.5.2 *software* e ferramentas utilizados

As ferramentas utilizadas na localização de classes no código fonte foram listadas na Tabela 6.

Tabela 6 – Ferramentas Utilizadas

Software	Versão	Finalidade
Python	3.6	Construção do motor de busca.
Framework Gensim	3.4	Fornecimento de modelos *Word2vec* e *Doc2vec*.
Blizzard	-	Ferramenta desenvolvida no Estudo de Referência para localização de *bugs* em classes.
IDE PycharmCommunity Edition	2019.3	Ambiente de Desenvolvimento Integrado .
R	3.5.1	*Software* estatístico - Gráficos e análises. estatísticas
Netbeans	8.1	IDE - desenvolvimento de programas de *parsing*
JavaParser	3.15.9	Biblioteca para *parsing* de código em Java

O experimento iniciou-se com a reprodução do experimento descrito no *Estudo de Referência*. Também foram utilizados os algoritmos *Word2vec* e *Doc2vec* presentes no *framework* Gensim. O mecanismo de consulta foi desenvolvido utilizando a IDE Pycharm que permite a edição de código em Python. Para tratar métodos do código fonte, foi necessário dividir as classes em métodos utilizando a ferramenta *JavaParser*[7]. Na Tabela 7, pode-se encontrar cada *software* estudado no Estudo de Referência com o respectivo número de versão, número de *bugs* a serem localizados bem como o número total de classes do código fonte.

Tabela 7 – *software* analisados

Software	Nº de *bugs*	Nº de classes/enum/interfaces
ecf	553	2802
eclipse.jdt.core	989	5908
eclipse.jdt.debug	557	1532
eclipse.jdt.ui	1115	10927
eclipse.pde.ui	872	5334
tomcat70	1053	1841

4.6 Discussão dos Resultados

Nesta seção, são discutidos os resultados obtidos a partir da instrumentalização empírica realizada neste trabalho. Foi realizado um teste de massa com a combinação de

[7] https://javaparser.org/

hiperparâmetros utilizados pelo algoritmo *Doc2vec* listados na Seção 4.2. Esta abordagem buscou responder a questão de pesquisa Q1. Outra abordagem tentada no sentido de aumentar a acurácia do processo de LBS refere-se à utilização de métodos do código fonte a invés de classes, o que poderia ser mais eficiente [104, 6]. Empreendeu-se esta abordagem com vistas a responder a questão de pesquisa Q2.

4.6.1 Combinação de hiperparâmetros do *Doc2vec* e Abordagens

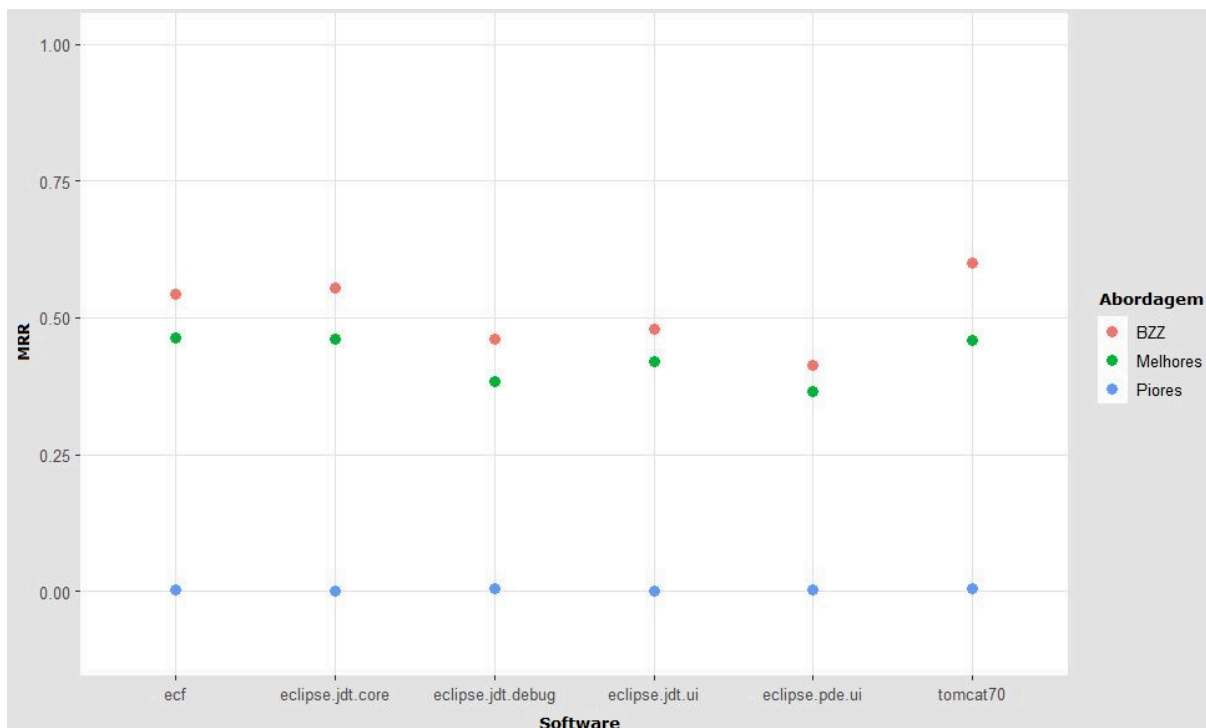
A Tabela 8 lista os melhores e piores valores de MRR obtidos para cada um dos 6 *software* testados, além dos resultados obtidos no Estudo de Referência. Importante lembrar que para cada *software* foram testadas 720 combinações de hiperparâmetros distintas e o tamanho de vetor configurado para cada parágrafo (classes e consulta) foi igual a 100. Foi destacado em cinza claro os resultados encontrados no Estudo de Referência, obtidos a partir da Ferramenta *Blizzard*, que continuou dominando os melhores resultados para o valor do MRR. Em seguida separou-se as melhores e piores configurações obtidas, sendo que para facilitar a comparação, estas foram destacadas respectivamente nas cores verde e vermelho.

Tabela 8 – *M*elhores e piores resultados de MRR para cada *software*

Software	WDS	NGS	HSM	DM	LRT	BZZ	MRR
			Blizzard				
ecf	-	-	-	-	-	Sim	0.5426
eclipse.jdt.core	-	-	-	-	-	Sim	0.5537
eclipse.jdt.debug	-	-	-	-	-	Sim	0.4616
eclipse.jdt.ui	-	-	-	-	-	Sim	0.4801
eclipse.pde.ui	-	-	-	-	-	Sim	0.4143
tomcat70	-	-	-	-	-	Sim	0.5999
			Melhores Resultados				
ecf	40	45	Não	PV-DM	Sim	Sim	0.4634
eclipse.jdt.core	40	20	Não	PV-DM	Sim	Sim	0.4606
eclipse.jdt.debug	20	Não	Sim	PV-DM	Sim	Sim	0.3846
eclipse.jdt.ui	40	10	Não	PV-DM	Sim	Sim	0.4208
eclipse.pde.ui	50	10	Não	PV-DM	Sim	Sim	0.3645
tomcat70	50	40	Não	PV-DM	Sim	Sim	0.4579
			Piores Resultados				
ecf	20	Não	Sim	PV-DBOW	Sim	Não	0.0042
eclipse.jdt.core	10	Não	Sim	PV-DBOW	Sim	Não	0.0001
eclipse.jdt.debug	40	Não	Sim	PV-DBOW	Não	Não	0.0064
eclipse.jdt.ui	40	Não	Sim	PV-DBOW	Não	Não	0.0012
eclipse.pde.ui	50	Não	Sim	PV-DBOW	Não	Não	0.0029
tomcat70	30	50	Não	PV-DBOW	Não	Não	0.0063

Para uma melhor entendimento, uma linha da Tabela 8 foi realçada com linha pontilhada, a qual passa-se a explicar agora. Trata-se da linha em que foi listado o melhor resultado para MRR do *software Eclipse Communication Framework* (ecf). As configurações de calibração do algoritmo *Doc2vec* utilizadas foram: uso de uma janela de tamanho 40; uso da função NGS igual a 45, o que implica que somente 46 palavras (45 palavras e um parágrafo) serão atualizados em uma passada (época) de treinamento da rede; não foi utilizada função de peso *Hierarquical SoftMax*; o modo de treinamento usado para geração de vetores de parágrafos usado nesta configuração foi PV-DM. Além dos hiperparâmetros citados, esta configuração também implica a combinação da abordagem LRT e a utilização do ranking obtido a partir da ferramenta *Blizzard* para impulsionar os resultados gerados pelo *Doc2vec*. As outras linhas da tabela podem ser interpretadas de maneira similar.

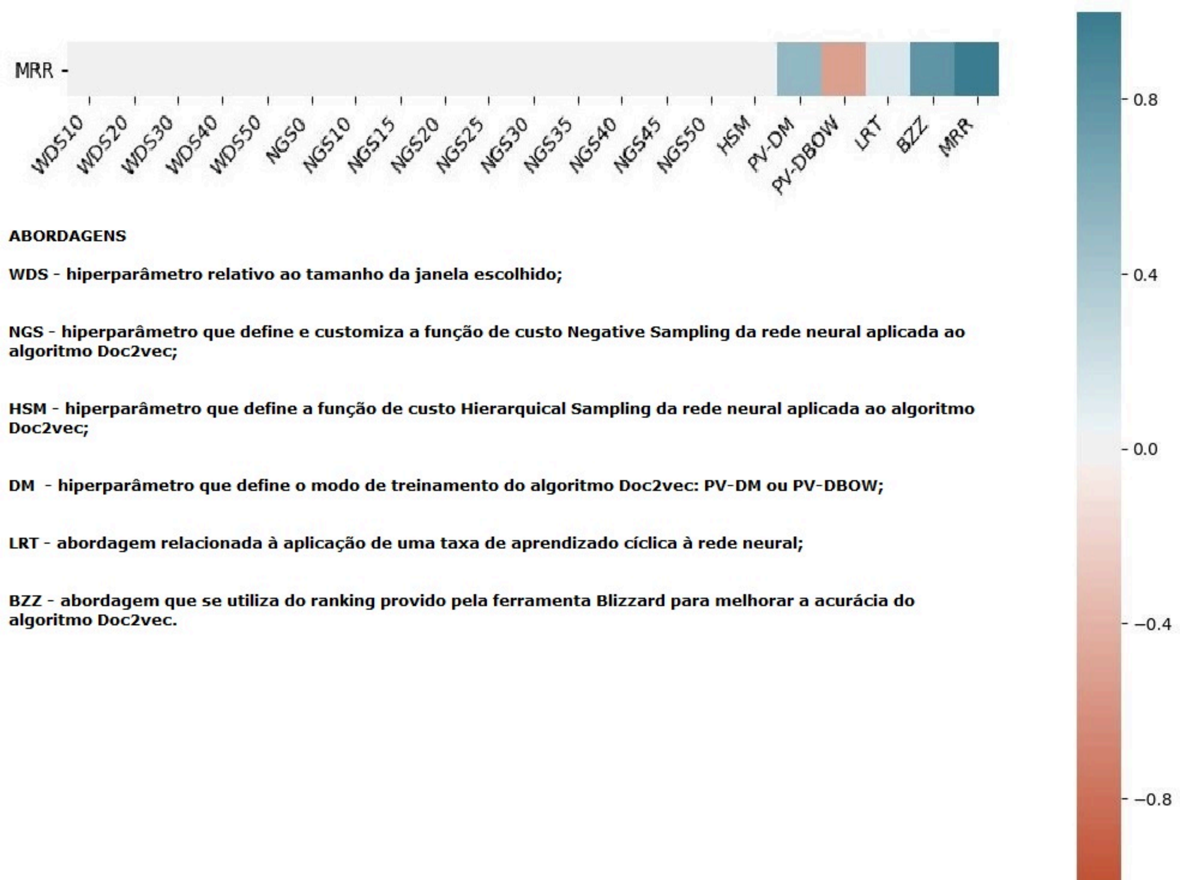
Figura 15 – Melhores e piores resultados para o teste de Massa



Com base na observação da Tabela 8, observa-se que usar um tamanho de janela maior ou igual a 40 é sempre vantajoso para os *software* em estudo, exceto o *eclipse.jdt.debug* cuja melhor configuração utilizou um tamanho igual a 20. Lau e Baldwin[106] argumenta que alguns modelos baseados no uso de *Doc2vec* podem precisar de um tamanho de janela maior. No experimento eles chegam a utilizar o parâmetro WDS com um valor igual a 15. Usar a função de custo NGS também se sobressai sobre a utilização da função HSM. Por unanimidade, a estratégia PV-DM sempre aparece nas melhores configurações testadas para todos os *software*. O gráfico da Figura 15 sintetiza os resultados da Tabela 8.

Para validar se os melhores e piores resultados obtidos retratam a relação entre cada variável e o valor do MRR passa-se agora à análise do gráfico da Figura 16. Uma cor mais

Figura 16 – Correlação entre abordagens e hiperparâmetros no teste de 720 combinações



azul significa correlação positiva e vermelha, correlação negativa. Quanto mais forte a cor maior o grau de correlação. Todas as configurações discretas utilizadas em comparação com os valores de MRR obtidos no teste de massa para o conjunto todos os 6 *software* foram usados. Desta forma, retratou-se o resultado de 4320 testes na tentativa de extrair padrões emergentes de correlação entre os valores de MRR e as demais variáveis. Importante salientar que os valores de MRR estão diretamente relacionados à acurácia do modelo. A análise do gráfico nos permite visualizar que um valor mais alto de MRR está associado à utilização da política de treinamento PV-DM, isto é, PV-DM=1. Outros hiperparâmetros como WDS, NGS, HSM e PV-DBOW não se mostraram relevantes quando usando classes em um contexto de LBS. O mesmo não pode ser afirmadado para o hiperparâmetro NGS quando usado para LCS, conforme visto no Capítulo 3 em que uma calibração de NGS=50 se sobressai frente à calibração padrão proposta pela literatura NGS=5. Isto responde à questão de pesquisa Q1 que se refere a influência dos hiperparâmetros sobre a acurácia do processo de LBS. De maneira similar, também é sempre vantajoso as políticas LRT e BZZ. Estas duas vertentes não estão diretamente associadas aos hiperparâmetros e figuram como resposta à questão de pesquisa Q2.

O modo de treinamento PV-DM funciona da mesma forma que a estratégia CBOW

(continuous bag-of-words) do algoritmo *Word2vec*[8]. CBOW é diversas vezes mais rápido para treinamento do que o modo *skip-gram* e sua acurácia na geração dos modelos de aprendizado de máquina é sensível positivamente a contextos em que palavras mais frequentes são observadas[9]. Esta tendência parece se acentuar quando o corpus tratado é o código fonte de programas. Para Babii, Janes e Robbes[107] a natureza do código fonte é muito repetitiva. Esta visão se confirma nas visões de Gabel e Su[108] e Hindle et al.[19].

Não foi possível superar os resultados obtidos a partir do Estudo de Referência que utiliza a ferramenta Blizzard. Uma alternativa para esta situação refere-se ao enriquecimento semântico com a utilização de outros contextos para geração dos vetores. O algoritmo *Doc2vec* é sensível à quantidade de dados provida para treinamento da Rede. Constatações semelhantes foram feitas por [30]. Tem-se a convicção de que tenha acontecido um processo de *"underfitting"*, isto é, os dados/contextos não foram suficientes para treinamento do modelo. Portanto o mesmo apresentou uma performance subótima. Em contra-posição , Dusserre e Padro[109] defende que as fontes de informações devem ser rigorosamente selecionadas. A especificidade do corpus usado para treinamento é mais importante, na melhoria da performance dos algoritmos *Word2vec* e *Doc2vec*, do que a quantidade de informação [64]. Neste sentido, Milanova et al.[87] relata um modelo de baixa qualidade, utilizando vetores de palavras treinadas em um domínio do conhecimento quando aplicado a outro domínio. Corroborando com esta constatação, um relato de baixo desempenho foi relatado no Capítulo 5, Seção 5.1, pelo autor do presente trabalho quando do treinamento adicional de vetores. Neste treinamento, *software* escritos em Java, mas sem nenhum critério de escolha foram utilizados. Por outro lado, o Apêndice D descreve um estudo preliminar, realizado com êxito, com *software* do mesmo domínio de problema.

4.6.2 Usando métodos para Localização de *Bugs*

Em outra tentativa para melhoria da acurácia, novos testes foram realizados, sendo a granularidade das entidades buscadas no código fonte diminuída de classes para métodos. Nos pré-testes aplicou-se diretamente os melhores valores discretos dos hiperparâmetros obtidos no experimento de LCS para guiar a escolha do tamanho do vetor de parágrafos nos testes de massa. Intuitivamente, fez-se isto, pois tratavam-se de tarefas similares em um mesmo domínio de problema: a manutenção de *software*. Estes testes envolvendo classes do código fonte em um contexto de LBS para as abordagens NGS e LRT estão disponíveis no Apêndice B. Até o presente momento já foram expostos os experimentos envolvendo os hiperparâmetros e abordagens: DM e BZZ. A partir de agora passa-se ao relato de experimentos em que foi realizada a segmentação do código das classes em

[8] https://towardsdatascience.com/understand-how-to-transfer-your-paragraph-to-vector-by-doc2vec-1e225ccf102

[9] https://towardsdatascience.com/nlp-101-word2vec-skip-gram-and-cbow-93512ee24314

métodos e procedeu-se a ordenação de tais métodos. Tratar métodos ao invés de classes é apontado por Tantithamthavorn, Ihara e Hata[104] como mais eficiente do que classes para localização da primeira entidade de código fonte relevante à consulta.

