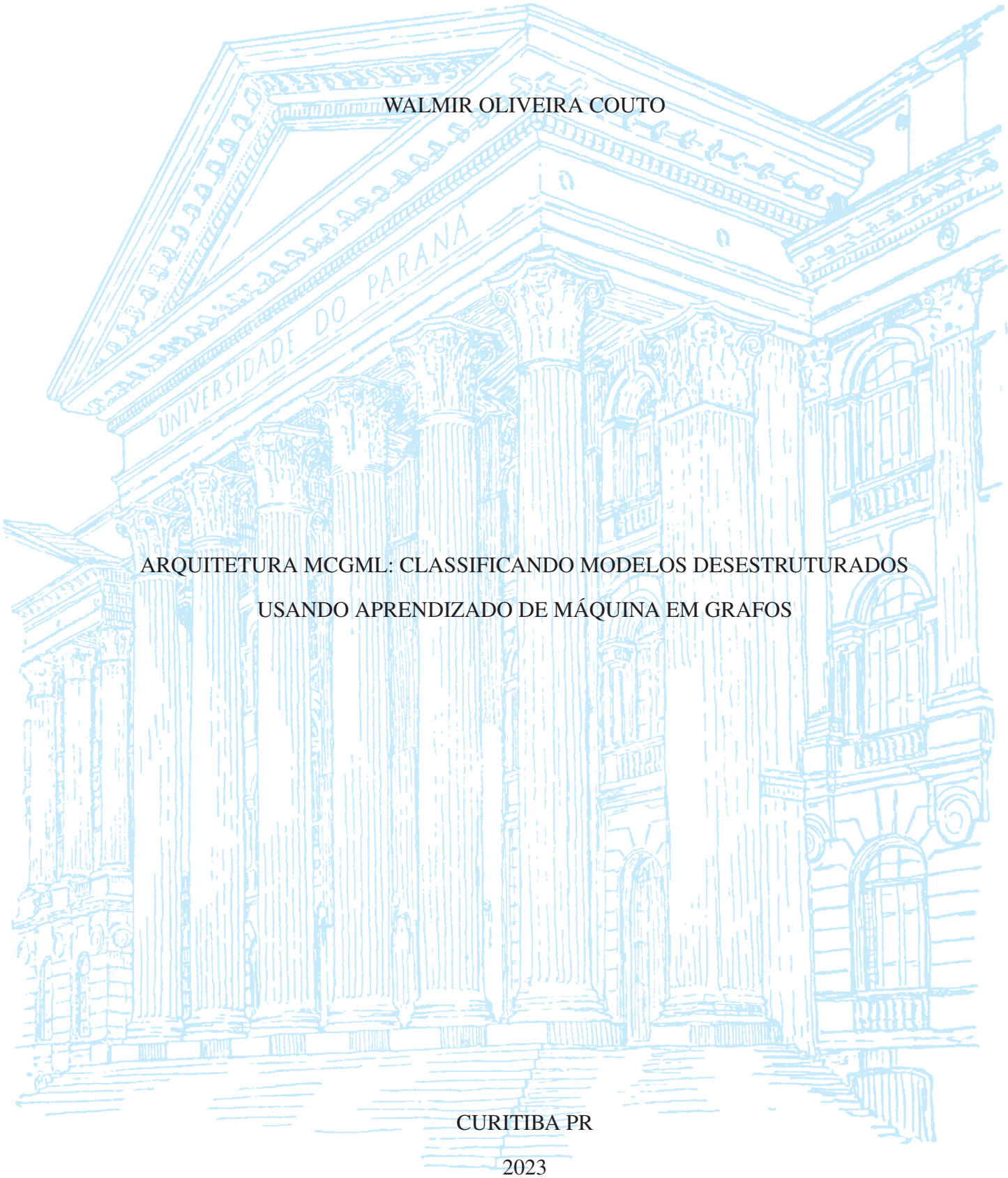
UNIVERSIDADE FEDERAL DO PARANÁ

WALMIR OLIVEIRA COUTO




# ARQUITETURA MCGML: CLASSIFICANDO MODELOS DESESTRUTURADOS USANDO APRENDIZADO DE MÁQUINA EM GRAFOS

CURITIBA PR

2023

WALMIR OLIVEIRA COUTO

# ARQUITETURA MCGML: CLASSIFICANDO MODELOS DESESTRUTURADOS USANDO APRENDIZADO DE MÁQUINA EM GRAFOS

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Andrey Ricardo Pimentel.

CURITIBA PR

2023

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

Couto, Walmir Oliveira
Arquitetura MCGML: classificando modelos desestruturados usando aprendizado de máquina em grafos / Walmir Oliveira Couto. – Curitiba, 2023.
1 recurso on-line : PDF.

Tese (Doutorado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.

Orientador: Andrey Ricardo Pimentel

1. Aprendizado do computador. 2. Sistemas auto-organizadores. 3. Inteligência artificial. 4. Grafos. I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Pimentel, Andrey Ricardo. IV. Título.

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

# TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **WALMIR OLIVEIRA COUTO** intitulada: **ARQUITETURA MCGML: CLASSIFICANDO MODELOS DESESTRUTURADOS USANDO APRENDIZADO DE MÁQUINA EM GRAFOS**, sob orientação do Prof. Dr. ANDREY RICARDO PIMENTEL, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 27 de Março de 2023.

Assinatura Eletrônica
28/03/2023 16:52:17.0
ANDREY RICARDO PIMENTEL
Presidente da Banca Examinadora

Assinatura Eletrônica
28/03/2023 11:15:22.0
EMERSON CORDEIRO MORAIS
Avaliador Externo (UNIVERSIDADE FEDERAL RURAL DA AMAZÔNIA)

Assinatura Eletrônica
28/03/2023 13:06:32.0
EDUARDO TODT
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
28/03/2023 13:19:54.0
FABIO PAULO BASSO
Avaliador Externo (UNIVERSIDADE FEDERAL DO PAMPA)

*“Cada sonho que você deixa pra trás,*
*é um pedaço do seu futuro que deixa*
*de existir!”*
*– Steve Jobs*

## AGRADECIMENTOS

Agradeço aos meus pais, Walmir Couto e Lucimar Couto, às minhas irmãs Geyna e Taiana pelo apoio durante esse tempo de estudo. Também agradeço a minha avó Maria de Lourdes (*In memoriam*) por todo incentivo em buscar sempre a realização através da educação qualificada. Esta conquista é para vocês: minha abençoada família.

Agradecer ao meu orientador Prof. Dr. Marcos Didonet Del Fabro pela excelente condução deste trabalho, auxiliando em todas as dúvidas dentro e fora do contexto da pesquisa, além do exemplo de profissionalismo os quais tomo como inspiração para minha carreira. Ao Prof. Dr. Andrey Ricardo Pimentel por ter aceito conduzir as orientações finais deste trabalho em uma atitude de profissionalismo e cordialidade para com o Prof. Didonet, meu muito obrigado por todo apoio! Agradeço também aos professores do departamento que compartilharam dicas valiosas durante os cafés e reuniões, em especial ao Prof. Dr. Eduardo Cunha, ao Prof. Dr. Luiz Eduardo e ao Prof. Dr. Marco Antonio Zanata. Gostaria de agradecer ao meu amigo Prof. Dr. Emerson Cordeiro Morais da Universidade Federal Rural da Amazônia - UFRA pelo apoio incondicional nas dúvidas sobre Redes Neurais e Machine Learning.

Aos membros da banca, Prof. Dr. Eduardo Todt, Prof. Dr. Emerson Morais, e o Prof. Dr. Fábio Basso, expresso aqui meus sinceros agradecimentos pelas valiosas contribuições.

Por fim, mas não menos importante, um agradecimento aos colegas do departamento pelas variadas conversas, muitas vezes descontraídas, as quais com certeza me ajudaram a enxergar diferentes pontos de vista e evoluir como ser humano: André Hochuli, Cides Bezerra, Daniel Ruiz, Diego Tomé, Hegler Tissot, Jeovane Alves, Leandro Pulgatti, Marcelino Abel, Matheus Nerone, Paulo Almeida, Pedro Holanda, Ramiro Lucas, Renato Melo e Teruo Maruyama.

E em uma atitude de Fé, agradeço a Deus por me conduzir nessa jornada e me dar forças para conquistar o objetivo.