Passa-se então a relatar os resultados expondo os valores de MRR obtidos para classes e métodos. Estes resultados foram listados na Tabela 9 para avaliação do leitor que pode assim evidenciar se a utilização de métodos como unidades granulares permite o aumento de acurácia. A Tabela também traz os resultados obtidos no estudo de Referência para comparação. Conforme pode ser visto, em nenhum dos *software* a utilização de métodos

Tabela 9 – Comparativo entre LBS usando métodos e classes

Software	WDS	NGS	HSM	DM	LRT	BZZ	Gran.	MRR
ecf (Blizzard)	-	-	-	-	-	Sim	Classe	0.5426
ecf	40	45	Não	PV-DM	Sim	Sim	Classe	0.4634
ecf	40	45	Não	PV-DM	Sim	Sim	Método	0.4233
eclipse.jdt.core (Blizzard)	-	-	-	-	-	Sim	Classe	0.5537
eclipse.jdt.core	40	20	Não	PV-DM	Sim	Sim	Classe	0.4606
eclipse.jdt.core	40	20	Não	PV-DM	Sim	Sim	Método	0.4140
eclipse.jdt.debug (Blizzard)	-	-	-	-	-	Sim	Classe	0.4616
eclipse.jdt.debug	20	Não	Sim	PV-DM	Sim	Sim	Classe	0.3846
eclipse.jdt.debug	20	Não	Sim	PV-DM	Sim	Sim	Método	
eclipse.jdt.ui (Blizzard)	-	-	-	-	-	Sim	Classe	0.4801
eclipse.jdt.ui	40	10	Não	PV-DM	Sim	Sim	Classe	0.4208
eclipse.jdt.ui	40	10	Não	PV-DM	Sim	Sim	Método	0.4078
eclipse.pde.ui (Blizzard)	-	-	-	-	-	Sim	Classe	0.4143
eclipse.pde.ui	40	10	Não	PV-DM	Sim	Sim	Classe	0.3645
eclipse.pde.ui	40	10	Não	PV-DM	Sim	Sim	Método	0.3436
tomcat70 (Blizzard)	-	-	-	-	-	Sim	Classe	0.5999
tomcat70	40	10	Não	PV-DM	Sim	Sim	Classe	0.4579
tomcat70	40	10	Não	PV-DM	Sim	Sim	Método	0.4515

como unidade granular no processo de LBS apresentou resultado superior aquele obtido utilizando classes. Assim, o Estudo de Referência não foi superado. Isto instigou o pesquisador na busca por novas alternativas para o algoritmo *Doc2vec* em sua tarefa de LBS. Assim, novas abordagens foram tentadas. Como forma de orientar o desenvolvedor, um conjunto de tentativas não exitosas ou que mereçam melhor tratamento foram expostas no Capítulo 5. Estas abordagens podem figurar como ponto inicial de onde ou não começar novos experimentos.

4.7 Resumo do Capítulo

Este capítulo apresentou uma aplicação do algoritmo *Doc2vec* na LBS sob o código fonte de 6 sistemas orientados a objeto. Obteve-se êxito em melhorar a acurácia do algoritmo *Doc2vec* ao descobrir empiricamente novas calibrações para os hiperparâmetros

do algoritmo. Este ganho se mostrou relevante também quando o algoritmo *Doc2vec* foi combinado com recentes abordagens que expressam o estado da arte.

Contudo, não foi possível superar o estado da arte, aqui representado pela ferramenta *Blizzard*, disponível no Estudo de Referência. Mesmo assim, ficou evidenciado que a correta seleção e calibração dos hiperparâmetros do algoritmo *Doc2vec* para localização de *bugs* pode ser melhorada e a combinação de técnicas pode trazer melhorias para o processo de LBS.

As abordagens apresentadas para melhoria do processo de localização de *bugs* usando *Doc2vec*, com granularidade em nível de classes e métodos, foram:

1. Uma combinação entre algoritmos (*Doc2vec*, *Blizzard*) que resultou em ganho de acurácia para o algoritmo *Doc2vec*;

2. Um teste de massa capaz de avaliar como os hiperparâmetros do algoritmo; *Doc2vec* influem sobre a acurácia do processo de LBS com recomendação de valores e configurações para o desenvolvedor interessado em utilizar o algoritmo; *Doc2vec*.

No Capítulo 5 será apresentado um conjunto de abordagens que não apresentaram resultados satisfatórios. Assim espera-se que o desenvolvedor possa se orientar quanto à viabilidade de implementação de tais propostas.

Capítulo 5

Resultados Negativos e Implicações

Este capítulo retrata uma série de estudos que foram realizados para Localização de Características de *Software* (LCS) e Localização de *Bugs* de *Software* (LBS). O objetivo do capítulo é apresentar um conjunto de alternativas estudadas, que não apresentaram um resultado plenamente satisfatório, ou mesmo que ainda precisam de algum tratamento diferenciado para serem efetivas. O propósito de incluir este capítulo neste trabalho é similar à inclusão de trilhas RENE (**RE**producibilidade e Resultados **NE**gativos) em conferências de Engenharia de *Software*. Muitas vezes a apresentação de resultados negativos pode ser útil para demonstrar a falta de **efeito** de uma determinada abordagem sob determinadas condições.

Na tentativa de aumentar a acurácia dos processos de LCS e LBS, este capítulo propôs o enriquecimento de vetores de palavras e parágrafos com base em *software* escritos na linguagem de Programação Java. Em outra tentativa, um mecanismo de penalização para classes pouco coesas foi implementado no sentido de avaliar como a alteração da implementação de tais classes impacta a acurácia. Avaliou-se também a criação de vetores de parágrafos levando-se em conta a classe sintática de cada palavra constituinte do parágrafo. Por fim, de maneira sucinta foram listadas outras tentativas não exitosas que foram realizadas e merecem ser melhor analisadas e evoluídas. Desta forma, espera-se que o leitor possa estabelecer uma fronteira entre abordagens efetivas e aquelas não apropriadas sob as condições apresentadas para LCS e LBS.

Este capítulo faz referência a outras abordagens e hiperparâmetros estudados anteriormente. Assim para facilitar o entendimento do leitor foram listadas as siglas reutilizadas e seus respectivos significados:

1. WDS - hiperparâmetro relativo ao tamanho da janela escolhido;

2. NGS - hiperparâmetro que define e customiza a função de custo *Negative Sampling* da rede neural aplicada ao algoritmo *Doc2vec*;

3. HSM - hiperparâmetro que define a função de custo *Hierarquical Sampling* da rede neural aplicada ao algoritmo *Doc2vec*;

4. DM - hiperparâmetro que define o modo de treinamento do algoritmo *Doc2vec*: PV-DM ou PV-DBOW;

5. LRT - abordagem relacionada à aplicação de uma taxa de aprendizado cíclica à rede neural;

6. BZZ - abordagem que se utiliza do ranking provido pela ferramenta Blizzard para melhorar a acurácia do algoritmo *Doc2vec*.

5.1 Estudo 1 - O uso de Enriquecimento Semântico (FED)

Esta abordagem buscou avaliar o efeito de se enriquecer vetores pré-treinados em um código fonte com outros contextos textuais oriundos do código fonte de outras aplicações escritas na linguagem de programação Java. O contexto e *dataset* escolhidos são os mesmos usados para o experimento de LBS, descrito no Capítulo 4. Assim, esperava-se que a acurácia do processo de LBS pudesse aumentar a partir do provimento de contextos adicionais em que novos padrões de uso de palavras e parágrafos pudessem ser evidenciados. Neste trabalho, esta abordagem recebeu a denominação FED, dada sua analogia com o processo de alimentação, neste caso com novos contextos.

Muitos trabalhos tem se dedicado à estratégia de melhorar a qualidade dos vetores de palavras e parágrafos gerados. A partir de um conjunto de informações adicionais procura-se melhorar a qualidade destes vetores gerados previamente em um corpus específico [110, 111, 112]. Contudo, a eficácia de tais metodologias, quando aplicado ao processo de LCS, não está totalmente clara. Desta forma, a hipótese levantada é que o enriquecimento supracitado seja capaz diminuir o custo de manutenção relativo à LCS ao ajustar os valores numéricos dos vetores representativos de classes do código fonte.

5.1.1 Materiais e Métodos

Os passos executados para esta abordagem podem ser assim enumerados:

1. Separação de um *software* orientado a objetos com suas respectivas descrições de *bugs*;

2. Criação do *Corpus* - Separação do código fonte do *software* orientado a objetos na granularidade de classes;

3. Pré-processamento do código fonte - Atividade que visa remover elementos com baixo valor semântico como dígitos, símbolos e *stop words*;

4. Extração de palavras e contextos do *corpus*;

5. Treinamento do modelo com geração de vetores de parágrafos e palavras com base no *software* relacionado ao *bug*;

6. Extração de novos contextos a partir dos outros *software* escritos na linguagem de programação Java que compõem o *dataset*;

7. Enriquecimento dos vetores a partir da continuidade do treinamento do modelo gerado pelo algoritmo *Doc2vec*;

8. Cálculo de similaridades entre descrições de *bug* e suas respectivas classes do código fonte com geração de *ranking*;

9. Cálculo do MRR.

A calibração de hiperparâmetros utilizada para este experimento envolveu os melhores resultados obtidos, para cada *software*, no teste de massa executado no Capítulo 4. Foram usadas um total de 50 épocas de treinamento. Para continuidade do treinamento com base em outros contextos utilizou-se mais 50 épocas. Testes discretos com outros números de épocas (10,20,500) foram realizados sem melhoria aparente da acurácia. Neste trabalho, esta continuidade do treinamento foi denominada enriquecimento semântico (FED). Está-se "alimentando" vetores previamente treinados com novas informações (contextos) para que se ajustem melhor ao que realmente representam. Os novos padrões de coocorrência de palavras será detectado nos novos contextos e reforçarão ou alterarão os já existentes no modelo.

5.1.2 Resultados e Discussão

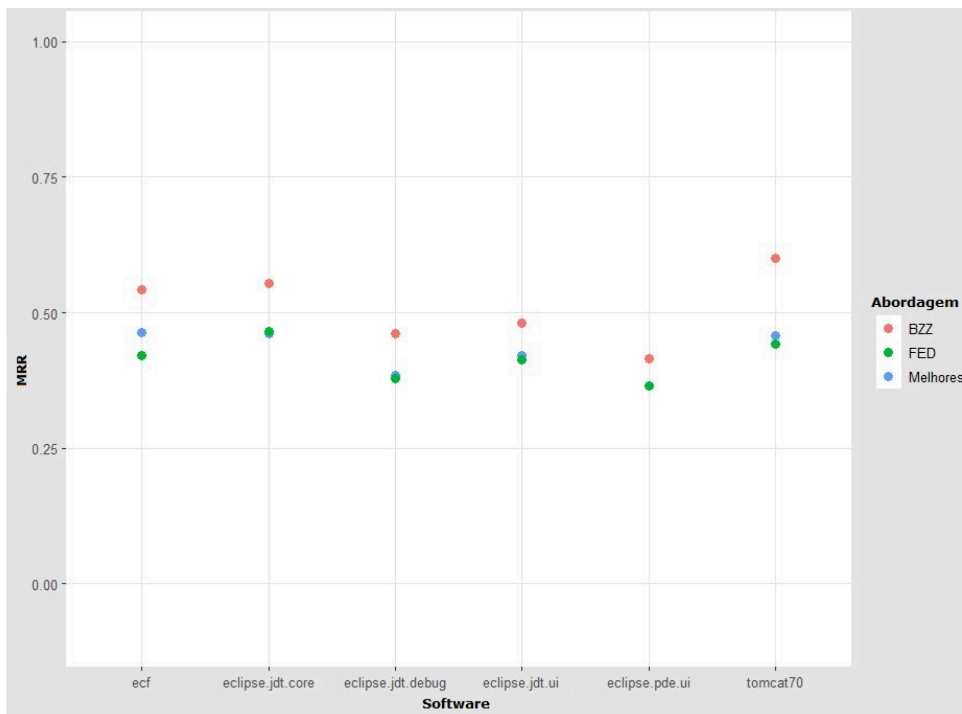
Na Tabela 10 foram expostos os resultados para a abordagem FED. Optou-se também por expor os resultados oriundos do Estudo de Referência e as melhores combinações obtidas a partir do teste de massa realizado no Capítulo 4. Assim, o leitor pode realizar uma comparação da performance da abordagem FED frente a outras abordagens.

A escolha de quais aplicações são relevantes para promover a melhoria dos vetores, e, consequentemente impactar a acurácia do processo de LBS, nem sempre é óbvia. Escolhas ruins podem levar o desenvolvedor a uma baixa performance como observado no experimento desta seção. Assim, ao se observar a Tabela 10, não existe diferença significativa entre a abordagem FED e as melhores combinações. Além disso, foi utilizada a melhor calibração durante a abordagem FED. Por outro lado, a abordagem FED não supera os resultados obtidos com a ferramenta *Blizzard*. Uma alternativa à utilização sem critério de aplicações escritas em Java poderia ser a utilização daquelas que estejam em um mesmo domínio específico de conhecimento da aplicação alvo. Um estudo preliminar com *software* do mesmo domínio foi exposto no Apêndice D e se mostrou promissor. Alguns

Tabela 10 – *R*esultados de MRR para cada *software* - Abordagem FED

Software	WDS	NGS	HSM	DM	LRT	BZZ	MRR
Blizzard							
ecf	-	-	-	-	-	Sim	0.5426
eclipse.jdt.core	-	-	-	-	-	Sim	0.5537
eclipse.jdt.debug	-	-	-	-	-	Sim	0.4616
eclipse.jdt.ui	-	-	-	-	-	Sim	0.4801
eclipse.pde.ui	-	-	-	-	-	Sim	0.4143
tomcat70	-	-	-	-	-	Sim	0.5999
Melhores Resultados							
ecf	40	45	Não	PV-DM	Sim	Sim	0.4634
eclipse.jdt.core	40	20	Não	PV-DM	Sim	Sim	0.4606
eclipse.jdt.debug	20	Não	Sim	PV-DM	Sim	Sim	0.3846
eclipse.jdt.ui	40	10	Não	PV-DM	Sim	Sim	0.4208
eclipse.pde.ui	50	10	Não	PV-DM	Sim	Sim	0.3645
tomcat70	50	40	Não	PV-DM	Sim	Sim	0.4579
Abordagem FED							
ecf	40	45	Não	PV-DM	Sim	Sim	0.4213
eclipse.jdt.core	40	20	Não	PV-DM	Sim	Sim	0.4653
eclipse.jdt.debug	20	Não	Sim	PV-DM	Sim	Sim	0.3784
eclipse.jdt.ui	40	10	Não	PV-DM	Sim	Sim	0.4129
eclipse.pde.ui	50	10	Não	PV-DM	Sim	Sim	0.3652
tomcat70	50	40	Não	PV-DM	Sim	Sim	0.4422

Figura 17 – Abordagem FED aplicada à Localização de *Bugs de Sofware*



estudos, ainda não utilizados no contexto de LCS e LBS, trazem esta mesma recomendação [109, 87, 64]. O gráfico da Figura 17 sintetiza a Tabela 10. Nele está estabelecida

relação MRR X *Software* para que o leitor possa ter uma visão das performances de cada abordagem citada.

5.2 Estudo 2 - Atribuição de pesos distintos aos elementos sintáticos do parágrafo (PDP)

Esta abordagem visou avaliar o peso sintático de cada palavra usada na construção de trechos do código fonte sobre o processo de LBS. Esta abordagem foi nomeada PDP. Assim foi considerado a categorização de palavras de acordo com suas funções como parte-do-discurso. As categorias linguísticas consideradas foram os substantivos (nomes), verbos e adjetivos. A natureza sintática das palavras dentro do código tem despertado o interesse da comunidade de Engenharia de *Software* em tarefas como classificação dos termos com base nos contextos em que estes aparecem dentro do código fonte [113].

5.2.1 Materiais e Métodos

Para este trabalho foi utilizado uma ferramenta de *Part-Of-Speech Tagger ((POS Tagger))*. Um *Part-Of-Speech Tagger* lê texto em alguma linguagem e associa *tags* aos elementos constituintes do discurso[1]. A Ferramenta utilizada foi NLTK (Natural Language Toolkit)[2] disponível para a linguagem de Programação Python. Nesta abordagem foi utilizado o algoritmo *Word2vec* visto que é gerado de forma separada a partir da combinação dos vetores individuais de palavras.

O processo de formação do vetor de parágrafo a ser utilizado para descrições de *bugs* (consulta) e classes (parágrafo) do código fonte está baseado no trabalho de Nagoudi, Ferrero e Schwab[89], e é regido pela seguinte Equação 5:

$$\begin{cases} V_{\text{paragrafo}} = \sum_{k=1}^{i} idf(w_{\text{k}}) * Pos_weight(Pos_{w_{\text{k}}}) * v_{\text{k}} \\ V_{\text{consulta}} = \sum_{k=1}^{j} idf(w'_{\text{k}}) * Pos_weight(Pos_{w_{\text{k}}'}) * v'_{\text{k}} \end{cases} \quad (5)$$

Em que para uma palavra w,
idf - inverso da frequência nos documentos;
Pos_weight($\text{Pos}_{\text{w}_\text{k}}$) * v_k) - peso atribuído à palavra conforme a *tag* por ela recebida do *POS Tagger*; v_k - vetor da palavra pertencente ao código da classe ou descrição de *bug*;

Os passos executados para esta abordagem podem ser assim enumerados:

1. Geração dos vetores de palavra com base no código fonte do *software* em que se deseja localizar o *bug*;

[1] https://nlp.stanford.edu/software/tagger.shtml
[2] https://www.nltk.org/

2. Cálculo do vetor de parágrafo para cada classe e consulta conforme a Equação 5;

3. Cálculo de similaridade entre o vetor da consulta e todas as classes;

4. ranqueamento de classes por ordem de maior valor de cosseno.

A Listagem 5.1 permite ao leitor como foi realizada a distribuição de pesos no experimento a cada palavra que compõe um parágrafo. Na linha 3 atribuiu-se um valor alto (0.9) para nomes; e um valor mais baixo para nomes próprios conforme a linha 5 (0.1). Um valor mediano (0.5) foi atribuído para verbos, vide linha 7. Por fim, os adjetivos também receberam um baixo peso (0.1), conforme exposto na linha 9. Nagoudi, Ferrero e Schwab[89] propõem originalmente 0.5, 0.4 e 0.1 respectivamente para nomes, verbos e adjetivos. Tal configuração foi testada também sem sucesso.

Listagem 5.1 – Exemplo de tags utilizadas no NLTK

```
1 pos_w=0.0
2         if(tipo=='NN' or tipo=='NNS'):
3             pos_w=0.9
4         if (tipo=='NNP'):
5             pos_w=0.1
6         if (tipo=='VB' or tipo=='VBP' or tipo=='VBN' or tipo=='VBG' or tipo=='VBD'):
7             pos_w=0.5
8         if (tipo == 'JJ'):
9             pos_w=0.1
```

Listagem 5.2 – Formação de vetor de parágrafo

```
1     for palavra in lista_palavras:
2         vetor_palavra = idf*pos_w*(WvModel[palavra])
3         vector_paragrafo = np.add(vector_paragrafo,vetor_palavra)
```

Por outro lado, a formação do vetor de parágrafo obedece a lógica expressa na Listagem 5.2. Para cada palavra do parágrafo/consulta, realiza-se o cálculo referente à Equação 5. O vetor de palavra assim calculado é adicionado aos outros vetores de palavras já adicionados à variável vetor_paragrafo, conforme pode ser visto na linha 3. Toda a metodologia, com fórmulas e maior detalhamento pode ser encontrado em [89].