## RESUMO

Modelos e metamodelos criados usando abordagens baseadas em modelos têm relações de conformidades restritas. No entanto, houve um aumento nos formatos de dados livres de *schemas* ou semiestruturados, tais como as representações orientadas a documentos, as quais geralmente são persistidas como documentos JSON. Apesar de não terem um metamodelo/*schema* explícitos, esses documentos poderiam ser categorizados visando obter conhecimento sobre seus domínios, e conformá-los parcialmente aos seus respectivos metamodelos. Abordagens recentes estão surgindo objetivando extrair informações ou combinar soluções de modelagem com *cognificação*. Existe porém uma carência de abordagens que explorem a classificação de formatos semiestruturados. Esta tese aborda justamente essas limitações, apresentando a Arquitetura MCGML (*Model Classification using Graph Machine Learning*): uma abordagem para analisar e classificar modelos desestruturados usando Aprendizado de Máquina em Grafos. Primeiro descrevemos nossa primeira etapa de pesquisa que diz respeito ao processo de extração de elementos de um metamodelo para dentro de uma rede neural Multi-Layer Perceptron (MLP) para que a mesma possa ser treinada. Em seguida convertemos documentos JSON em um formato de entrada codificado para a MLP. Apresentaremos nesta tese, o passo a passo para classificar documentos JSON de acordo com metamodelos existentes extraídos a partir de um repositório. Em seguida apresentamos a Arquitetura MCGML com o objetivo de classificar e analisar similaridades em modelos e metamodelos através da análise de grafos e os algoritmos de Machine Learning (ML). Mostramos também uma série de experimentos para demonstrar a viabilidade da arquitetura MCGML e sua efetividade em classificar documentos.



Palavras-chave: classificando modelos desestruturados, reconhecimento de modelos, aprendizado de máquina em grafos

## ABSTRACT

Models and metamodels created using model-based approaches have restrict conformance relations. However, there has been an increase of semi-structured or schema-free data formats, such as document-oriented representations, which are often persisted as JSON documents. Despite not having an explicit schema/metamodel, these documents could be categorized to discover their domain and to partially conform to a metamodel. Recent approaches are emerging to extract information or to couple modeling with cognification. However, there is a lack of approaches exploring semi-structured formats classification. In this thesis, we address precisely these limitations, we present MCGML Architecture (Model Classification using Graph Machine Learning): an approach to analyze and classify unstructured models using Graph Machine Learning. First, we describe our first research stage that concerns how to extract metamodels elements into a Multi-Layer Perceptron (MLP) network to be trained. Then, we translate the JSON documents into the input format of the encoded MLP. We present the step-by-step tasks to classify JSON documents according to existing metamodels extracted from a repository. Then we present the MCGML Architecture in order to classify and analyze similarities in models and metamodels through graph analysis and Machine Learning (ML) algorithms. We have conducted a set of experiments, showing that the approach is feasible, and its effectiveness in classifying documents.



Keywords: classifying unstructured models, model recognition, machine learning in graph

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE ACRÔNIMOS

| | |
|---|---|
| AURORA | Classificador Automatizado de Repositórios de Metamodelos usando Redes Neurais |
| BFS | Breadth-first Sampling |
| CNN | Convolutional Neural Network |
| CMOF | complete MOF |
| CrossSim | Cross Project Relationships for Computing Open Source Software Similarity |
| DFS | Depth-first Sampling |
| DTD | Document Type Definition |
| DSLs | Domain Specific Languages |
| DSVL | Domain Specific Visual Language |
| EMF | Eclipse Modeling Framework |
| EMOF | essential MOF |
| EMP | Eclipse Modeling Project |
| FN | False Negative |
| FOSS | Free and Open-Source Software |
| FP | False Positive |
| FPR | False Positive Rate |
| GBDT | Gradient Boosted Decision Tree |
| GCN | Graph Convolutional Networks |
| GEF | Graphical Editing Framework |
| GMF | Graphical Modeling Framework |
| GML | Graph Machine Learning |
| IA | Inteligência Artificial |
| LDA | Latent Dirichlet Allocation |
| LSTM | Long-Short Term Memory |
| MAG | Microsoft Academic Graph |
| MBE | Model-Based Engineering |
| MCGML | Model Classification using Graph Machine Learning |
| MDA | Model-Driven Architecture |
| MDD | Model-Driven Development |
| MDE | Model Driven Engineering |
| MDSE | Model Driven Software Engineering |
| ML | Machine Learning |