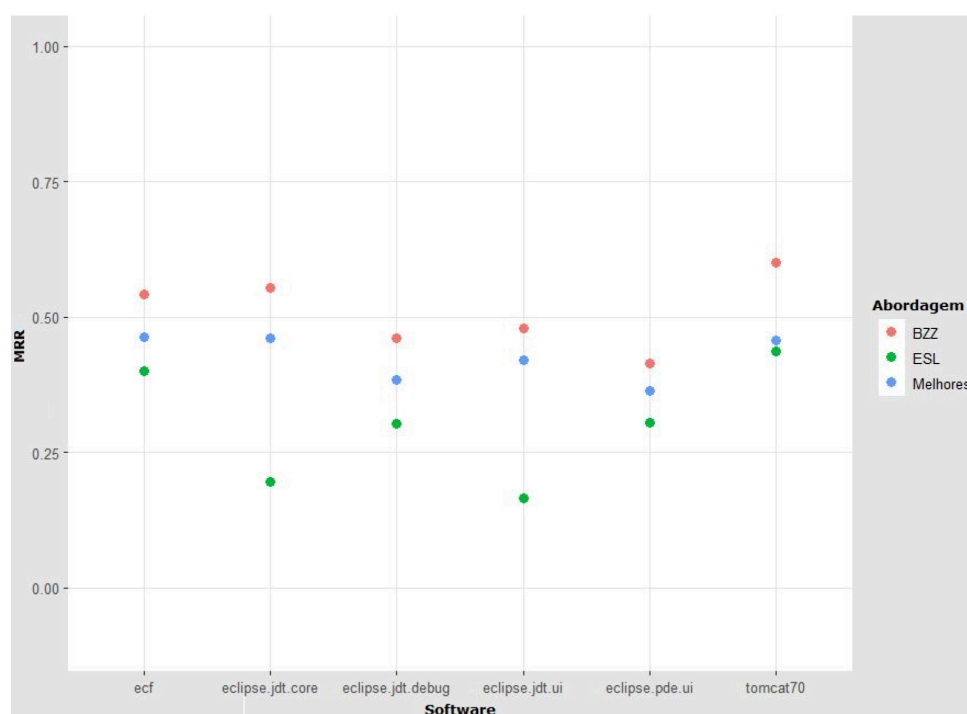
5.2.2 Resultados e Discussão

Como pode ser visto, os resultados indicam que o uso de recursos de PLN como o *POS Tagger*, como implementado nesta seção, não é adequado para a melhora da acurácia do processo de LBS. Todos os resultados, para os 6 *software*, alcançaram resultado inferior aqueles obtidos a partir da melhor calibração de parâmetros, mesmo que usando esta última. Apesar do resultado ruim, isto esta em conformidade com que afirma Alsuhaibani et al.[113]. Segundo os autores, informações relevantes que podem ser abstraídas do

Tabela 11 – Resultados de MRR para cada *software* - Abordagem ESL

Software	WDS	NGS	HSM	DM	LRT	BZZ	MRR
Blizzard							
ecf	-	-	-	-	-	Sim	0.5426
eclipse.jdt.core	-	-	-	-	-	Sim	0.5537
eclipse.jdt.debug	-	-	-	-	-	Sim	0.4616
eclipse.jdt.ui	-	-	-	-	-	Sim	0.4801
eclipse.pde.ui	-	-	-	-	-	Sim	0.4143
tomcat70	-	-	-	-	-	Sim	0.5999
Melhores Resultados							
ecf	40	45	Não	PV-DM	Sim	Sim	0.4634
eclipse.jdt.core	40	20	Não	PV-DM	Sim	Sim	0.4606
eclipse.jdt.debug	20	Não	Sim	PV-DM	Sim	Sim	0.3846
eclipse.jdt.ui	40	10	Não	PV-DM	Sim	Sim	0.4208
eclipse.pde.ui	50	10	Não	PV-DM	Sim	Sim	0.3645
tomcat70	50	40	Não	PV-DM	Sim	Sim	0.4579
Abordagem ESL							
ecf	40	45	Não	PV-DM	Sim	Sim	0.4000
eclipse.jdt.core	40	20	Não	PV-DM	Sim	Sim	0.1958
eclipse.jdt.debug	20	Não	Sim	PV-DM	Sim	Sim	0.3027
eclipse.jdt.ui	40	10	Não	PV-DM	Sim	Sim	0.1645
eclipse.pde.ui	50	10	Não	PV-DM	Sim	Sim	0.3050
tomcat70	50	40	Não	PV-DM	Sim	Sim	0.4359

Figura 18 – Abordagem ESL aplicada à Localização de *Bugs de Sofware*



contexto em que as palavras ocorrem no código fonte são perdidas. Isto se deve ao fato da técnica de PLN empregada analisar o termo somente no escopo da sentença em que

ele ocorre. Informações do seu posicionamento global em relação a todo o *software* são perdidas nesta abordagem simplista.O gráfico da Figura 18 sintetiza a Tabela 11. Nele está estabelecida relação MRR X *Software* para que o leitor possa ter uma visão das performances de cada abordagem citada.

5.3 Estudo 3 - Uso do *fastText* (FST)

O problema tratado nesta seção é a baixa acurácia do processo de LCS tratado no Capítulo 3. Até o momento neste trabalho, sempre foram utilizados os algoritmos *Doc2vec* ou *Word2vec* para geração dos vetores de parágrafos em processsos de LCS. Assim, esta seção propôs utilizar outro mecanismo de geração a partir da utilização da ferramenta *fastText*.

5.3.1 Materiais e Métodos

Os passos seguidos para implementação desta abordagem podem ser enumerados:

1. Seleção e *parsing* de descrições de características;
2. Seleção e *parsing* dos métodos do código fonte;
3. Treinamento do modelo *fastText*;
4. Formação dos vetores de parágrafos (classes e consulta) a partir da soma dos vetores de palavras individuais;
5. Cálculo e ordenação de similaridades entre consulta e classes.

Listagem 5.3 – Treinamento de vetores usando o algoritmo fastText

```
1 fasttextModel = gensim.models.FastText(size=tamanho_vetor, window=5, workers=8)
2 fasttextModel.build_vocab(metodos)
3 fasttextModel.train(metodos, total_examples=fasttextModel.corpus_count, epochs=50)
```

A Listagem 5.3 ilustra os passos adotados para criação e treinamento do modelo usando o algoritmo *fastText*.

5.3.2 Resultados e Discussão

Conforme a Tabela 12, FST obteve sempre resultados inferiores à abordagem DV** + LRT + NGS50 que representa um novo estado da arte conforme publicado em [101].

O *fastText* mostrou-se inferior ao algoritmo *Word2vec* em um conjunto de experimentos utilizando um conjunto de dados utilizando a língua Inglesa[3]. Trata-se de dados extraídos

[3] http://blog.conceptnet.io/posts/2017/how-luminoso-made-conceptnet-into-the-best-word-vectors-and-won-at-semeval/

Tabela 12 – Comparação do MRR entre abordagens DV*, DV**+LRT+NGS50 e FST

Software	Abordagens/K	100	200	300	400	500
ArgoUML V0.22	DV*	0.0775	0.0570	0.0625	0.0587	0.0601
	DV**+LRT+NGS50	**0.1762**(σ=0.0213)	**0.1804**(σ=0.0158)	**0.1757**(σ=0.0144)	**0.1803**(σ=0.0132)	**0.1799**(σ=0.0126)
	FST	0.0699	0.0798	0.0643	0.0976	0.0773
ArgoUMLV0.24	DV*	0.0827	0.0906	0.0874	0.0691	0.0942
	DV**+LRT+NGS50	**0.1337**(σ=0.0079)	**0.1283**(σ=0.0088)	**0.1293**(σ=0.0082)	**0.1307**(σ=0.008)	**0.1305**(σ=0.0081)
	FST	0.0836	0.0906	0.0796	0.1055	0.1129
ArgoUML V0.26.2	DV*	0.0847	0.0890	0.0813	0.0834	0.0805
	DV**+LRT+NGS50	**0.1332**(σ=0.0047)	**0.1337**(σ=0.0065)	**0.1352**(σ=0.0067)	**0.1339**(σ=0.0062)	**0.1389**(σ=0.0073)
	FST	0.0818	0.0669	0.0740	0.0649	0.0816
JabRef V2.6	DV*	0.0450	0.0373	0.0455	0.0382	0.0428
	DV**+LRT+NGS50	**0.0765**(σ=0.0168)	**0.0866**(σ=0.0178)	0.0731(σ=0.0163)	0.0734(σ=0.0162)	**0.0742**(σ=0.0163)
	FST	0.0316	0.0296	0.0291	0.0293	0.0285
jEdit V4.3	DV*	0.0872	0.0791	0.0825	0.0814	0.0679
	DV**+LRT+NGS50	**0.1805**(σ=0.0088)	**0.172**(σ=0.0096)	**0.1731**(σ=0.0092)	**0.1715**(σ=0.0091)	**0.1752**(σ=0.0091)
	FST	0.0389	0.0429	0.0443	0.0447	0.0408
muCommander V0.8.5	DV*	0.0652	0.0623	0.0703	0.0606	0.0538
	DV**+LRT+NGS50	**0.1605**(σ=0.0118)	**0.1732**(σ=0.0142)	**0.1629**(σ=0.0122)	**0.1727**(σ=0.0134)	**0.1708**(σ=0.0127)
	FST	0.0638	0.0803	0.0658	0.0697	0.0876

do *SemEval 2017 Task 2*[4], um dos maiores desafios mundiais para geração de *benchmarks* para avaliação de técnicas de similaridade semântica usando vetores de palavras. Esta mesma tendência foi observada em estudos comparativos entre modelos de geração de vetores de palavras [65].

Desta forma, a partir da análise dos resultados expostos nesta seção, a superioridade do algoritmo *Word2vec* sobre o *fastText*, no quesito similaridade de documentos, se confirmou agora em um contexto de LBS.

5.4 Resumo do Capítulo

Neste capítulo foram relatados vários experimentos que não obtiveram êxito na melhoria da acurácia para os problemas de LCS e LBS. As abordagens testadas foram enumeradas na Tabela 13.

Tabela 13 – *A*bordagens testadas sem sucesso

*E*studo	Sigla	Descrição da abordagem	Aplicação	Capítulo Referência
Estudo 1	FED	Abordagem voltada ao enriquecimento semântico de vetores de palavras e documentos	LBS	4
Estudo 2	PDP	Utilização Atribuição de pesos distintos aos elementos sintáticos do parágrafo	LBS	4
Estudo 3	FST	Utilização da ferramenta *fastText* como mecanismo gerador de vetores de parágrafos	LCS	3

Em todos os testes para LBS foram usadas as calibrações apontadas pelo teste de massa realizado no Capítulo 4. As exceções se deram nos experimentos FST e PFR que

[4] https://www.aclweb.org/portal/content/semeval-2017-task-2-monolingual-and-cross-lingual-word-similarity

estavam relacionados a LCS. Nenhuma das abordagens expostas neste capítulo conseguiu superar as técnicas relacionadas ao estado da arte apontadas neste trabalho.

Capítulo 6

Trabalhos Relacionados

Tarefas de PLN tem se desenvolvido com o apoio de um conjunto de técnicas como *LSI, LDA, Glove, FastText, Word2vec* e *Doc2vec*. Por um longo tempo, as principais técnicas aplicadas para recuperação de rastreabilidade entre artefatos foram LSI e LDA, mas seus protagonismos tem dado lugar aos modelos baseados em RNAs. Como vimos no Capítulo 2, tais técnicas são baseadas em modelos algébricos lineares que utilizam a frequência de termos em matrizes esparsas para representação de documentos textuais. Outros modelos, como por exemplo, o VSM ou a regressão logística também tem sido gradativamente substituídos por modelos mais compactos e mais precisos baseados em RNAs [114]. Esta capítulo objetiva expor e discutir os principais trabalhos relacionados à LCS e LBS, baseados em técnicas baseadas em frequência de termos e nas abordagens baseadas em RNAs.

6.1 Localização de Características usando o *grep*

Existem vários métodos para identificação de conceitos e características no código fonte. O mais intuitivo e amplamente utilizado meio consiste na realização de uma busca textual dentro do código fonte buscando pelo casamento de termos de interesse do desenvolvedor [34]. Este padrão de busca se identifica com o *modus operandi* do utilitário *grep* do sistema operacional Unix e um conjunto de ferramentas que trabalham de maneira semelhante tem sido amplamente utilizados pelos desenvolvedores.

Cada trecho de código retornado por uma pesquisa usando o *grep*[1] necessita ser analisado para que novas consultas sejam geradas no intuito de localizar mais variáveis, funções e comentários. Tripathy e Naik[34] apontam as seguintes deficiências neste tipo de técnica:

1. Ele é baseado na concepção e nomenclatura que o desenvolvedor tem de conceitos que estão no código fonte;

2. As técnicas falham quando os conceitos estão ocultos no código fonte.

[1] http://www.gnu.org/software/grep/

3. Se o desenvolvedor sugerir termos incorretos para localização de identificadores, o processo também ficará comprometido. Isto implica que o desenvolvedor tenha que ter um conhecimento prévio do *software* ou domínio de problema para construir consultas relevantes.

Para Dit et al.[6], em trabalhos envolvendo a LCS, técnicas baseadas em *"grep"* tem sido utilizadas como *baseline* de comparação. Assim Bohnet, Voigt e Doellner[36], descrevem a inviabilidade de se promover a LCS em grandes e complexos *software* escritos em C++, a partir da utilização prática de técnicas baseadas em *grep*. Segundo os autores, o número de elementos de código retornado é muito grande, o que dificulta a compreensão e análise. Assim propõem uma ferramenta baseada em análise dinâmica em que, a partir dos rastros de execução de características, o desenvolvedor pode entender como as mesmas são implementadas no código fonte do *software*.

6.2 Localização de Características usando *Latent Semantic Indexing* e *PROMESIR*

Marcus et al.[9] mostraram a efetividade de se utilizar LSI para LCS. O objetivo pretendido era mapear conceitos expressos em linguagem natural pelo desenvolvedor para partes dos código fonte que os implementassem. Para isto eles realizaram uma análise estática do código fonte do *software* NCSA Mosaic, um antigo navegador *web* escrito em C. Eles demonstraram superioridade do método que se utilizava de LSI em relação às técnicas baseadas em *grep*.

Como uma evolução da utilização da técnica LSI em LCS, Poshyvanyk et al.[14] propõem a técnica *PROMESIR*, uma combinação de LSI [9] com um gerador de *rankings* probabilísticos [115, 116]. Este gerador tem seu núcleo baseado na execução de cenários em uma análise dinâmica (*Scenario based Probabilistic Ranking* (SPR)). As duas técnicas ranqueiam elementos do código fonte de acordo com a relevância em relação à uma descrição de característica. Os *rankings* são então combinados para produzir o *rank* final dentro da técnica PROMESIR proposta. Uma avaliação baseada na análise dos *software* Eclipse e Mozzila indicaram a superioridade da combinação em relação ao uso individual das técnicas LSI e SPR. Resultado semelhante aconteceu neste trabalho. A partir da combinação de abordagens diferentes foi obtido um ganho na acurácia no processo de LCS. As abordagens utilizadas foram DV**+LRT e DV**+NGS.

6.3 Localização de Características usando *Latent Dirichlet Allocation* e Algoritmos Genéticos

Em 2010, Lukins, Kraft e Etzkorn[10] utilizaram LDA para localização de *bugs* em implementações de métodos com o uso abordagem de uma abordagem estática. Um total de 322 *bugs*, oriundos de três projetos (Eclipse, Mozilla e Rhino), foram analisados. Os resultados são claros em afirmar que LDA se mostrou superior em relação ao LSI na tarefa de localização de *bugs*. Características e *bugs* são muito similares em seu processo de localização. É importante ressaltar que em ambas as atividades, a consulta pode ser um conjunto de palavras que descreve um *bug* ou uma característica que deseja-se localizar [6].

Em outra análise, LDA é uma técnica dependente da calibração de um conjunto de hiperparâmetros e sua natureza não determinística promove excessiva oscilação nos tópicos gerados conforme apontado por Agrawal, Fu e Menzies[51]. Assim, buscando superar estas limitações, Panichella et al.[15] propuseram LDA-GA, uma técnica que tenta descobrir a correta calibração dos hiperparâmetros. Os autores aplicaram sua técnica em um conjunto de atividades ligadas à Engenharia de *Software* como recuperação de rastreabilidade, LCS e categorização e nomeação de artefatos de *software*. O estudo utilizou seis *software* de código aberto e descobriu que a utilização de LDA em tarefas de LCS demanda uma cuidadosa calibração, em função das inúmeras configurações possíveis para os valores dos parâmetros. Os autores relatam um ganho de acurácia da nova técnicas em relação a outras técnicas que se prestam à calibração de parâmetros. Apesar do aparente sucesso da técnica LDA, seu custo computacional é alto e a acurácia inferior em LCS, quando comparado com os novos modelos baseados em RNAs [11].

6.4 Melhoria de vetores com enriquecimento semântico

Em Kim et al.[29], os autores buscam melhorar o problema da detecção de intenção em sentenças do discurso falado usando técnicas de PLN. No trabalho ressalta-se que o enriquecimento tem se restringido à granularidade de palavras, em tarefas como busca por similaridade e analogia de termos, sem atingir o nível de sentenças ou parágrafos. Outro ponto é que dicionários desempenham o papel de repositórios de contextos para o enriquecimento. Os autores utilizam então um conjunto de vetores de palavras pré-treinados a partir da utilização do algoritmo *Glove* com uso de dimensionalidade igual a 200. Por fim, uma técnica para análise de sentenças é proposta dentro do contexto do reconhecimento de discurso falado. Em outro trabalho, Bojanowski et al.[30] propõem a utilização de *n*-grams , isto é, fragmentos de palavras enriquecer o espaço embarcado

gerado. Sua aplicação foi testada nas atividades de busca de similaridade e analogia entre termos do texto. A abordagem é baseada no modelo *skip-gram* de Mikolov et al.[31]. Cada fragmento foi representado por um vetor, sendo possível a descoberta de novos termos que estavam implícitos ao se utilizar uma abordagem tradicional. O trabalho citado assemelha-se muito ao modo de trabalho implementado pelo algoritmo *fastText* que foi abordado no Capítulo 5. Por fim, Ye et al.[32] demonstram a viabilidade de se aprimorar a localização de *bugs* a partir da utilização de fontes de informações diversas além do próprio código fonte. Estratégia semelhante será discutida com detalhes no Apêndice D.

Como foi visto muitos trabalhos vem se dedicando à melhoria da qualidade dos vetores de palavras gerados. A partir de um conjunto de informações adicionais procura-se melhorar a qualidade da representação de palavras no espaço vetorial semântico gerado [110, 111, 112].

6.5 O uso de vetores de Palavras e Parágrafos sobre o código fonte

O uso de vetores de palavras e parágrafos tem se consolidado como ferramenta na localização de características e *bugs* no código fonte. Recentemente muitos trabalhos tem se voltado para a utilização de vetores de palavras no processamento de tarefas de PLN, e em especial, no nicho de Engenharia de *Software* [117] para diferentes níveis de granularidade do código fonte [20].

Em um dos Trabalhos de Referência, vide Capítulo 3, Corley, Damevski e Kraft[11] tentam recuperar um conjunto de métodos relevantes do código fonte em resposta a uma consulta baseada em descrições de características de *software*. A consulta é então submetida a um mecanismos de busca baseado no algoritmo *Doc2vec*. Os autores reportam a superioridade da abordagem desenvolvida em relação à técnica LDA. O modelo é mais compacto e mais rápido do que o modelo LDA. Além disso, é ressaltada a importância da correta calibração dos parâmetros do algoritmo *Doc2vec* em tarefas específicas como a LCS.

No Capítulo 4, que retrata um experimento de LBS [5], os autores promovem a reformulação da consulta (descrição do *bug*) para aumentar a acurácia do processo de recuperação da rastreabilidade entre a descrição do *bug* e as classes do código fonte analisado. Assim os relatórios de *bugs* são previamente classificados quanto à natureza do tipo informação contida no mesmo. Posteriormente, termos são adicionados ou removidos do relatório, para reformulação da consulta a ser submetida ao mecanismo de busca criado.

Em um experimento voltado à localização de *bugs*, Ye et al.[32] trabalham de maneira similar ao que foi feito na abordagem FED, tratada no Capítulo 5, Seção 5.1. O trabalho realizado enriquece *word embeddings*, treinados a partir do código fonte do Eclipse. Além disso, utilizam uma série de fontes de informações externas ao código fonte para produção

dos vetores: a API de programação do Eclipse, do JDT, do *Birt* e da máquina virtual Java *Standard Edition*, tutoriais do Java, o guia de desenvolvedor de *Plugins* dentre outros documentos relacionados ao ambiente de programação do Eclipse. Eles testaram 583 relatórios de *bugs* do *Birt*, 1656 da plataforma Eclipse, 632 do JDT e 817 da biblioteca SWT. O modelo skip-gram foi utilizado para geração dos vetores de palavras provenientes da documentação e do código fonte. Os autores concluem que seus experimentos levaram a um incremento de acurácia do processo frente a outras abordagens correlatas.

Alon et al.[118] propõe aprender vetores representativos de *snippets* de código para usá-los na predição de nomes de métodos. Eles utilizam um *dataset* com 14 milhões de implementações de métodos em uma abordagem denominada *cod2vec*.

Figura 19 – Polissemia entre diferentes domínios de problema

Keyword	Most similar in full SO word2vec	Most similar in Java-specific word2vec	Most similar in Google news word2vec
abort	interrupted, terminate, aborting, aborts, exit	interrupted, terminate, shutdown, disconnect, abnormally	abort, aborting, aborts, abort_fetuses, terminating_pregnancy
cookie	sessionid, coockie, aspxauth, session, coookie	certificate, URL, JSESSIONID, password, Cookies	cupcake, Trefoils, cookie_recipe, oatmeal_cookie, oatmeal_raisin_cookies
fork	forking, fork/exec, forks, /execve, vfork	task, forked, forking, upi1985, forks	pancake_turner, forking, ##mm_Marzocchi, wooden_skewer, ricer
smell	antipattern, smells, code-smell, spaghetti, smelling	valid, rife, messy, duplicated, smelly	odor, smelling, aroma, pungent_odor, pungent_smell
virus	malwarebytes, malware, mcafee, antimalware, viruses	Netstat, Havok_, antivirus, BorlandC, rocksaw	avian_flu_virus, viruses, flu_virus, bird_flu_virus, swine_flu_virus

Efstathiou, Chatzilenas e Spinellis[117] treinaram um modelo com o algoritmo *Word2vec* a partir de 15GB de informações oriundas de posts do *Stack Overflow*. O objetivo principal é tratar problemas de polissemia que podem acontecer no campo da Engenharia de *Software*. Isto significa que um termo pode ter diferentes significados quando considerado em domínios distintos. Assim, se aplicamos vetores criados em um nicho de conhecimento em outro, uma baixa acurácia do modelo será esperada. Os autores relatam ainda que modelos de vetores de palavras pré-treinados com em conjuntos de documentos genéricos ou de senso comum possuem limitado valor para aplicação em domínios específicos de problemas. A tabela mostrada na Figura 19 ilustra esta situação. Por exemplo a palavra "smell" em um contexto geral significa aroma, odor. Já no contexto da linguagem Java aparece como duplicação. Por fim, no contexto específico da Engenharia de Software seria uma anti-padrão. No final do trabalho os autores apresentam um modelo com melhor aplicabilidade e representatividade de palavras usadas no contexto da linguagem Java. No Capítulo 4 e no Apêndice D esta necessidade de filtro de informações por domínio do problema é referenciada como uma das formas de melhorar a representação de vetores de parágrafos aplicados à LCS e LBS.

6.6 Resumo do Capítulo

Neste capítulo procurou-se evidenciar os principais trabalhos voltados à melhoria do processo de LCS. Assim, listou-se um conjunto de técnicas que partem desde a simples localização de termos dentro do código fonte até modernas técnicas baseadas em aprendizado de máquina. Observou-se que com o passar dos anos as técnicas de LCS foram sendo substituídas por novas bem como tentou-se evoluir ou melhorar as já existentes. Além disso, traçou-se um perfil das propostas destinadas à melhoria de representatividade de vetores de palavras e parágrafos quando aplicados a tarefas de Engenharia de *Software*. As principais técnicas listadas neste capítulo foram: LSI, LDA e técnicas baseadas em RNAs para produção de vetores de palavras e parágrafos.

Para cada uma das técnicas citadas foi feito um mapeamento no sentido de apontar possíveis evoluções na utilização das mesmas. Assim *PROMESIR*, LDA-GA e o enriquecimento de palavras e parágrafos representam evoluções dos modelos previamente listados. Por fim, a relação entre este trabalho e os avanços na representação de vetores de palavras e parágrafos foi exposta.

Capítulo 7

Conclusão

A atividade de manutenção de *software* é considerada como uma das mais onerosas dentre aquelas desenvolvidas durante o ciclo de vida do sistema. Desta forma, diversas técnicas oriundas das áreas de RI e aprendizado de máquina tem sido instrumentalizadas, dentro do contexto de Engenharia de *Software*, para mitigar os custos associados à compreensão de programas. Contudo, a aplicação destas técnicas de maneira *ad hoc*, isto é, com calibração padrão de hiperparâmetros impede que os desenvolvedores utilizem tais técnicas em larga escala. Além disso, aspectos evolutivos relacionados às técnicas citadas tem sido desprezados ou subutilizados em experimentos para LCS e LBS. Neste sentido, este trabalho descobriu novas combinações de hiperparâmetros e outras técnicas no estado da arte que permitissem a melhoria do processo de LCS e LBS usando o algoritmo *Doc2vec*.

No estudo apresentado no Capítulo 3, descobriu-se empiricamente que, a combinação entre uma taxa de aprendizado cíclica e a calibração da função de custo *Negative Sampling*, quando aplicada à LCS, pode superar experimentos no estado da arte. Foram superadas as técnicas baseadas nos algoritmos LDA e *Doc2vec*, este último quando utilizado de maneira ad hoc.

No contexto de LBS, o Capítulo 4 descreveu a aplicação do algoritmo *Doc2vec* em conjunto com a ferramenta *Blizzard*. Esta combinação permitiu alcançar significativa melhoria da acurácia na LBS quando foi utilizado o algoritmo *Doc2vec*. Apesar de não superar os resultados obtidos pela ferramenta *Blizzard*, ficaram claramente determinados os fatores e hiperparâmetros do algoritmo *Doc2vec* que favoreceram sua melhoria de performance.

Por fim, o Capítulo 5 trouxe a descrição de um conjunto de experimentos que objetivou a melhoria dos processos de LCS e LBS, mas que não produziram os resultados esperados. Considerou-se importante, sob o ponto de vista de continuidade da pesquisa nesta área, relatar e conhecer quais caminhos não obtiveram êxito nas mais variadas opções de metodologias a serem seguidas.

De maneira geral, as principais contribuições deste trabalho foram:

1. Descoberta e aplicação de calibrações para o algoritmo *Doc2vec*, não citadas na literatura científica ou experimentos no estado da arte, quando aplicado à LCS e LBS;

2. Melhoria do desempenho (ganho de acurácia) do algoritmo *Doc2vec* ao ser aplicado à LBS a partir da combinação deste com ferramentas e técnicas utilizadas no estado da arte;

3. Exploração e exposição de um conjunto de abordagens que se mostraram inefetivas na melhoria da acurácia do processo de LCS e LBS.

Esta tese gerou até o momento a publicação do seguinte artigo:
SILVA, A.C., MAIA, M.A. Improving feature location accuracy via paragraph vector tuning, *Information and Software Technology*, Volume 116, 2019, 106177, ISSN 0950-5849, https://doi.org/10.1016/j.infsof.2019.106177.

7.1 Ameaças à Validade

As ameaças à validade foram concentradas nesta seção pois todas as abordagens lidam basicamente com os mesmos desafios. O caráter aleatório da geração de pesos iniciais para as redes neurais é intrínseco a este trabalho. Esta ameaça foi mitigada com 10 execuções e obtenção do valor médio para a métrica MRR no Capítulo 3. O mesmo não pôde ser feito no Capítulo 4 em função do número de *bugs* a serem localizados. Espera-se que com o avanço dos *frameworks* utilizados neste trabalho e utilização de servidores e paralelização possa-se fazer uso mais intensivo de paralelismo e consequente diminuir o tempo de execução de cada consulta.

Em relação à análise dos resultados, a métrica MRR foi escolhida como a principal forma de avaliação. Contudo, outras métricas, como MAP e a análise dos Top@k métodos retornados, são passíveis de serem utilizadas para este fim. A razão da escolha de MRR foi a definição de um ponto inicial para que o desenvolvedor possa começar sua atividade de manutenção.

Em relação à validação externa, algumas questões precisam ser citadas. Apesar de ter sido usado um *dataset* amplamente utilizado, faz-se ainda necessária a investigação das abordagens utilizadas neste trabalho em outros *datasets*, bem como em outras tarefas da área de Engenharia de *Software*.

Sob o ponto de vista de validade interna, todo trabalho experimental está sujeito a *bugs* eventuais nos *scripts* desenvolvidos. Um esforço para mitigar esta ameaça foi o uso de uma rigorosa modelagem de dados relacional que permite garantir consistência nos dados utilizados, a não ser aquelas eventualmente oriundas dos *datasets* originais providos por seus autores.

7.2 Trabalhos futuros

A correta calibração das técnicas, quando aplicada à LCS e LBS, continua sendo uma tarefa desafiadora por sua complexidade e especificidade. Neste trabalho, a partir de um estudo experimental, foram descobertos novos valores para hiperparâmetros que estão associados ao algoritmo *Doc2vec*, uma das principais técnicas atualmente aplicada a RI no contexto de Engenharia de *Software*. Cite-se por exemplo o tamanho da janela WDS que mostrou contribuir com a acurácia dos processos de LCS e LBS quando calibrado com valores sequer sugeridos na literatura, como por exemplo, WDS=40 ou WDS=50. Outro hiperparâmetro relevante é a função de custo NGS que comumente é utilizada com um valor 5 em experimentos com o *Doc2vec* seja em tarefas de PLN [31] ou LCS [11]. No experimento deste trabalho, um valor de NGS 50 se mostrou mais efetivo do que um NGS igual 5.

Entretanto, explorar todo o espaço de busca de combinações de hiperparâmetros não é trivial e mostrou-se computacionalmente caro, limitando este trabalho a um escopo discreto de valores que foram explorados nos Capítulos 3 e 4. Assim, trata-se de uma tarefa de otimização em que deseja-se calibrar uma máquina para que esta apresente melhor desempenho. Panichella et al.[15] trataram este problema propondo um algoritmo genético (AG) capaz de descobrir combinações de hiperparâmetros para a técnica LDA. Estas combinações, ótimas ou subótimas, permitiram a LBS com a melhor eficiência sem que fosse necessário testar todas as combinações possíveis. Desta forma, uma alternativa, à utilização de força bruta para descoberta dos valores de hiperparâmetros ótimos, é o desenvolvimento de um algoritmo genético capaz desta tarefa. Os hiperparâmetros WDS, NGS, DM além do número de treinamentos da RNA poderiam figurar como características dos indivíduos do AG. A função de custo proposta seria exercida pelo valor do MRR calculado para cada conjunto de consultas em estudo.

Em três pontos neste trabalho foi realçada a relevância de se entender melhor como a adição de mais dados, para treinamento do modelo resultante do algoritmo *Doc2vec*, pode ser realizada. Assim, no Capítulo 4, Seção 4.6, foi discutido que simplesmente adicionar mais dados ao modelo sem levar em conta a especificidade dos problema e *software* analisados, não traz melhorias para a qualidade dos vetores resultantes. No Capítulo 5 uma tentativa frustrada de se enriquecer vetores a partir de aplicações Java, utilizadas sem quaisquer critérios de escolha, também não se mostrou efetiva. Por fim, no Apêndice D foi relatada a experiência positiva em se utilizar *software* do mesmo domínio do problema para enriquecimento e ajuste de vetores de parágrafos utilizados em um contexto de LCS. Nesta última tentativa os *software* foram escolhidos de maneira manual levando-se em conta a expertise do autor do trabalho. Um tratamento em maior escala que permita tratar uma quantidade maior de *software* relacionados acaba sendo inviável quando realizada de forma manual. Além disso, *software* em outras linguagens de programação também

poderiam ser utilizados sem que o autor precisasse conhecê-los de antemão. Neste sentido, uma proposta viável a ser implementada está relacionada à utilização de técnicas de categorização de *software* amplamente estudadas na Engenharia de *Software* [119, 120, 121]. Desta forma, uma separação mais racional e em larga escala de *software*, com grande similaridade, pode ser alcançada. Isto permitirá uma avaliação final do uso de aplicações de mesmo domínio do problema para enriquecimento semântico de vetores aplicados à LCS e LBS.

Rastreabilidade de *software* é definida como a habilidade de se poder relacionar artefatos criados durante o desenvolvimento de um *software* sob diferentes perspectivas e níveis de abstração [122]. Estes links devem ser criados e mantidos para prover aos desenvolvedores uma melhor compreensão do sistema. Como exemplo pode-se rastrear requisitos para suas respectivas implementações no código fonte. Quando estes links não são criados ou não estão disponíveis torna-se necessária a recuperação destes relacionamentos. Assim, a tarefa de recuperação de links é aquela em que o *stakeholder* analisa dois conjuntos de artefatos de *software* e geralmente usa uma técnica de RI para recuperar e apresentar ao usuário pares de artefatos similares [123].

Kaushik, Tahvildari e Moore[124] propõem recuperar a rastreabilidade entre *bugs* e casos de teste de um sistema utilizando técnicas de RI. Para Falessi D.[125], ferramentas automatizadas retornam uma longa lista ordenada de artefatos que precisa ser inspecionada manualmente. Assim, quanto mais precisa a técnica de RI empregada, menor serão os esforços dispendidos pelo desenvolvedor na tentativa de recuperar e relacionar artefatos de *software*. Este trabalho lidou com este problema promovendo a recuperação da rastreabilidade entre *bugs*/características e os respectivos trechos de código fonte que os implementam. Desta forma, as melhorias e recomendações providas aqui se habilitam para serem testadas empiricamente e utilizadas em uma gama maior de tarefas de recuperação de rastreabilidade.

O mecanismo de geração de vetores de palavras e parágrafos produzidos pelo *fastText* foi utilizado neste trabalho no Capítulo 5, Seção 5.3. Entretanto, esta abordagem não apresentou melhores resultados do que o *Doc2vec*. Assim, uma alternativa viável é utilizar o Glove [63]. O Glove foi citado como um mecanismo alternativo de geração de vetores de palavras utilizado comumente em tarefas de PLN.

De maneira resumida, propõem-se como trabalhos futuros os seguintes itens:

1. O desenvolvimento de um algoritmo genético capaz de descobrir, em um espaço de busca formado por combinações de hiperparâmetros do algoritmo *Doc2vec*, um conjunto ótimo ou subótimo de calibrações, usando como função *fitness* o valor de MMR calculado para um conjunto de consultas;

2. Utilização de técnicas categorização de *software* para seleção de aplicações a serem utilizadas no enriquecimento semântico de vetores de parágrafos aplicados à LCS e

LBS;

3. Avaliação do motor de busca proposto em um conjunto maior de atividades da área de Engenharia de *Software* como a recuperação da rastreabilidade entre outros artefatos de *software*.

4. Criação, armazenamento e utilização de *vetores de parágrafos* gerados por outros mecanismos de geração de vetores de parágrafos como Glove.

7.3 Considerações finais

Para concluir, pode-se dizer que a combinação de abordagens (DV**+LRT+NGS50) para melhoria da RNA utilizada pelo algoritmo *Doc2vec* foi eficiente para melhorar a acurácia do processo de LCS. Já a performance do algoritmo *Doc2vec* foi melhorada em experimentos de LBS, mas não superou o estado da arte. Os hiperparâmetros WDS, NGS, DM juntamente com as abordagens LRT e BZZ tiveram sua influência mensurada sobre a performance do algoritmo *Doc2vec*.

Referências

1 SOMMERVILLE, I. **Software Engineering**. 8. ed. Harlow, England: Addison-Wesley, 2007.

2 PENNY G. E TAKANG, A. A. **Software Maintenance: Concepts And Practice**. 2. ed. World Scientific Publishing Company, 2003. DOI: https://doi.org/10.1142/5318. ISBN 9789814485616. Disponível em: <https://books.google.com.br/books?id=ZBzJCgAAQBAJ>. Acesso em: 13 abr. 2018.

3 NISHIZONO, K. et al. Source code comprehension strategies and metrics to predict comprehension effort in software maintenance and evolution tasks - an empirical study with industry practitioners. In: **Proceedings of the 2011 27th IEEE International Conference on Software Maintenance**. Washington, DC, USA: IEEE Computer Society, 2011. (ICSM '11), p. 473–481. ISBN 978-1-4577-0663-9. DOI: https://doi.org/10.1109/ICSM.2011.6080814. Disponível em: <https://ieeexplore.ieee.org/document/6080814>. Acesso em: 12 abr. 2018.

4 EADDY, M. et al. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In: **Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension**. Washington, DC, USA: IEEE Computer Society, 2008. (ICPC '08), p. 53–62. ISBN 978-0-7695-3176-2. DOI: https://doi.org/10.1109/ICPC.2008.39. Disponível em: <https://ieeexplore.ieee.org/document/4556117>. Acesso em: 29 maio 2018.

5 RAHMAN, M. M.; ROY, C. K. Improving ir-based bug localization with context-aware query reformulation. In: **Proc. ESEC/FSE**. [S.l.: s.n.], 2018. p. 621–632. DOI: https://doi.org/10.1145/3236024.3236065.

6 DIT, B. et al. Feature location in source code: A taxonomy and survey. **Journal of Software Maintenance and Evolution: Research and Practice**, Wiley Online Library, New Jersey, USA, v. 25, n. 1, p. 53–95, January 2011. DOI: https://doi.org/10.1002/smr.567. Disponível em: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.567>. Acesso em: 27 jun. 2018.

7 MENS, K.; MENS, T.; WERMELINGER, M. Maintaining software through intentional source-code views. In: **Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering**.