| | |
|---|---|
| MLP | Multi-Layer Perceptron |
| MOF | Meta Object Facility |
| NLP | Natural Language Processing |
| OHE | One-hot Encoding |
| OMG | Object Management Group |
| OOM | Object-Oriented Modeling |
| OSS | Open Source Software |
| PPGInf | Programa de Pós-graduação em Informática |
| RNN | Recurrent Neural Network |
| ROC | Receiver Operator Characteristic |
| RNA | Redes Neurais Artificiais |
| SNAP | Stanford Network Analysis |
| SOAs | Service-Oriented Architectures |
| SVM | Support Vector Machines |
| TDM | Term Document Matrix |
| TMF | Textual Modeling Framework |
| TN | True Negative |
| TP | True Positive |
| TPR | True Positive Rate |
| UFPR | Universidade Federal do Paraná |
| UML | Unified Modeling Language |
| XMI | XML Metadata Interchange Format |
| XMI[DI] | XMI Diagram Interchange |
| XML | Extensible Markup Language |

**SUMÁRIO**

# 1 INTRODUÇÃO

O capítulo introdutório define a organização desta tese, e tem como objetivo apresentar a pesquisa através da contextualização do problema, da motivação, dos objetivos, bem como das soluções sugeridas e das contribuições dos estudos.

## 1.1 CONSIDERAÇÕES INICIAIS

Nas engenharias, modelos são usados como o principal artefato para extrair o conhecimento em relação a diferentes domínios (Bézivin, 2005), este campo científico é conhecido como engenharia orientada a modelo (*Model Driven Engineering* - MDE). Um exemplo disso é a engenharia de software orientada a modelos (*Model Driven Software Engineering* - MDSE), onde utilizamos a modelagem como uma forma de produzir - de maneira semi-automatizada - sistemas computacionais (Brambilla et al., 2012; Lano, 2012). Dentro deste contexto, modelos e metamodelos criados, usando abordagens baseadas em modelos, possuem uma relação de conformidade restrita. Porém temos visto um aumento de formatos de dados semiestruturados ou livres de *schema*, tais como as representações orientadas a documentos, que normalmente são criadas como documentos JSON. Mesmo não sendo um *schema* de metamodelos explícito, esses documentos poderiam ser categorizados com o objetivo de descobrir os domínios os quais pertencem e, de certa forma, deixá-los em conformidade com um determinado metamodelo. Para isso, abordagens recentes sugerem tratar a extração de informações, ou inter-relacionar a área da modelagem com a área da cognificação.

Grace et al. (2017) afirmou em sua pesquisa que "nos próximos dez anos, a Inteligência Artificial (IA) superará os humanos em várias atividades, tais como: na tradução de línguas, na realização de trabalhos de pesquisa do colégio, ou mesmo realizando atividades de um cirurgião". E dentro deste contexto da IA, Xie (2018) introduz conceitos de Softwares Inteligentes, e Cabot et al. (2017) introduz o conceito da Engenharia de Software Cognificada. As técnicas de Aprendizado de Máquina (*Machine Learning* - ML), mais precisamente os métodos de aprendizado supervisionado, possibilitam que máquinas aprendam padrões a partir de fontes de dados existentes, e dessa forma, essas máquinas são capazes de realizar predições relacionadas a novos dados. Ou seja, com base em um determinado conjunto de dados de entrada e saída, um algoritmo de ML seria capaz de aprender o mapeamento a partir de exemplos de entrada e sua respectiva saída, e dessa forma, esse algoritmo seria capaz de predizer, a partir de uma nova entrada, qual seria sua saída correspondente (Wu et al., 2016). O trabalho de Burgueño (2019) propõe utilizar as vantagens da Inteligência Artificial, em particular das Redes Neurais de Memória Curta e Longa (*Long-Short Term Memory* - LSTM), para automatizar o processo de inferência de uma transformação de modelo a partir de um conjunto de pares de modelos de entrada e de saída. Dessa forma, uma vez que o mapeamento da transformação foi aprendido pela rede LSTM, o sistema da rede neural estaria apto a transformar automaticamente novos modelos de entrada em seus respectivos modelos de saída, sem a necessidade de escrever nenhuma linha de código específica para realizar a transformação correspondente.