New York, NY, USA: ACM, 2002. (SEKE ’02), p. 289–296. ISBN 1-58113-556-4. DOI: https://doi.org/10.1145/568760.568812. Disponível em: <https://dl.acm.org/citation.cfm?id=568812>. Acesso em: 04 abr. 2018.

8 SOBREIRA, V.; MAIA, M. A visual trace analysis tool for understanding feature scattering. In: . [S.l.: s.n.], 2008. p. 337 – 338. DOI: https://doi.org/10.1109/WCRE.2008.40.

9 MARCUS, A. et al. An information retrieval approach to concept location in source code. In: **Proceedings of the 11th Working Conference on Reverse Engineering**. Washington, DC, USA: IEEE Computer Society, 2004. (WCRE ’04), p. 214–223. ISBN 0-7695-2243-2. ISSN 1095-1350. DOI: https://doi.org/10.1109/WCRE.2004.10. Disponível em: <https://ieeexplore.ieee.org/document/1374321>. Acesso em: 29 jun. 2018.

10 LUKINS, S.; KRAFT, N. A.; ETZKORN, L. H. Bug localization using latent dirichlet allocation. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 52, n. 9, p. 972–990, set. 2010. ISSN 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2010.04.002. Disponível em: <http://dx.doi.org/10.1016/j.infsof.2010.04.002>. Acesso em: 17 abr. 2018.

11 CORLEY, C. S.; DAMEVSKI, K.; KRAFT, N. A. Exploring the use of deep learning for feature location. In: **2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. Washington, DC, USA: IEEE Computer Society, 2015. (ICSME), p. 556–560. DOI: https://doi.org/10.1109/ICSM.2015.7332513. Disponível em: <https://ieeexplore.ieee.org/document/7332513>. Acesso em: 02 maio 2018.

12 YE, X. et al. From word embeddings to document similarities for improved information retrieval in software engineering. In: **Proceedings of the 38th International Conference on Software Engineering**. Austin, TX: [s.n.], 2016. DOI: https://doi.org/10.1145/2884781.2884862.

13 MIKOLOV, T. et al. Distributed representations of words and phrases and their compositionality. Curran Associates Inc., USA, p. 3111–3119, 2013. Disponível em: <http://dl.acm.org/citation.cfm?id=2999792.2999959>. Acesso em: 02 jun. 2018.

14 POSHYVANYK, D. et al. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. **IEEE Transactions on Software Engineering**, IEEE Computer Society, Washington, DC, USA, v. 33, n. 6, p. 420–432, June 2007. ISSN 0098-5589. DOI: https://doi.org/10.1109/TSE.2007.1016. Disponível em: <doi.ieeecomputersociety.org/10.1109/TSE.2007.1016>. Acesso em: 01 jul. 2018.

15 PANICHELLA, A. et al. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In: **2013 35th International Conference on Software Engineering (ICSE)**. San Francisco, CA, USA: IEEE Press Piscataway, 2013. (ICSE’13), p. 522–531. ISBN 978-1-4673-3076-3. ISSN 0270-5257. DOI: https://doi.org/10.1109/ICSE.2013.6606598. Disponível em: <https://ieeexplore.ieee.org/document/6606598>. Acesso em: 20 abr. 2018.

16 MAIA, M.; SILVA, A.; SILVA, I. On the influence of latent semantic analysis parameterization for bug localization. **Revista de Informática Teórica e Aplicada**, v. 20, p. 49, 11 2013. DOI: https://doi.org/10.22456/2175-2745.31690.

17 BRINK, H.; RICHARDS, J.; FETHEROLF, M. **Real-World Machine Learning, Brink-Richards-Fetherolf-ronin, 2017: Real-World Machine Learning**. Manning Publications, C., 2017. (Real-World Machine Learning). ISBN 9781617291920. Disponível em: <https://books.google.com.br/books?id=OGlhDwAAQBAJ>.

18 RATH, M.; LO, D.; MäDER, P. Analyzing requirements and traceability information to improve bug localization. In: **Proceedings of the 15th International Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2018. (MSR ’18), p. 442–453. ISBN 978-1-4503-5716-6. DOI: https://doi.org/10.1145/3196398.3196415. Disponível em: <http://doi.acm.org/10.1145/3196398.3196415>. Acesso em: 19 jun. 2018.

19 HINDLE, A. et al. On the naturalness of software. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 59, n. 5, p. 122–131, apr 2016. ISSN 0001-0782. DOI: https://doi.org/10.1145/2902362. Disponível em: <https://doi.org/10.1145/2902362>.

20 CHEN, Z.; MONPERRUS, M. A literature study of embeddings on source code. **CoRR**, abs/1904.03061, 2019. Disponível em: <http://arxiv.org/abs/1904.03061>.

21 IEEE STANDARD FOR SOFTWARE MAINTENANCE. **IEEE 1219**: The process for managing and executing software maintenance activities is described. Piscataway, NJ, USA, 1993. 45 p. DOI: https://doi.org/10.1109/IEEESTD.1993.115570. Disponível em: <https://ieeexplore.ieee.org/document/257623>. Acesso em: 19 abr. 2018.

22 BINKLEY, D.; LAWRIE, D. Information retrieval applications in software maintenance and evolution. In: **Encyclopedia of Software Engineering**. Pennsylvania State University University Park, PA, USA: Taylor & Francis, 2011. DOI: https://doi.org/10.1081/E-ESE-120044704. Disponível em: <http://www.cs.loyola.edu/~lawrie/papers/lawrieIRinME.pdf>. Acesso em: 20 jun. 2018.

23 BENNETT, K. H.; RAJLICH, V. T. Software maintenance and evolution: A roadmap. In: **Proceedings of the Conference on The Future of Software Engineering**. New York, NY, USA: ACM, 2000. (ICSE ’00), p. 73–87. ISBN 1-58113-253-0. DOI: https://doi.org/10.1145/336512.336534. Disponível em: <http://doi.acm.org/10.1145/336512.336534>. Acesso em: 02 jul. 2018.

24 BANKER, R. D.; DAVIS, G. B.; SLAUGHTER, S. Software development practices, software complexity, and software maintenance performance: A field study. **Management Science**, v. 44, n. 4, p. 433–450, 09 1998. ISSN 00251909, 15265501. DOI: https://doi.org/10.1287/mnsc.44.4.433. Disponível em: <http://www.jstor.org/stable/2634607>. Acesso em: 29 jul. 2018.

25 ROBSON, D. et al. Approaches to program comprehension. **Journal of Systems and Software**, v. 14, n. 2, p. 79 – 84, 1991. ISSN 0164-1212. DOI: https://doi.org/10.1016/0164-1212(91)90092-K. Disponível em: <http://www.sciencedirect.com/science/article/pii/016412129190092K>. Acesso em: 21 ago. 2018.

26 O’BRIEN, M. P. **Software comprehension: A review and research direction**. [S.l.], 2003. Disponível em: <https://ulsites.ul.ie/csis/sites/default/files/csis_software_comprehension.pdf>. Acesso em: 14 jun. 2018.

27 RAJLICH, V. Comprehension and evolution of legacy software. In: **Proceedings of the (19th) International Conference on Software Engineering**. Boston, MA, USA: ACM, 1997. (ICSE '97), p. 669–670. DOI: https://doi.org/10.1145/253228.253820. Disponível em: <http://www.cs.wayne.edu/~severe/publications/Rajlich.ICSE.1997.Comprehension.Evolution.Legacy.Software.pdf>. Acesso em: 01 ago. 2018.

28 BJöRKENSTAM, K. N. **What is a corpus and why are corpora important tools?** 2014. Acesso em: 1 dez. 2019. Disponível em: <https://nordiskateckensprak.files.wordpress.com/2014/01/knb_whatisacorpus_cph-2013_outline.pdf>.

29 KIM, J.-K. et al. Intent detection using semantically enriched word embeddings. **2016 IEEE Spoken Language Technology Workshop (SLT)**, IEEE, p. 414–419, Dec. 2016. DOI: https://doi.org/10.1109/SLT.2016.7846297. Disponível em: <https://ieeexplore.ieee.org/document/7846297>. Acesso em: 29 ago. 2018.

30 BOJANOWSKI, P. et al. Enriching word vectors with subword information. **CoRR**, Association for Computational Linguistics, v. 5, p. 135–146, 2017. DOI: https://doi.org/10.1162/tacl_a_00051. Disponível em: <http://arxiv.org/abs/1607.04606>. Acesso em: 10 jul. 2018.

31 MIKOLOV, T. et al. Efficient estimation of word representations in vector space. **CoRR**, abs/1301.3781, 2013. Disponível em: <https://arxiv.org/abs/1301.3781>. Acesso em: 21 jul. 2018.

32 YE, X. et al. From word embeddings to document similarities for improved information retrieval in software engineering. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 38., 2016. **2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)**. Austin, TX, USA: IEEE/ACM, 2016. p. 404–415. ISSN 1558-1225. DOI: https://doi.org/10.1145/2884781.2884862. Disponível em: <https://ieeexplore.ieee.org/document/7886921>. Acesso em: 15 set. 2018.

33 EISENBARTH, T.; KOSCHKE, R.; SIMON, D. Locating features in source code. **IEEE Trans. Softw. Eng.**, IEEE Computer Society, Piscataway, NJ, USA, v. 29, n. 3, p. 210–224, mar. 2003. ISSN 0098-5589. DOI: https://doi.org/10.1109/TSE.2003.1183929. Disponível em: <https://ieeexplore.ieee.org/document/1183929>. Acesso em: 10 maio 2018.

34 TRIPATHY, P.; NAIK, K. **Software Evolution and Maintenance**. Hoboken, Nova Jersey, EUA: John Wiley Sons, 2014. ISBN 9780470603413. Disponível em: <https://books.google.com.br/books?id=0UXxBQAAQBAJ>. Acesso em: 14 set. 2018.

35 PETRENKO, M.; RAJLICH, V. Concept location using program dependencies and information retrieval (depir). **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 55, n. 4, p. 651–659, abr. 2013. ISSN 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2012.09.013. Disponível em: <http://dx.doi.org/10.1016/j.infsof.2012.09.013>. Acesso em: 17 maio 2018.

36 BOHNET, J.; VOIGT, S.; DOELLNER, J. Locating and understanding features of complex software systems by synchronizing time-, collaboration- and code-focused views on execution traces. In: INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, 16., 2008, Amsterdam, The Netherlands. **2008 16th IEEE International Conference on Program Comprehension**.

Washington, D.C., EUA: IEEE Computer Society, 2008. p. 268–271. ISSN 1092-8138. DOI: https://doi.org/10.1109/ICPC.2008.21. Disponível em: <https://ieeexplore.ieee.org/abstract/document/4556142>. Acesso em: 09 jul. 2018.

37 SOBRINHO, E. V. d. P.; LUCIA, A. D.; MAIA, M. d. A systematic literature review on bad smells — 5 w's: which, when, what, who, where. **IEEE Transactions on Software Engineering**, p. 1–1, 2018. ISSN 2326-3881. DOI: https://doi.org/10.1109/TSE.2018.2880977.

38 ZELLER, A. **Why Programs Fail, Second Edition: A Guide to Systematic Debugging**. 2nd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009. ISBN 0123745152.

39 ROBILLARD, M. P.; WEIGAND-WARR, F. Concernmapper: Simple view-based separation of scattered concerns. In: **Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange**. New York, NY, USA: ACM, 2005. (eclipse '05), p. 65–69. ISBN 1-59593-342-5. DOI: https://doi.org/10.1145/1117696.1117710. Disponível em: <http://doi.acm.org/10.1145/1117696.1117710>. Acesso em: 03 ago. 2018.

40 LUCIA, A. D. et al. Recovering traceability links in software artifact management systems using information retrieval methods. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 16, n. 4, set. 2007. ISSN 1049-331X. Disponível em: <http://doi.acm.org/10.1145/1276933.1276934>. Acesso em: 10 abr. 2018.

41 GANEGEDARA, T. **Natural Language Processing with TensorFlow: Teach language to machines using Python's deep learning library**. Birmingham, UK: Packt Publishing, 2018. ISBN 9781788477758. Disponível em: <https://books.google.com.br/books?id=LHxeDwAAQBAJ>. Acesso em: 12 out. 2018.

42 CAMBRONERO, J. et al. When deep learning met code search. In: . [s.n.], 2019. abs/1905.03813. Disponível em: <http://arxiv.org/abs/1905.03813>.

43 LIU, D. et al. Feature location via information retrieval based filtering of a single scenario execution trace. In: **Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: ACM, 2007. (ASE '07), p. 234–243. ISBN 978-1-59593-882-4. DOI: https://doi.org/10.1145/1321631.1321667. Disponível em: <http://doi.acm.org/10.1145/1321631.1321667>. Acesso em: 18 maio 2018.

44 Ali, N. et al. Improving bug location using binary class relationships. In: **2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation**. [S.l.: s.n.], 2012. p. 174–183. DOI: https://doi.org/10.1109/SCAM.2012.26.

45 LI, K. **Combining Static and Dynamic Analysis for Bug Detection and Program Understanding**. Tese (Doutorado) — University of Massachusetts Amherst, 2016. DOI: https://doi.org/10.1007/978-3-540-73770-4_1.

46 DEERWESTER, S. et al. Indexing by latent semantic analysis. **Journal of the American Society for Information Science**, Wiley Online Library, New Jersey, USA, v. 41, n. 6, p. 391–407, January 1990. DOI: https://doi.org/10.1002/(SICI)1097-4571(199009)41:6<391::AID-ASI1>3.0.CO;2-9. Disponível em: <https://doi.org/10.1002/(SICI)1097-4571(199009)41:6<391::AID-ASI1>3.0.CO;2-9>. Acesso em: 12 jul. 2018.

47 BLEI, D. M.; NG, A. Y.; JORDAN, M. I. Latent dirichlet allocation. **J. Mach. Learn. Res.**, JMLR.org, v. 3, p. 993–1022, mar. 2003. ISSN 1532-4435. DOI: https://doi.org/10.1162/jmlr.2003.3.4-5.993. Disponível em: <http://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf>. Acesso em: 19 abr. 2018.

48 JELODAR, H. et al. Latent dirichlet allocation (LDA) and topic modeling: models, applications, a survey. **CoRR**, abs/1711.04305, November 2017. DOI: https://doi.org/10.1007/s11042-018-6894-4. Disponível em: <https://arxiv.org/abs/1711.04305>. Acesso em: 10 ago. 2018.

49 TIAN, K.; REVELLE, M.; POSHYVANYK, D. Using latent dirichlet allocation for automatic categorization of software. In: **Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories**. Washington, DC, USA: IEEE Computer Society, 2009. (MSR ’09), p. 163–166. ISBN 978-1-4244-3493-0. DOI: https://doi.org/10.1109/MSR.2009.5069496. Disponível em: <http://dx.doi.org/10.1109/MSR.2009.5069496>. Acesso em: 20 abr. 2018.

50 ASUNCION, H. U.; ASUNCION, A. U.; TAYLOR, R. N. Software traceability with topic modeling. In: **2010 ACM/IEEE 32nd International Conference on Software Engineering**. Washington, DC, USA: IEEE, 2010. v. 1, p. 95–104. ISSN 1558-1225. DOI: https://doi.org/10.1145/1806799.1806817. Disponível em: <https://ieeexplore.ieee.org/abstract/document/6062077>. Acesso em: 01 maio 2018.

51 AGRAWAL, A.; FU, W.; MENZIES, T. What is wrong with topic modeling? (and how to fix it using search-based se). **Information and Software Technology**, Elsevier, v. 98, p. 74–88, June 2018. ISSN 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.02.005. Disponível em: <https://www.sciencedirect.com/science/article/abs/pii/S0950584917300861>. Acesso em: 24 out. 2018.

52 BARONI, M.; DINU, G.; KRUSZEWSKI, G. Don’t count, predict! a systematic comparison of context-counting vs.context-predicting semantic vectors. In: **Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics**. Stroudsburg PA 18360, USA: Association for Computational Linguistics, 2014. v. 1, p. 238–247. DOI: https://doi.org/10.3115/v1/P14-1023. Disponível em: <http://aclweb.org/anthology/P14-1023>. Acesso em: 17 jun. 2018.

53 MIKOLOV, T.; YIH, W.-t.; ZWEIG, G. Linguistic regularities in continuous space word representations. In: NORTH AMERICAN CHAPTER OF THE ASSOCIATION FOR COMPUTATIONAL LINGUISTICS - HUMAN LANGUAGE TECHNOLOGIES (NAACL HLT), 2013, Atlanta, Georgia, USA. **Proceedings of NAACL-HLT 2013**. Stroudsburg, PA, USA: Association for Computational Linguistics, 2013. p. 746–751. Disponível em: <https://www.aclweb.org/anthology/N13-1090>. Acesso em: 25 ago. 2018.

54 SILVA, I. N.; SPATTI, H. D.; FLAUZINO, R. A. **Redes Neurais Artificiais para engenharia e ciências aplicadas**. São Paulo/SP: Artliber Editora, 2010. ISBN 978-85-88098-53-4.

55 BISHOP, C. M. **Pattern Recognition and Machine Learning (Information Science and Statistics)**. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN 0387310738.

56 SKANSI, S. **Introduction to Deep Learning: From Logical Calculus to Artificial Intelligence**. Springer International Publishing, 2018. (Undergraduate Topics in Computer Science). ISBN 9783319730042. Disponível em: <https://books.google.com.br/books?id=5cNKDwAAQBAJ>.

57 LEE, J. H.; SHIN, J.; REALFF, M. J. Machine learning: Overview of the recent progresses and implications for the process systems engineering field. **Computers Chemical Engineering**, v. 114, p. 111–121, 2018. ISSN 0098-1354. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0098135417303538>. Acesso em: 28 maio 2018.

58 BENGIO, Y. et al. Greedy layer-wise training of deep networks. In: **Proceedings of the 19th International Conference on Neural Information Processing Systems**. Cambridge, MA, USA: MIT Press, 2006. (NIPS'06), p. 153–160. Disponível em: <http://dl.acm.org/citation.cfm?id=2976456.2976476>. Acesso em: 29 maio 2018.

59 GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. Cambridge: The MIT Press, 2016. ISBN 0262035618, 9780262035613.

60 LAMM, E.; UNGER, R. **Biological Computation**. Taylor & Francis, 2011. (Chapman & Hall/CRC Mathematical and Computational Biology). ISBN 9781420087956. Disponível em: <https://books.google.com.br/books?id=OgK42PCvAy4C>.

61 LI, B. et al. Scaling word2vec on big corpus. **Data Science and Engineering**, 06 2019. DOI: https://doi.org/10.1007/s41019-019-0096-6.

62 LE, Q. V.; MIKOLOV, T. Distributed representations of sentences and documents. **CoRR**, abs/1405.4053, 2014. Disponível em: <http://arxiv.org/abs/1405.4053>.

63 PENNINGTON, J.; SOCHER, R.; MANNING, C. Glove: Global vectors for word representation. In: . [S.l.: s.n.], 2014. v. 14, p. 1532–1543. DOI: https://doi.org/10.3115/v1/D14-1162.

64 LAI, S. et al. How to generate a good word embedding? **IEEE Intelligent Systems**, PP, p. 1–1, 06 2017. DOI: https://doi.org/10.1109/MIS.2017.2581325.

65 WANG, B. et al. Evaluating word embedding models: Methods and experimental results. **CoRR**, abs/1901.09785, 2019. DOI: https://doi.org/10.1017/ATSIP.2019.12. Disponível em: <http://arxiv.org/abs/1901.09785>.

66 COLLOBERT, R.; WESTON, J. A unified architecture for natural language processing: Deep neural networks with multitask learning. In: **Proceedings of the 25th International Conference on Machine Learning**. New York, NY, USA: Association for Computing Machinery, 2008. (ICML '08), p. 160–167. ISBN 9781605582054. DOI: https://doi.org/10.1145/1390156.1390177.

67 MNIH, A.; HINTON, G. E. A scalable hierarchical distributed language model. In: KOLLER, D. et al. (Ed.). **Advances in Neural Information Processing Systems 21**. [S.l.]: Curran Associates, Inc., 2009. p. 1081–1088.

68 PENNINGTON, J.; SOCHER, R.; MANNING, C. Glove: Global vectors for word representation. In: **Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)**. Doha, Qatar: Association for

Computational Linguistics, 2014. p. 1532–1543. DOI: https://doi.org/10.3115/v1/D14-1162.

69 ERIC, K. **Understanding Multi-Dimensionality in Vector Space Modeling**. 2019. Acesso em: 01 jan. 2019. Disponível em: <https://aegis4048.github.io/understanding_multi-dimensionality_in_vector_space_modeling>.

70 DENG, L.; LIU, Y. **Deep Learning in Natural Language Processing**. Singapore: Springer Nature Singapore Pte Ltd., 2018. ISBN 9789811052095. Disponível em: <https://books.google.com.br/books?id=y_lcDwAAQBAJ>. Acesso em: 20 jul. 2018.

71 PARK, J. et al. **Advances in Computer Science and Ubiquitous Computing**. Cinngapura: Springer Singapore, 2017. (Lecture Notes in Electrical Engineering). ISBN 9789811076053. Disponível em: <https://books.google.com.br/books?id=A-lDDwAAQBAJ>. Acesso em: 14 jul. 2018.

72 DONG, G.; LIU, H. **Feature Engineering for Machine Learning and Data Analytics**. Boca Raton, FL, USA: CRC Press, 2018. (Chapman & Hall/CRC Data Mining and Knowledge Discovery Series). ISBN 9781351721271. Disponível em: <https://books.google.com.br/books?id=661SDwAAQBAJ>. Acesso em: 02 set. 2018.

73 AGGARWAL, C. **Neural Networks and Deep Learning: A Textbook**. Springer International Publishing, 2018. ISBN 9783319944630. Disponível em: <https://books.google.com.br/books?id=achqDwAAQBAJ>.

74 RONG, X. word2vec parameter learning explained. **CoRR**, abs/1411.2738, 2014. Disponível em: <http://arxiv.org/abs/1411.2738>. Acesso em: 10 ago. 2018.

75 JIANG, S. et al. Integrating rich document representations for text classification. In: **2016 IEEE Systems and Information Engineering Design Symposium (SIEDS)**. [S.l.: s.n.], 2016. p. 303–308. DOI: https://doi.org/10.1109/SIEDS.2016.7489319.

76 PAUKKERI, M.-S. et al. Effect of dimensionality reduction on different distance measures in document clustering. In: LU, B.-L.; ZHANG, L.; KWOK, J. (Ed.). **Neural Information Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 167–176. DOI: https://doi.org/10.1007/978-3-642-24965-5$_{1}$9.

77 ABUALIGAH, L.; HANANDEH, E. Applying genetic algorithms to information retrieval using vector space model. **International Journal of Computer Science, Engineering and Applications**, v. 5, p. 19–28, 02 2015. DOI: https://doi.org/10.5121/ijcsea.2015.5102.

78 YOUNG, T. et al. Recent trends in deep learning based natural language processing. **CoRR**, abs/1708.02709, 2017. DOI: https://doi.org/10.1109/MCI.2018.2840738. Disponível em: <http://arxiv.org/abs/1708.02709>.

79 ANTONIOL, G. et al. Recovering traceability links between code and documentation. **IEEE Transactions on Software Engineering**, v. 28, n. 10, p. 970–983, Oct 2002. ISSN 0098-5589. DOI: https://doi.org/10.1109/TSE.2002.1041053.

80 MANNING, C. D.; RAGHAVAN, P.; SCHüTZE, H. **Introduction to Information Retrieval**. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521865719, 9780521865715.

81 SAHA, R. K. et al. Improving bug localization using structured information retrieval. In: INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), 28., 2013, Palo Alto, USA. **2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. IEEE/ACM, 2013. p. 345–355. DOI: https://doi.org/10.1109/ASE.2013.6693093. Disponível em: <https://ieeexplore.ieee.org/document/6693093>. Acesso em: 19 set. 2018.

82 BATTITI, R. Using mutual information for selecting features in supervised neural net learning. **Trans. Neur. Netw.**, IEEE Press, Piscataway, NJ, USA, v. 5, n. 4, p. 537–550, jul. 1994. ISSN 1045-9227. DOI: https://doi.org/10.1109/72.298224. Disponível em: <https://doi.org/10.1109/72.298224>. Acesso em: 11 ago. 2018.

83 HINTON, G. E.; VINYALS, O.; DEAN, J. Distilling the knowledge in a neural network. **CoRR**, abs/1503.02531, 2015. Disponível em: <https://arxiv.org/abs/1503.02531>. Acesso em: 24 jul. 2018.

84 VILLIERS, J. de; BARNARD, E. Backpropagation neural nets with one and two hidden layers. **IEEE Transactions on Neural Networks**, v. 4, n. 1, p. 136–141, Jan 1993. ISSN 1045-9227. DOI: https://doi.org/10.1109/72.182704. Disponível em: <https://ieeexplore.ieee.org/document/182704>. Acesso em: 12 maio 2018.

85 HAN, S. et al. Learning both weights and connections for efficient neural networks. In: **Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1**. Cambridge, MA, USA: MIT Press, 2015. (NIPS'15), p. 1135–1143. Disponível em: <http://dl.acm.org/citation.cfm?id=2969239.2969366>. Acesso em: 23 jul. 2018.

86 CASELLES-DUPRÉ, H.; LESAINT, F.; ROYO-LETELIER, J. Word2vec applied to recommendation: Hyperparameters matter. In: **Proceedings of the 12th ACM Conference on Recommender Systems**. New York, NY, USA: ACM, 2018. (RecSys '18), p. 352–356. ISBN 978-1-4503-5901-6. DOI: https://doi.org/10.1145/3240323.3240377. Disponível em: <http://doi.acm.org/10.1145/3240323.3240377>. Acesso em: 18 jun. 2018.

87 MILANOVA, I. et al. Automatic text generation in macedonian using recurrent neural networks. In: GIEVSKA, S.; MADJAROV, G. (Ed.). **ICT Innovations 2019. Big Data Processing and Mining**. Cham: Springer International Publishing, 2019. p. 1–12. ISBN 978-3-030-33110-8. DOI: https://doi.org/10.1007/978-3-030-33110-8_1.

88 ŘEHŮŘEK, R.; SOJKA, P. Software framework for topic modelling with large corpora. In: **Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks**. Valletta, Malta: ELRA, 2010. p. 45–50. DOI: https://doi.org/10.13140/2.1.2393.1847. Disponível em: <http://www.lrec-conf.org/proceedings/lrec2010/workshops/W10.pdf>. Acesso em: 20 maio 2018.

89 NAGOUDI, E. M. B.; FERRERO, J.; SCHWAB, D. Lim-lig at semeval-2017 task1: Enhancing the semantic similarity for arabic sentences with vectors weighting. In: . [S.l.: s.n.], 2017. DOI: https://doi.org/10.18653/v1/S17-2017.

90 HAMED, Z.; BRUCE, C. W. Estimating embedding vectors for queries. In: ACM INTERNATIONAL CONFERENCE ON THE THEORY OF INFORMATION

RETRIEVAL, Newark, Delaware, USA. **Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval**. New York, NY, USA: ACM, 2016. (ICTIR '16), p. 123–132. ISBN 978-1-4503-4497-5. DOI: https://doi.org/10.1145/2970398.2970403. Disponível em: <http://doi.acm.org/10.1145/2970398.2970403>. Acesso em: 02 set. 2018.

91 DIT, B. et al. A dataset from change history to support evaluation of software maintenance tasks. In: **Proceedings of the 10th Working Conference on Mining Software Repositories**. Piscataway, NJ, USA: IEEE Press, 2013. (MSR '13), p. 131–134. ISBN 978-1-4673-2936-1. DOI: https://doi.org/10.1109/MSR.2013.6624019. Disponível em: <http://dl.acm.org/citation.cfm?id=2487085.2487114>. Acesso em: 17 jun. 2018.

92 EDDY, B. P.; KRAFT, N. A.; GRAY, J. Impact of structural weighting on a latent dirichlet allocation–based feature location technique. **Journal of Software: Evolution and Process**, p. 1–25, July 2017. DOI: https://doi.org/10.1002/smr.1892. Disponível em: <https://onlinelibrary.wiley.com/doi/full/10.1002/smr.1892>. Acesso em: 22 set. 2018.

93 MEDINI, S. et al. Scan: An approach to label and relate execution trace segments. In: . [S.l.: s.n.], 2012. v. 26, p. 135–144. ISBN 978-1-4673-4536-1. DOI: https://doi.org/10.1109/WCRE.2012.23.

94 DIT, B. et al. Supporting and accelerating reproducible empirical research in software evolution and maintenance using tracelab component library. **Empirical Software Engineering**, v. 20, 12 2014. DOI: https://doi.org/10.1007/s10664-014-9339-3.

95 SMITH, L. Cyclical learning rates for training neural networks. In: **2017 IEEE Winter Conference on Applications of Computer Vision**. Santa Rosa, CA, USA: CoRR, 2017. (WACV'17), p. 464–472. ISBN 978-1-5090-4822-9. DOI: https://doi.org/10.1109/WACV.2017.58. Disponível em: <doi.ieeecomputersociety.org/10.1109/WACV.2017.58>. Acesso em: 01 jul. 2018.

96 BEIGI, H. **Fundamentals of Speaker Recognition**. New York, NY, USA: Springer US, 2011. (SpringerLink : Bücher). ISBN 9780387775920. Disponível em: <https://books.google.com.br/books?id=qIMDvu3gJCQC>. Acesso em: 03 jun. 2018.

97 RIGUTINI, L. et al. Sortnet: Learning to rank by a neural preference function. **IEEE Transactions on Neural Networks**, v. 22, n. 9, p. 1368–1380, Sept 2011. DOI: https://doi.org/10.1109/TNN.2011.2160875. Acesso em: 25 set. 2018.

98 KRUSKAL-WALLIS Test. In: THE Concise Encyclopedia of Statistics. New York, NY: Springer New York, 2008. p. 288–290. ISBN 978-0-387-32833-1.

99 SPEER, R.; CHIN, J. An ensemble method to produce high-quality word embeddings. **CoRR**, abs/1604.01692, 2016. Disponível em: <http://arxiv.org/abs/1604.01692>. Acesso em: 15 jun. 2018.

100 HUTTER, F.; HOOS, H.; LEYTON-BROWN, K. An efficient approach for assessing hyperparameter importance. In: XING, E. P.; JEBARA, T. (Ed.). **Proceedings of the 31st International Conference on Machine Learning**. Bejing, China: PMLR, 2014. (Proceedings of Machine Learning Research, 1), p. 754–762. Disponível em: <http://dl.acm.org/citation.cfm?id=3044805.3044891>. Acesso em: 21 abr. 2018.

101 SILVA, A. C. e; MAIA, M. de A. Improving feature location accuracy via paragraph vector tuning. **Information and Software Technology**, v. 116, 2019. DOI: https://doi.org/10.1016/j.infsof.2019.106177.

102 ZHAI, C. **Statistical Language Models for Information Retrieval**. Morgan & Claypool, 2009. (Synthesis lectures on human language technologies). ISBN 9781598295900. Disponível em: <https://books.google.com.br/books?id= FwtPRUBqUuQC>.

103 ILYAS, I.; SOLIMAN, M. **Probabilistic Ranking Techniques in Relational Databases**. Morgan & Claypool Publishers, 2011. (Synthesis lectures on data management). ISBN 9781608455676. Disponível em: <https://books.google.com.br/ books?id=Kv-3v97MK0IC>.

104 TANTITHAMTHAVORN, C.; IHARA, A.; HATA, H. Impact analysis of granularity levels on feature location technique. In: **Requirements Engineering**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. v. 22, n. 9, p. 135–149. DOI: https://doi.org/10.1007/978-3-662-43610-$3_1$1.

105 ENRíQUEZ, F.; TROYANO, J. A.; LóPEZ-SOLAZ, T. An approach to the use of word embeddings in an opinion classification task. **Expert Systems with Applications**, v. 66, p. 1 – 6, 2016. ISSN 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2016.09.005.

106 LAU, J. H.; BALDWIN, T. An empirical evaluation of doc2vec with practical insights into document embedding generation. **CoRR**, abs/1607.05368, 2016. DOI: https://doi.org/10.18653/v1/W16-1609. Disponível em: <http://arxiv.org/abs/1607. 05368>.

107 BABII, H.; JANES, A.; ROBBES, R. Modeling vocabulary for big code machine learning. **CoRR**, abs/1904.01873, 2019. Disponível em: <http://arxiv.org/abs/1904. 01873>.

108 GABEL, M.; SU, Z. A study of the uniqueness of source code. In: . [S.l.: s.n.], 2010. p. 147–156. DOI: https://doi.org/10.1145/1882291.1882315.

109 DUSSERRE, E.; PADRO, M. Bigger does not mean better! we prefer specificity. In: **IWCS - 2017 - 12th International Conference on Computational Semantics Short papers**. [s.n.], 2017. Disponível em: <https://www.aclweb.org/anthology/ W17-6908>.

110 ROTHE, S.; SCHÜTZE, H. Autoextend: Extending word embeddings to embeddings for synsets and lexemes. **CoRR**, Association for Computational Linguistics, Beijing, China, abs/1507.01127, p. 1793–1803, July 2015. DOI: https://doi.org/10.3115/v1/P15-1173. Disponível em: <http://arxiv.org/abs/1507. 01127>. Acesso em: 23 ago. 2018.

111 ÇELIKYILMAZ, A. et al. Enriching word embeddings using knowledge graph for semantic tagging in conversational dialog systems. In: **AAAI Spring Symposia**. Menlo Park, Califórnia, EUA: AAAI - Association for the Advancement of Artificial Intelligence, 2015. Disponível em: <https://www.aaai.org/ocs/index.php/SSS/SSS15/ paper/download/10333/10034>. Acesso em: 12 set. 2018.

112 SONG, Y.; ROTH, D. Unsupervised sparse vector densification for short text similarity. In: **2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies**. Denver, Colorado, USA: The Association for Computational Linguistics, 2015. p. 1275–1280. ISBN 9781941643495. DOI: https://doi.org/10.3115/v1/N15-1138. Disponível em: <http://repository.ust.hk/ir/Record/1783.1-79883>. Acesso em: 10 set. 2018.

113 ALSUHAIBANI, R. S. et al. Heuristic-based part-of-speech tagging of source code identifiers and comments. In: **2015 IEEE 5th Workshop on Mining Unstructured Data (MUD)**. [S.l.: s.n.], 2015. p. 1–6. DOI: https://doi.org/10.1109/MUD.2015.7327960.

114 GOLDBERG, Y. A primer on neural network models for natural language processing. **J. Artif. Int. Res.**, AI Access Foundation, USA, v. 57, n. 1, p. 345–420, set. 2016. ISSN 1076-9757. Disponível em: <http://dl.acm.org/citation.cfm?id=3176748.3176757>. Acesso em: 21 jun. 2018.

115 ANTONIOL, G.; GUEHENEUC, Y. G. Feature identification: a novel approach and a case study. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 21., 2005. **21st IEEE International Conference on Software Maintenance (ICSM'05)**. Budapest, Hungary: IEEE, 2005. v. 32, n. 9, p. 357–366. ISSN 1063-6773. DOI: https://doi.org/10.1109/ICSM.2005.48. Disponível em: <https://ieeexplore.ieee.org/document/1707664>. Acesso em: 23 jul. 2018.

116 ANTONIOL, G.; GUEHENEUC, Y. Feature identification: An epidemiological metaphor. In: . [s.n.], 2006. v. 32, n. 9, p. 627–641. ISSN 0098-5589. DOI: https://doi.org/10.1109/TSE.2006.88. Disponível em: <https://ieeexplore.ieee.org/document/1707664>. Acesso em: 10 ago. 2018.

117 EFSTATHIOU, V.; CHATZILENAS, C.; SPINELLIS, D. Word embeddings for the software engineering domain. In: **2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)**. [S.l.: s.n.], 2018. p. 38–41. ISSN 2574-3848.

118 ALON, U. et al. code2vec: Learning distributed representations of code. **CoRR**, abs/1803.09473, 2018. Disponível em: <http://arxiv.org/abs/1803.09473>.

119 COLLING, M. et al. Categorizing software applications for maintenance. In: **2011 27th IEEE International Conference on Software Maintenance (ICSM)**. [S.l.: s.n.], 2011. p. 343–352. DOI: https://doi.org/10.1109/ICSM.2011.6080801.

120 NAFI, K. W. et al. A universal cross language software similarity detector for open source software categorization. **Journal of Systems and Software**, v. 162, p. 110491, 2020. ISSN 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.110491.

121 CATAL, C.; TUGUL, S.; AKPINAR, B. Automatic software categorization using ensemble methods and bytecode analysis. **International Journal of Software Engineering and Knowledge Engineering**, v. 27, n. 07, p. 1129–1144, 2017. DOI: https://doi.org/10.1142/S0218194017500425.

122 SPANOUDAKIS, G.; ZISMAN, A. Software traceability: A roadmap. **Handbook of Software Engineering and Knowledge Engineering**, v. 3, 08 2005. DOI: https://doi.org/10.1142/9789812775245$_0$014.

123 MILLS, C. Automating traceability link recovery through classification. In: **ESEC/FSE 2017**. [S.l.: s.n.], 2017. DOI: https://doi.org/10.1145/3106237.3121280.

124 KAUSHIK, N.; TAHVILDARI, L.; MOORE, M. Reconstructing traceability between bugs and test cases: An experimental study. In: **2011 18th Working Conference on Reverse Engineering**. [S.l.: s.n.], 2011. p. 411–414. DOI: https://doi.org/10.1109/WCRE.2011.58.

125 FALESSI D., D. P. M. C. G. Estimating the number of remaining links in traceability recovery. **Empirical Software Engineering**, v. 22, p. 996–1027, 10 2017. DOI: https://doi.org/10.1007/s10664-016-9460-6.