Ao longo dos últimos anos vários repositórios de metamodelos surgiram em resposta às necessidades da comunidade da Engenharia Orientada a Modelos (*Model-Driven Engineering* - MDE) por sistemas avançados de apoio ao reuso de artefatos de modelagem. Fazer a gestão de um grande número de artefatos de modelagem é um trabalho que exige grande esforço no gerenciamento e no reuso de artefatos armazenados em repositórios de modelos. A clusterização

de software é uma área de pesquisa que possui inúmeras aplicações nos problemas de engenharia reversa e manutenção de software. Ela consiste basicamente em promover a categorização automatizada dos artefatos de software (tais como: classes, funções ou arquivos) dentro de estruturas de alto nível baseadas em suas similaridades (Beck and Diehl, 2013). No contexto da Mineração de Dados (*Data Mining*), Classificação e Clusterização são operações importantes utilizadas para obter conhecimento detalhado sobre dados disponíveis, e para identificar padrões repetitivos. O trabalho realizado por Basciani et al. (2016) propõe a aplicação de técnicas de clusterização para organizar automaticamente metamodelos armazenados, e com isso fornecer aos usuários uma visão geral dos domínios de aplicação cobertos pelos metamodelos disponíveis, onde metamodelos mutuamente similares são agrupados com base em uma medida de proximidade cuja definição pode ser feita de acordo com uma pesquisa específica e requisitos de navegação.

Nguyen et al. (2019) propuseram, recentemente, aplicar técnicas de *Machine Learning* para classificar automaticamente metamodelos presentes em repositórios de metamodelos, onde os algoritmos de ML tentam simular as atividades do aprendizado humano com o objetivo de adquirir conhecimento sobre o mundo real de maneira autônoma. Eles criaram a ferramenta AURORA (um Classificador Automatizado de Repositórios de Metamodelos usando Redes Neurais) que implementa uma rede neural *feed-forward* capaz de classificar metamodelos a partir de um conjunto de treinamento com metamodelos identificados. E após este processo de treinamento, o classificador AURORA é capaz de classificar metamodelos não identificados. Porém esta ferramenta apresenta como limitação a não possibilidade de classificar modelos desestruturados em relação a possíveis similaridades com metamodelos existentes.

Outra área de pesquisa que vem sendo estudada na tentativa de ampliar a representação do aprendizado em IA é a área de *Machine Learning* (ML) em grafos. Porém o principal desafio nesta área é encontrar uma forma de representar ou codificar (*encoder*) informações sob a forma de estruturas em grafos que possam ser facilmente exploradas pelos modelos de *machine learning*, pois não há uma forma direta para codificar as informações que estão em uma estrutura em grafo para um vetor de características que possa ser facilmente utilizado pelos algoritmos de ML. Em virtude das dificuldades das abordagens de representação do aprendizado em capturar a diversidade dos padrões de conectividade em grafos, Grover and Leskovec (2016) propuseram *node2vec*, um *framework* algorítmico para aprender de forma contínua a representar características dos vértices (nodos) de um grafo. Este *framework* realiza o mapeamento dos vértices em um espaço vetorial com características de dimensionalidade baixa capaz de maximizar a probabilidade de preservar as vizinhanças dos vértices do grafo através da definição de uma noção flexível para essas vizinhanças, e um procedimento de caminhada aleatória tendenciosa (*biased random walk*) capaz de explorar de maneira eficiente diversas vizinhanças, e assim obter melhores representações do aprendizado. E este espaço vetorial de dimensionalidade baixa pode ser utilizado como recurso de entrada para os algoritmos de ML.

Vale lembrar que essas abordagens apresentam limitações, principalmente quando falamos em classificação de modelos desestruturados de acordo com metamodelos correspondentes. E com o objetivo de oferecer contribuições na tentativa de diminuir essas limitações, e explorar esse campo aberto de pesquisa, esta tese teve inicio com o autor deste trabalho (Couto. et al., 2020) apresentando uma metodologia para analisar e classificar documentos JSON tendo como base metamodelos existentes. Para isso, extraímos metamodelos existentes, utilizando uma solução *One-hot encoding*, para uma rede neural *Multi-Layer Perceptron* (MLP), traduzindo os elementos do metamodelo em neurônios de entrada da rede. Essa rede neural foi treinada e então utilizada para classificar documentos JSON de entrada, os quais são também traduzidos em dados de entrada para serem classificados.