Apêndices

APÊNDICE A

Valores numéricos para MRR com desvio padrão

Este apêndice traz os resultados obtidos para o experimento do Capítulo 3. O experimento em questão envolveu a localização de caraterísticas de software (LCS). Os software testados foram: o ArgoUML[1] que é um diagramador *Unified Modeling Language* (UML); o JEdit[2] que é um editor de texto para programação; o JabRef[3] que é um gerenciador de referências bibliográficas; e o muCommander[4] que é um gerenciador de arquivos multiplataforma. Para cada software um conjunto de 5 tamanhos discretos de vetores foram testados: 100, 200, 300, 400 e 500. Na Tabela 14, cada célula numérica é representada pela combinação do parâmetro K, que representa o tamanho do vetor, com uma abordagem. Todos os valores numéricos para MRR bem como o desvio padrão foram apresentados.

[1] <http://argouml.tigris.org>
[2] <http://www.jedit.org>
[3] <http://jabref.sourceforge.net>
[4] <http://www.mucommander.com>

Tabela 14 – MRR médio para as abordagens DV**+LRT e DV**+NGS

Software	Abordagens/K	100	200	300	400	500
ArgoUML v0.22	DV*	0.0775	0.0570	0.0625	0.0587	0.0601
	LDA*	0.0175	0.0295	0.0271	0.0611	0.0220
	DV**	0.0699(σ=0.0164)	0.0717(σ=0.0124)	0.0667(σ=0.0114)	0.0728(σ=0.0121)	0.0627(σ=0.0135)
	DV**+LRT	0.1083(σ=0.0101)	0.1035(σ=0.0102)	0.1072(σ=0.0120)	0.1065(σ=0.0114)	0.1095(σ=0.0110)
	DV**+NGS10	0.0827(σ=0.0159)	0.0847(σ=0.0129)	0.0805(σ=0.0119)	0.0883(σ=0.0150)	0.0923(σ=0.0158)
	DV**+NGS20	0.1099(σ=0.0092)	0.1124(σ=0.0165)	0.1000(σ=0.0173)	0.1074(σ=0.0175)	0.1014(σ=0.0179)
	DV**+NGS30	0.1116(σ=0.0110)	0.1128(σ=0.0118)	0.1084(σ=0.0114)	0.1031(σ=0.0134)	0.1121(σ=0.0136)
	DV**+NGS40	0.1195(σ=0.0099)	0.1168(σ=0.0134)	0.1161(σ=0.0149)	0.1122(σ=0.0139)	0.1142(σ=0.0135)
	DV**+NGS50	0.1100(σ=0.0145)	0.1171(σ=0.0164)	0.1157(σ=0.0145)	0.1147(σ=0.0153)	0.1188(σ=0.0157)
	DV**+LRT+NGS50	**0.1762**(σ=0.0213)	**0.1804**(σ=0.0158)	**0.1757**(σ=0.0144)	**0.1803**(σ=0.0132)	**0.1799**(σ=0.0126)
ArgoUML v0.24	DV*	0.0827	0.0906	0.0874	0.0691	0.0942
	LDA*	0.0246	0.0152	0.0260	0.0258	0.0380
	DV**	0.0768(σ=0.0137)	0.0733(σ=0.0132)	0.0811(σ=0.0161)	0.0750(σ=0.0144)	0.0803(σ=0.0143)
	DV**+LRT	0.0693(σ=0.0088)	0.0728(σ=0.0093)	0.0790(σ=0.0100)	0.0776(σ=0.0094)	0.0775(σ=0.0090)
	DV**+NGS10	0.0818(σ=0.0151)	0.0782(σ=0.0149)	0.0793(σ=0.0134)	0.0775(σ=0.0131)	0.0823(σ=0.0128)
	DV**+NGS20	0.0676(σ=0.0130)	0.0773(σ=0.0127)	0.0771(σ=0.0139)	0.0778(σ=0.0132)	0.0818(σ=0.0131)
	DV**+NGS30	0.0812(σ=0.0129)	0.0839(σ=0.0128)	0.0928(σ=0.0136)	0.0924(σ=0.0146)	0.0899(σ=0.0139)
	DV**+NGS40	0.0936(σ=0.0189)	0.0859(σ=0.0160)	0.0963(σ=0.0167)	0.0882(σ=0.0170)	0.0906(σ=0.0180)
	DV**+NGS50	0.0990(σ=0.0164)	0.0965(σ=0.0134)	0.1041(σ=0.0141)	0.0896(σ=0.0160)	0.1017(σ=0.0148)
	DV**+LRT+NGS50	**0.1337**(σ=0.0079)	**0.1283**(σ=0.0088)	**0.1293**(σ=0.0082)	**0.1307**(σ=0.008)	**0.1305**(σ=0.0081)
ArgoUML v0.26.2	DV*	0.0847	0.0890	0.0813	0.0834	0.0805
	LDA*	0.0493	0.0628	0.0857	0.0703	0.0811
	DV**	0.0726(σ=0.0025)	0.0751(σ=0.0060)	0.0755(σ=0.0054)	0.0721(σ=0.0062)	0.0717(σ=0.0060)
	DV**+LRT	0.0817(σ=0.0044)	0.0846(σ=0.0052)	0.0859(σ=0.0062)	0.0830(σ=0.0065)	0.0832(σ=0.0064)
	DV**+NGS10	0.0863(σ=0.0087)	0.0849(σ=0.0077)	0.0834(σ=0.0074)	0.0911(σ=0.0077)	0.0851(σ=0.0073)
	DV**+NGS20	0.0877(σ=0.0058)	0.0935(σ=0.0083)	0.0941(σ=0.0088)	0.0906(σ=0.0083)	0.0959(σ=0.0083)
	DV**+NGS30	0.0983(σ=0.0048)	0.0954(σ=0.0089)	0.0956(σ=0.0101)	0.0999(σ=0.0098)	0.0982(σ=0.0094)
	DV**+NGS40	0.0988(σ=0.0035)	0.0986(σ=0.0054)	0.1021(σ=0.0076)	0.0986(σ=0.0074)	0.1016(σ=0.0072)
	DV**+NGS50	0.1019(σ=0.0070)	0.0967(σ=0.0078)	0.1074(σ=0.0101)	0.1028(σ=0.0102)	0.0968(σ=0.0101)
	DV**+LRT+NGS50	**0.1332**(σ=0.0047)	**0.1337**(σ=0.0065)	**0.1352**(σ=0.0067)	**0.1339**(σ=0.0062)	**0.1389**(σ=0.0073)
JabRef v2.6	DV*	0.0450	0.0373	0.0455	0.0382	0.0428
	LDA*	0.0055	0.0364	**0.1304**	**0.0781**	0.0548
	DV**	0.0144(σ=0.0029)	0.0191(σ=0.0054)	0.0217(σ=0.0091)	0.0197(σ=0.0091)	0.0181(σ=0.0083)
	DV**+LRT	0.0231(σ=0.0077)	0.0197(σ=0.0059)	0.0191(σ=0.0053)	0.0194(σ=0.0049)	0.0208(σ=0.0045)
	DV**+NGS10	0.0252(σ=0.0052)	0.0243(σ=0.0087)	0.0234(σ=0.0098)	0.0256(σ=0.0110)	0.0251(σ=0.0114)
	DV**+NGS20	0.0280(σ=0.0102)	0.0337(σ=0.0098)	0.0298(σ=0.0109)	0.0295(σ=0.0113)	0.0363(σ=0.0121)
	DV**+NGS30	0.0452(σ=0.0118)	0.0392(σ=0.0118)	0.0388(σ=0.0126)	0.0366(σ=0.0117)	0.0389(σ=0.0111)
	DV**+NGS40	0.0394(σ=0.0155)	0.0392(σ=0.0143)	0.0392(σ=0.0155)	0.0387(σ=0.0139)	0.0344(σ=0.0152)
	DV**+NGS50	0.0431(σ=0.0185)	0.0357(σ=0.0153)	0.0370(σ=0.0140)	0.0364(σ=0.0133)	0.0345(σ=0.0122)
	DV**+LRT+NGS50	**0.0765**(σ=0.0168)	**0.0866**(σ=0.0178)	0.0731(σ=0.0163)	0.0734(σ=0.0162)	**0.0742**(σ=0.0163)
jEdit v4.3	DV*	0.0872	0.0791	0.0825	0.0814	0.0679
	LDA*	0.0670	0.0432	0.0641	0.0693	0.0607
	DV**	0.0654(σ=0.0085)	0.0579(σ=0.0101)	0.0608(σ=0.0090)	0.0670(σ=0.0094)	0.0555(σ=0.0097)
	DV**+LRT	0.1103(σ=0.0113)	0.1102(σ=0.0092)	0.1043(σ=0.0089)	0.1058(σ=0.0100)	0.1071(σ=0.0093)
	DV**+NGS10	0.0745(σ=0.0064)	0.0805(σ=0.0120)	0.0765(σ=0.0121)	0.0724(σ=0.0117)	0.0697(σ=0.0119)
	DV**+NGS20	0.0877(σ=0.0104)	0.0860(σ=0.0106)	0.0797(σ=0.0116)	0.0762(σ=0.0114)	0.0743(σ=0.0113)
	DV**+NGS30	0.0886(σ=0.0144)	0.0902(σ=0.0140)	0.0870(σ=0.0139)	0.0844(σ=0.0127)	0.0820(σ=0.0128)
	DV**+NGS40	0.1012(σ=0.0161)	0.0903(σ=0.0166)	0.0936(σ=0.0161)	0.0865(σ=0.0155)	0.0891(σ=0.0144)
	DV**+NGS50	0.0979(σ=0.0121)	0.0984(σ=0.0112)	0.0969(σ=0.0103)	0.0955(σ=0.0118)	0.0991(σ=0.0114)
	DV**+LRT+NGS50	**0.1805**(σ=0.0088)	**0.172**(σ=0.0096)	**0.1731**(σ=0.0092)	**0.1715**(σ=0.0091)	**0.1752**(σ=0.0091)
muCommander v0.8.5	DV*	0.0652	0.0623	0.0703	0.0606	0.0538
	LDA*	0.0392	0.0217	0.0198	0.0559	0.0329
	DV**	0.0906(σ=0.0092)	0.0896(σ=0.0082)	0.0881(σ=0.0100)	0.0883(σ=0.0090)	0.0868(σ=0.0086)
	DV**+LRT	0.0943(σ=0.0062)	0.0890(σ=0.0076)	0.0921(σ=0.0074)	0.0912(σ=0.0072)	0.0927(σ=0.0084)
	DV**+NGS10	0.0910(σ=0.0106)	0.0946(σ=0.0111)	0.0944(σ=0.0127)	0.0932(σ=0.0136)	0.0939(σ=0.0129)
	DV**+NGS20	0.1174(σ=0.0115)	0.1092(σ=0.0122)	0.1080(σ=0.0128)	0.1137(σ=0.0138)	0.1018(σ=0.0137)
	DV**+NGS30	0.1111(σ=0.0161)	0.1069(σ=0.0122)	0.1114(σ=0.0136)	0.1083(σ=0.0122)	0.1015(σ=0.0126)
	DV**+NGS40	0.1121(σ=0.0130)	0.1120(σ=0.0097)	0.1118(σ=0.0104)	0.1183(σ=0.0121)	0.1091(σ=0.0127)
	DV**+NGS50	0.1190(σ=0.0121)	0.1206(σ=0.0160)	0.1186(σ=0.0151)	0.1170(σ=0.0151)	0.1185(σ=0.0159)
	DV**+LRT+NGS50	**0.1605**(σ=0.0118)	**0.1732**(σ=0.0142)	**0.1629**(σ=0.0122)	**0.1727**(σ=0.0134)	**0.1708**(σ=0.0127)

APÊNDICE B

Pré-testes para definição do tamanho do vetor de Parágrafos

Na tentativa de evitar um número excessivo de testes para se averiguar como os hiper-parâmetros influem sobre a acurácia do processo de LBS, realizado no Capítulo 4, foram feitos testes preliminares discretos com o objetivo de definir um tamanho de vetor para os documentos (classes do código fonte) que pudesse ser mais adequado ao teste de massa de hiper-parâmetros, ao invés de fazer um palpite sem nenhum critério. Utilizou-se então as configurações e hiper-parâmetros com melhor performance obtidas a partir do experimento de LCS, tratado no Capítulo 3.

Tabela 15 – Configurações discretas de Hiper-parâmetros para pré-testes

*A*bordagem	NGS	WDS
DV**+NGS5+WDS5	5	5
DV**+LRT+NGS50+WDS5	50	25
DV**+LRT+NGS50+WDS25	50	25
DV**+LRT+NGS50+WDS50	50	50
DV**+LRT+NGS50+WDS50+BZZ	50	50
DV**+LRT+NGS0+WDS50+BZZ+HSM	0	50

Os resultados obtidos com os pré-testes foram condensados na 16. O leitor notará que novos parâmetros como HSM e BZZ foram adicionados. Estes novos parâmetros são discutidos no Capítulo 4. Destaca-se duas linhas em cinza na tabela: primeiro, aquela em que aparece o MRR obtido no Experimento de Referência; e em seguida a melhor performance para um tamanho de vetor igual a 100. Deste fato optou-se por se utilizar este valor nos testes de massa. Dentre todos os valores (100, 200, 300, 400 e 500) é o valor que apresenta melhor performance temporal na execução de testes além de ter performance em termos de MRR equivalente aos outros tamanhos de vetor.

Tabela 16 – MRR para as abordagens DV**, BZZ, LRT, NGS50, WDS50, HSM

Software	Abordagem./K	100	200	300	400	500
ECF	BZZ(0.5426)	-	-	-	-	-
	DV**+NGS5+WDS5	0.117711	0.117688	0.123772	0.119148	0.115668
	DV**+LRT+NGS50+WDS5	0.144988	0.166212	0.161082	0.162855	0.169514
	DV**+LRT+NGS50+WDS25	0.351496	0.336642	0.327721	0.337151	0.33775
	DV**+LRT+NGS50+WDS50	0.302158	0.306281	0.312849	0.318592	0.322375
	DV**+LRT+NGS50+WDS50+BZZ	**0.430145**	0.424504	0.42311	0.425344	0.421191
	DV**+LRT+NGS0+WDS50+BZZ+HSM	0.424378	0.414705	0.423264	0.41634	0.418958
ECLIPSE.JDT.CORE	BZZ(0.5537)	-	-	-	-	-
	DV**+NGS5+WDS5	0.0482505	0.0285912	0.0296657	0.0251637	0.0251581
	DV**+LRT+NGS50+WDS5	0.166188	0.166672	0.174122	0.15861	0.180425
	DV**+LRT+NGS50+WDS25	0.32705	0.315216	0.320836	0.310001	0.328736
	DV**+LRT+NGS50+WDS50	0.30515	0.324612	0.330725	0.329066	0.326925
	DV**+LRT+NGS50+WDS50+BZZ	**0.453717**	0.466397	0.45718	0.464471	0.461525
	DV**+LRT+NGS0+WDS50+BZZ+HSM	0.451401	0.472132	0.478766	0.476205	0.46602
ECLIPSE.JDT.DEBUG	BZZ(0.4616)	-	-	-	-	-
	DV**+NGS5+WDS5	0.105458	0.108511	0.110967	0.100204	0.109168
	DV**+LRT+NGS50+WDS5	0.163065	0.166277	0.173374	0.154726	0.16812
	DV**+LRT+NGS50+WDS25	0.255103	0.258166	0.260675	0.255791	0.256758
	DV**+LRT+NGS50+WDS50	0.270586	0.275884	0.269817	0.271504	0.281793
	DV**+LRT+NGS50+WDS50+BZZ	0.372926	0.378936	0.375366	0.379213	0.373309
	DV**+LRT+NGS0+WDS50+BZZ+HSM	**0.375354**	0.369278	0.364882	0.370429	0.376051
ECLIPSE.JDT.UI	BZZ(0.4801)	-	-	-	-	-
	DV**+NGS5+WDS5	0.0809112	0.0876864	0.0797209	0.0795583	0.0752116
	DV**+LRT+NGS50+WDS5	0.191208	0.200338	0.219331	0.218866	0.222321
	DV**+LRT+NGS50+WDS25	0.277009	0.311259	0.314122	0.307827	0.318067
	DV**+LRT+NGS50+WDS50	0.269934	0.311295	0.311666	0.311699	0.317437
	DV**+LRT+NGS50+WDS50+BZZ	**0.390686**	0.388422	0.386987	0.392856	0.383676
	DV**+LRT+NGS0+WDS50+BZZ+HSM	0.382939	0.363677	0.377853	0.381579	0.38074
ECLIPSE.PDE.UI	BZZ(0.4143)	-	-	-	-	-
	DV**+NGS5+WDS5	0.0647386	0.054178	0.0586524	0.0520612	0.0525863
	DV**+LRT+NGS50+WDS5	0.138277	0.161859	0.158421	0.174716	0.16044
	DV**+LRT+NGS50+WDS25	0.237527	0.247945	0.245616	0.246918	0.232351
	DV**+LRT+NGS50+WDS50	0.23307	0.255177	0.257141	0.256154	0.246852
	DV**+LRT+NGS50+WDS50+BZZ	**0.353958**	0.353991	0.355521	0.354002	0.356002
	DV**+LRT+NGS0+WDS50+BZZ+HSM	0.324047	0.311029	0.321095	0.319144	0.320783
TOMCAT 7.0	BZZ(0.5999)	-	-	-	-	-
	DV**	0.0957512	0.0773128	0.0774405	0.0861522	0.0822015
	DV**+LRT+NGS50+WDS5	0.153177	0.153065	0.164844	0.157304	0.155111
	DV**+LRT+NGS50+WDS25	0.314533	0.344652	0.345395	0.351833	0.346196
	DV**+LRT+NGS50+WDS50	0.337543	0.34927	0.344263	0.351595	0.349388
	DV**+LRT+NGS50+WDS50+BZZ	0.45154	0.459429	0.462264	0.455303	0.45405
	DV**+LRT+NGS0+WDS50+BZZ+HSM	**0.47852**	0.48621	0.489012	0.485767	0.486317

APÊNDICE C

Um exemplo utilizando o algoritmo Doc2vec do Framework Gensim

Este apêndice apresenta um exemplo básico de todos os comandos Python utilizados para realização da localização de características de software. As saídas de cada comando e fase do processo são explicadas.

Linguagem de Programação: Python 3.5
Framework: Gensim 3.4
Algoritmo: Doc2vec
Software: ArgoUML-v0.22[1]
WDS:5
Função de custo da RNA: Negative Sampling (NGS=5)
Modo de Treinamento: PV-DM

Tarefa: Localizar métodos no código fonte em resposta a uma descrição de Característica

---

Passo 1: Selecionar métodos: realizar uma consulta sql no banco e selecionar métodos.

Passo 2: Efetuar o parsing: remover dígitos, palavras com tamanho menor que 3 e caracteres especiais; separar palavras no padrão CamelCase.