Esta tese explora justamente esse campo de pesquisa que aborda a classificação de modelos, apresentando a arquitetura MCGML (*Model Classification using Graph Machine Learning*) para classificar modelos desestruturados usando aprendizado de máquina em grafos, capaz de realizar análises de similaridades entre diferentes metamodelos, utilizando como *framework driver* o *Stanford Network Analysis* - SNAP e o algoritmo *node2vec* (Grover and Leskovec, 2016). Demonstramos através dos experimentos realizados e dos resultados obtidos que a arquitetura proposta pode contribuir para mitigar o problema das análises de similaridades e inferências em diferentes metamodelos, ou mesmo em repositórios de metamodelos, e com isso auxiliar na precisão da classificação de modelos desestruturados em metamodelos correspondentes através dos algoritmos de ML em grafos.

Exploramos o poderio de análise em larga escala da biblioteca SNAP (em Python) para realizar uma análise detalhada e comparativa dos efeitos que as configurações de parâmetros do algoritmo *node2vec* podem gerar no processo de análise de similaridades através das caminhadas aleatórias tendenciosas, melhorando assim a preparação do espaço vetorial de baixa dimensionalidade que é usado como recurso de entrada para os algoritmos de classificação de ML em grafos.

## 1.2 MOTIVAÇÃO

Com a crescente adoção da MDE, o problema de classificação de modelos desestruturados em metamodelos correspondentes vem sendo tratado como um dos principais gargalos na busca por agilidade no processo de desenvolvimento de software orientado a modelo (MDSE), e como uma limitação crítica de várias abordagens (Tisi et al., 2011).

A motivação desta tese está: *(i)* na contribuição com os estudos na área de classificação de modelos, *(ii)* na exploração dos recursos disponíveis na plataforma SNAP em busca de desempenho e escalabilidade nas análises de grandes modelos em grafos, *(iii)* no estudo e análise dos parâmetros e funcionalidades do algoritmo *node2vec* em relação a inferências de similaridades entre elementos do grafo realizadas através das caminhadas aleatórias tendenciosas (criando a nossa própria instanciação do *node2vec*), *(iv)* e na análise em até que ponto podemos atenuar o problema de classificação de modelos desestruturados em metamodelos correspondentes utilizando a preparação do espaço vetorial de baixa dimensionalidade (*embedding*) para ser utilizado como recurso de entrada para os algoritmos de ML em grafos.

## 1.3 PERGUNTAS DE PESQUISA

Conforme visto na seção 1.1, as abordagens que tratam a classificação de documentos ou modelos desestruturados apresentam limitações relacionadas a precisão e desempenho, principalmente quando tentamos classificar modelos que podem conter elementos de diferentes domínios de negócio. Já existem iniciativas para executar a tarefa de classificação utilizando recursos da IA, ou inter-relacionar com área da cognificação. Os gargalos estão intimamente ligados ao problema de extração e preparação do espaço vetorial que irá servir de recurso de entrada para os treinamentos com redes neurais, ou mesmo para os algoritmos de ML.

Passos importantes na tentativa de apresentar contribuições ao problema de execução de classificação de modelos ou mesmo artefatos de software em repositórios foram dados por Nguyen et al. (2019); Capiluppi et al. (2020). Nguyen et al. (2019) aplicam técnicas de *Machine Learning* para classificar automaticamente metamodelos presentes em repositórios de metamodelos, porém não conseguem classificar modelos desestruturados em relação a possíveis similaridades com metamodelos existentes. Já Capiluppi et al. (2020) aplicam técnicas de clusterização onde uma

amostra de sistemas de software pode ser dividida em subconjuntos (ou *clusters*), onde cada um desses *clusters* conteria sistemas similares, exibindo as diferenças existentes com outros *clusters*, porém esta abordagem é focada na classificação de artefatos de software e não em modelos.

Dentro desta contextualização, esta tese apresenta o seguinte problema de pesquisa: é possível melhorar a precisão na classificação e o tempo de execução nas tarefas de classificar modelos desestruturados em metamodelos existentes utilizando a arquitetura baseada ML em grafos?

Modularizando este problema de pesquisa, nos deparamos com outros problemas envolvidos neste cenário, dentre os quais destacamos:

- Como viabilizar a extração dos elementos de um modelo desestruturado para grafos com a finalidade de ser analisado utilizando como *framework driver* o *Stanford Network Analysis* - SNAP e o algoritmo *node2vec*?
- Como trabalhar os modelos desestruturados e os metamodelos em grafos para serem transformados em vetores de baixa dimensionalidade (*embedding*) para que possam ser utilizados como recurso de entrada para os algoritmos de ML em grafos?
- Como executar as tarefas de classificação e clusterização de artefatos de metamodelos utilizando a arquitetura MCGML proposta?