Passo 3: Criar uma lista no formato ([vetor de termos do documento], [doc_id]). No exemplo trata-se do método "org.argouml.uml.ui.TabStyle.addTargetListener(TargetListener)" do ArgoUML-v0.22. Neste trabalho, todos os métodos foram armazenados em um banco de dados Postgres[2]. Assim, o valor 5618 é o código deste método dentro do banco de

[1] http://argouml-downloads.tigris.org/argouml-0.22/
[2] https://www.postgresql.org/

dados.

Exemplo de item da lista:

```
TaggedDocument(['adds', 'listener', 'param', ' listener ', ' listener ', 'target', ' listener ', 'target', ' listener ', ' listener ', ' listener ', ' list ', 'target', ' listener ', ' listener '], [ 'mtCodigo −5618'])
```

Comandos em Python para criação da lista:

```
it = LabeledLineSentence(metodos, labelMetodos)
```

Passo 4: Preparar a criação do Modelo chamando seu construtor para passagem de valores de hiper-parâmetros.

Comandos em Python para criação inicial do modelo:

```
import gensim
dvSumModel = gensim.models.Doc2Vec(size=self.numTopicos, window=5, epochs=5, negative=5,
    workers=4)
```

onde:

dvSumModel - nome do modelo

size - tamanho do vetor de termos e documentos;

window - tamanho da janela ou contexto de palavras

epochs - número de épocas de treinamento

negative - valor atribuído ao número de termos fora do contexto da palavra alvo que terão seus pesos atualizados.

workers - número de threads utilizadas para treinar o modelo

Passo 5: Criação de um vocabulário do modelo. Também chamado de dicionário no Gensim, este item mantém um registro único de cada palavra/termo presente no corpus. Posteriormente um vocabulário pode ser consultado para se saber a frequência dos termos bem como descartar aqueles que são muito raros ou muito frequentes. Comando do Python para criação do vocabulário:

```
dvSumModel.build\_vocab(it)
```

Exemplo de vocabulário formado a partir do corpus do ArgoUml v0.22:

```
['prop', 'panel', 'subactivity', 'lookup', 'icon', 'config', 'loader', 'props', 'orientation',  ...
    'trig', 'outs', 'umlobject']
```

Passo 6: Treinamento do modelo: Nesta fase o modelo é treinado. A RNA, implementada no algoritmo Doc2vec, terá seus pesos atualizados para posteriormente formar os vetores representativos de termos e documentos.

Comando do Python para treinamento da RNA:

```
dvSumModel.train(it, total_examples=dvSumModel.corpus_count)
```

Após o treinamento do modelo o vetor de parágrafo pode ser obtido diretamente a

partir do modelo através do seu identificador. O identificador é uma *tag* associada a cada documento e pode ser consultada a partir da variável it. No passo 3, "mtCodigo-5618" é um identificador de parágrafo (método do código fonte) Por fim, basta calcular o cosseno entre cada vetor representativo dos métodos e o vetor da descrição da característica desejada. O vetor de parágrafo gerado para o método "mtCodigo-5618" é uma estrutura de dados de 100 dimensões e foi mostrado na Listagem C.1.

Listagem C.1 – Exemplo de vetor de parágrafo gerado pelo Doc2vec

```
1
2  [−8.25152919e−03 1.63782667e−02 9.38116666e−03 2.09129266e−02
3   −1.11152988e−03 −3.51653527e−03 1.86320283e−02 −1.29962042e−02
4   −1.82770342e−02 −8.81156325e−03 3.15681286e−03 −1.14169568e−02
5   −1.00744991e−02 8.82592611e−03 3.57388606e−04 −2.74178619e−03
6   −3.12027391e−02 −1.65877175e−02 1.17450086e−02 −1.24238841e−02
7    7.98524078e−03 2.75780968e−02 −2.36650556e−02 1.92767270e−02
8   −7.53427390e−03 −2.37081423e−02 1.66543778e−02 1.60062462e−02
9    3.32998820e−02 3.32623683e−02 −2.00133901e−02 −4.37893532e−03
10  −8.39441922e−03 6.76897541e−03 2.06623040e−03 5.70386788e−03
11  −2.11115973e−03 −3.14131961e−03 −2.18405109e−02 6.42285869e−03
12  −6.45616744e−03 2.35729143e−02 −1.81457605e−02 −1.48586398e−02
13   1.61345527e−02 1.60866641e−02 2.43565533e−03 −5.26155299e−03
14  −1.94518715e−02 4.57816124e−02 5.01669897e−03 1.85159259e−02
15  −6.43053651e−03 4.77764942e−03 −3.01912259e−02 −1.99533068e−02
16   8.69760383e−03 −2.17432063e−02 −2.61693709e−02 3.94639745e−03
17   5.25881303e−03 −1.50517945e−03 −2.59147380e−02 −2.58173719e−02
18   9.53245349e−03 −1.24429520e−02 −1.82362627e−02 8.85153934e−03
19  −4.65552602e−03 6.18099142e−03 −1.16333105e−02 −1.63708664e−02
20  −2.05295347e−02 −1.00197690e−03 −9.56567237e−05 −2.38231774e−02
21  −1.80935450e−02 1.94754044e−03 −4.28705895e−03 2.53210077e−03
22   2.47576628e−02 −1.22716725e−02 −1.74854752e−02 −8.81842617e−03
23   3.31373257e−03 −5.43481484e−03 1.72456484e−02 −8.49785376e−03
24   1.45683615e−02 1.58421732e−02 −1.00011360e−02 2.25394126e−02
25  −1.65527035e−02 −1.49811236e−02 7.07798917e−03 −1.04037439e−03
26  −5.12913044e−04 4.66404343e−03 −9.55620408e−03 −1.70612168e−02]
```

Uma consulta, com descrição de característica submetida ao motor de busca, para o ArgoUML-v0.22 é ilustrada na Listagem C.2. Seu vetor equivalente, aqui denominado p1, é mostrado na Listagem C.3 é calculado a partir da somatória dos seus termos constituintes.

Listagem C.2 – Exemplo de consulta

```
1 extra searchpath added time project saved testcase start argouml create name apos apos create attr
      apos attr apos doubleclicking attr compartment create oper doubleclicking attr compartment save
      project apos test zargo apos copy test zargo project test zargo save theoretically zargo files
      identically changes following analysis reveals differences files contained zip file test zargo
      differences fixed affect files iteratively save load cycle test simple ideally analysis project
      diagram types test argo line searchpath href quot project dir quot added test todo identical
      test xmi identical test diagram pgml strange empty fig added name quot fig quot description
      quot org tigris gef presentation fig quot fill quot quot fillcolor quot quot stroke quot quot
      strokecolor quot quot test usecase diagram pgml identical course empty
```

Listagem C.3 – Vetor p1 da consulta

```
1  [ 5.37734659e+01 2.57176933e+01 −6.41775772e+00 −3.79192569e+00
2   −2.20368443e+01 −1.79766619e+01 1.96347009e+01 7.30300915e+00
3   −7.31869076e−01 −2.01846394e+01 −6.89085530e+00 2.26591607e+01
4    4.81705142e−02 −5.05299575e+01 −3.96106148e+01 3.64930783e+01
5   −2.15713049e+00 −1.90894586e+01 −3.63634447e+01 1.32164377e+01
6    1.63655502e+00 7.15038301e+00 1.11202638e+00 3.79308770e+01
7    1.15979282e+01 −2.73832280e+01 −8.07401224e+00 −3.37690757e+01
8    1.65605145e+01 −2.53766093e+01 1.11029402e+00 −1.51644576e+00
9    4.43243907e+00 −5.10686638e−01 5.27429135e+00 1.05977216e+01
10  −1.88790881e+01 −1.24569332e+01 1.81537233e+01 2.04692718e+01
11  −1.17601324e+01 −2.26877198e+01 1.84537106e+01 −4.16995703e+00
12  −2.02995247e+01 −2.48501488e+00 2.38314143e+01 −1.67986251e+01
13  −7.77513166e+00 −5.56595349e+00 3.53128942e+01 1.14169096e+00
14   2.31783711e+01 2.11129068e+01 2.86120923e+01 4.70789627e+01
15   4.24493529e+01 2.22425947e+01 2.60564959e+01 4.83738571e+00
16  −3.56271180e+01 −2.52631332e+01 2.08204015e+01 −1.51756735e+01
17  −3.08651277e+00 −6.17635742e+00 −1.35490164e+01 1.33267830e+01
18   4.82729627e+01 9.46457292e+00 2.28775295e+01 −4.34508875e+01
19  −8.73263615e+00 3.77129310e+01 3.00539474e+01 2.49604228e+01
20  −2.59795278e+01 −2.32642474e+01 −7.01420910e+00 −3.93197041e+00
21  −1.46086774e+01 −2.09714190e+01 1.13923078e+01 1.94062799e+01
22  −4.76263367e+00 −3.08372302e+00 4.12455122e+01 −5.16716368e+01
23  −1.37920716e+01 1.25138284e+01 −3.57853639e+01 −4.16403710e+01
24  −1.32043526e+01 6.43616192e+01 3.70025555e+00 1.73735886e+01
25   2.12912118e+01 −4.78082145e+01 −1.56905074e+01 4.91451407e+01]
```

As similaridades (valores de cosseno) entre a consulta e cada método do código fonte é calculado a partir do seguinte comando em Python:

```
1 sims = dvSumModel.docvecs.most_similar([p1], topn=tamanhoCorpusPrincipal)
```

Algumas similaridades calculadas entre a consulta e cada método do código fonte são mostradas na Listagem C.4. Por questões de espaço não colocamos aqui todas as

similaridades dos 12353 métodos do ArgoUML-v0.22.

Listagem C.4 – Similaridade entre p1 e cada método do ArgoUML

```
1
2 [( 'mtCodigo−1381', 0.8714036345481873), ('mtCodigo−8730', 0.8682155609130859), ('mtCodigo
      −10626', 0.8567337989807129), ('mtCodigo−4838', 0.8510525822639465), ...]
```

APÊNDICE D

Enriquecimento semântico com software do mesmo domínio de problema

Neste apêndice são apresentados os materiais e a metodologia para a abordagem que tem como objetivo o enriquecimento de vetores de parágrafos. Partiu-se então do código fonte de *software* do mesmo domínio do problema para geração de contextos adicionais. Deseja-se extrair padrões a serem empregados no ajuste fino de vetores representativos de palavras e parágrafos, neste último caso, métodos do código fonte.

A Figura 20 ilustra o processo de LCS estudado aqui. Pode-se assim enumerar o conjunto de passos necessários para alcançar o objetivo proposto:

1. Criação do *Corpus* - Separação do código fonte do *software* orientado a objetos na granularidade de métodos;

2. Pré-processamento do código fonte - Atividade que visa remover elementos com baixo valor semântico;

3. Extração de palavras e contextos do *corpus*;

4. Geração de *vetores de palavras* e *vetores de parágrafos*;

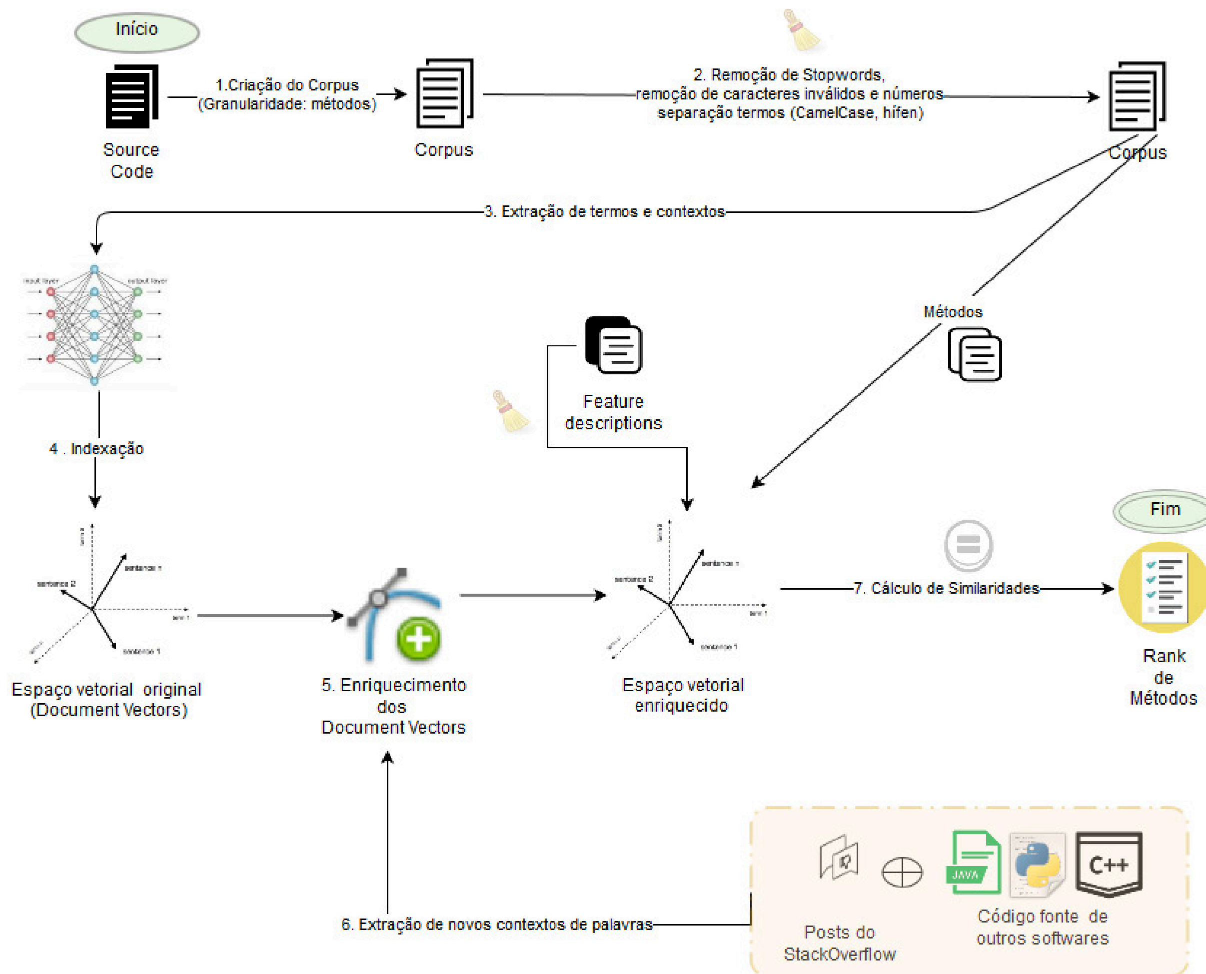
5. Extração de novos contextos a partir de *software* do mesmo domínio;

6. Enriquecimento dos vetores gerados com continuidade do treinamento do modelo gerado pelo algoritmo Doc2vec;

7. Cálculo de similaridades entre uma descrição de característica e todos os métodos do código fonte tendo como saída um *rank* de métodos.

Durante os testes foi utilizado somente o código fonte proveniente de outros *software* do mesmo domínio. Inicialmente a eficácia desta abordagem englobou três versões de

Figura 20 – Enriquecimento de *vetores de parágrafos* para LCS



ArgoUML que foram trabalhadas no Capítulo 3. Assim foram usadas as versões 0.22, 0.24 e 0.26.2 para avaliação da abordagem aqui proposta. A Tabela 17 resume o conjunto de sistemas utilizados para enriquecimento. Um total de 129459 implementações de métodos foram usadas para enriquecer os vetores gerados em cada uma das versões do ArgoUML. Isto corresponde aproximadamente a um total de 9GB de informações externas. Assim um treinamento inicial é realizado com cada *software* alvo (ArgoUML 0.22, 0.24 ou 0.26) conforme ilustrado no Passo 4 da Figura 20. Posteriormente, o modelo obtido continua seu treinamento com novos contextos a partir dos *software* listados na Tabela 17 . Foram escolhidos 6 *software* de código aberto com ligação a temática UML: o editor UML Violet [1]; JModeller[2] e seu *framework* de base JHotDraw[3]; o editor UML Modelio[4], o editor UML plantuml[5] e o *plug-in* do Eclipse Papyrus [6].

[1] https://github.com/violetumleditor/violetumleditor
[2] http://www.jhotdraw.org/downloads/JModeller54b1.zip
[3] http://www.jhotdraw.org/downloads/
[4] https://sourceforge.net/projects/modeliouml/files/3.0.1/
[5] https://sourceforge.net/projects/plantuml/
[6] https://www.eclipse.org/papyrus/downloads/index.php

Tabela 17 – *Software* utilizado na abordagem

Software	Version
Violet UML Editor	2.5.3
JModeller	5.4b1
JHotDraw	7.0.9
Modelio UML	3.0.1
plantuml	1.2018.5
Papyrus Eclipse plugin	3.3.0-RC1

Tabela 18 – Abordagem de enriquecimento semântico: Média do MRR para 10 execuções

Software	Abordagens/K	100	200	300	400	500
ArgoUML v0.22	DV*	0.0775	0.0570	0.0625	0.0587	0.0601
	DV**	0.0699(σ=0.0164)	0.0717(σ=0.0124)	0.0667(σ=0.0114)	0.0728(σ=0.0121)	0.0627(σ=0.0135)
	DV**+FED	**0.1260**(σ=0.0165)	**0.1102**(σ=0.0203)	**0.1048**(σ=0.0188)	**0.1117**(σ=0.0172)	**0.0993**(σ=0.0169)
ArgoUML v0.24	DV*	0.0827	**0.0906**	**0.0874**	0.0691	**0.0942**
	DV**	0.0768(σ=0.0137)	0.0733(σ=0.0132)	0.0811(σ=0.0161)	0.0750(σ=0.0144)	0.0803(σ=0.0143)
	DV**+FED	**0.0881**(σ=0.0128)	0.0882(σ=0.0118)	0.0857(σ=0.0118)	**0.0862**(σ=0.0113)	0.0902(σ=0.0121)
ArgoUML v0.26.2	DV*	0.0847	0.0890	0.0813	0.0834	0.0805
	DV**	0.0726(σ=0.0025)	0.0751(σ=0.0060)	0.0755(σ=0.0054)	0.0721(σ=0.0062)	0.0717(σ=0.0060)
	DV**+FED	**0.1036**(σ=0.0099)	**0.0989**(σ=0.0085)	**0.1025**(σ=0.0087)	**0.1035**(σ=0.0081)	**0.0994**(σ=0.0079)

<DV* = Experimento Referência || DV** = Reprodução do Experimento Referência>

D.0.1 Resultados Preliminares

Os resultados obtidos a partir da aplicação desta abordagem foram compilados na Tabela 18. Os resultados do Estudo de Referência e da abordagem aqui proposta foram assinalados com fundo cinza. Já os melhores resultados em valores absolutos foram colocados com a fonte em negrito.

No total de resultados mostrados na Tabela 18 a abordagem de enriquecimento semântico é responsável por 80% dos melhores valores obtidos se levarmos em consideração cada combinação (*software versus* K). Contudo o Estudo de Referência ainda consegue obter valor superior em 20% dos casos com sua abordagem DV*.