## 1.4 OBJETIVOS

Visando alcançar de forma efetiva o desenvolvimento desta pesquisa, estabelecemos o seguinte objetivo geral seguido de seus desdobramentos.

### 1.4.1 Objetivo Geral

O objetivo geral desta pesquisa é desenvolver a arquitetura MCGML para classificar modelos desestruturados, realizar análises de similaridades entre artefatos de metamodelos em repositórios utilizando aprendizado de máquina em grafos.

### 1.4.2 Objetivos Específicos

Modularizando a arquitetura proposta, estruturamos as seguintes etapas para alcançar o objetivo geral e validar a pesquisa:

- Pesquisar sobre análise de similaridades de modelos em grafos utilizando o algoritmo *node2vec*;
- Pesquisar como configurar as passagens de parâmetros para realizarmos corretamente as caminhadas aleatórias tendenciosas na busca de inferências entre os elementos do grafo;
- Criar o vetor de baixa dimensionalidade (realizar os *embeddings*) a partir das análises com o *node2vec* para servir de entrada para os algoritmos de ML em grafos na tarefa de classificação, clusterização de modelos desestruturados de acordo com artefatos de metamodelos existentes em repositórios;
- Viabilizar o processo de extração, *encoder*, e *decoder* dos modelos e metamodelos para a arquitetura MCGML;

- Validar ganhos de precisão e desempenho em relação a tarefa de classificação e clusterização, comparando alguns tipos de configurações e parâmetros obtidos com o uso da arquitetura MCGML;
- Comparar o desempenho e a precisão dos métodos de aprendizado presentes na arquitetura MCGML com trabalhos que realizam classificação de metamodelos em repositórios, e como podemos integrar a arquitetura MCGML como uma ferramenta de classificação e clusterização em repositórios de metamodelos.

## 1.5 METODOLOGIA

De modo geral, a metodologia de pesquisa é o caminho que o trabalho vai seguir para alcançar seus objetivos. A metodologia é um conjunto de processos de uma pesquisa com o objetivo de se obter clareza sobre os processos e os modo de execução dos trabalhos envolvidos. A partir da metodologia, a pesquisa ganha autenticidade, confiabilidade e valor científico. A metodologia de pesquisa nada mais é do que a descrição do processo de pesquisa do trabalho. Isto é, a definição de quais serão os procedimentos para a coleta e para a análise da dados. Delimitar uma metodologia não é adotar um conjunto de diretrizes fixas que se deve seguir à risca, mas construir um procedimento de pesquisa que se adapte aos objetivos do trabalho.

A metodologia de pesquisa utilizada nesta tese está baseada na caracterização do problema de pesquisa: realizar tarefas de classificação e análises de similaridades de artefatos de modelos e metamodelos utilizando *Machine Learning* em grafos. Onde o conceito da Arquitetura MCGML proposta engloba a atividade mental que conduz um conhecimento visando obter respostas para as perguntas de pesquisa descritas na Seção 1.3. Os métodos científicos utilizados referem-se a estratégias de investigação da pesquisa, fazendo uso do método experimental de observação e medição, e do método hipotético-dedutivo. E como base nas metodologias adotadas definimos os seguintes estágios de pesquisa:

- Delimitação do Problema;
- Levantamento de Dados (definindo o *benchmark*);
- Avaliação Experimental;
- Análise dos Resultados Obtidos;
- Conclusões.

As fases da pesquisa (seu *roadmap*) estão ilustradas na Figura 1.1.

## 1.6 CONTRIBUIÇÕES DO ESTUDO

Citamos anteriormente que as abordagens que oferecem recursos de classificação de modelos e metamodelos apresentam limitações, principalmente quando falamos em classificação de modelos desestruturados de acordo com metamodelos correspondentes.

Podemos destacar como contribuições desta pesquisa o trabalho realizado pelo autor desta tese e seus orientadores (Couto. et al., 2020) apresentando uma metodologia para analisar e classificar documentos JSON tendo como base metamodelos existentes, extraindo metamodelos existentes, utilizando uma solução *One-hot encoding*, para uma rede neural *Multi-Layer Perceptron* (MLP), traduzindo os elementos do metamodelo em neurônios de entrada

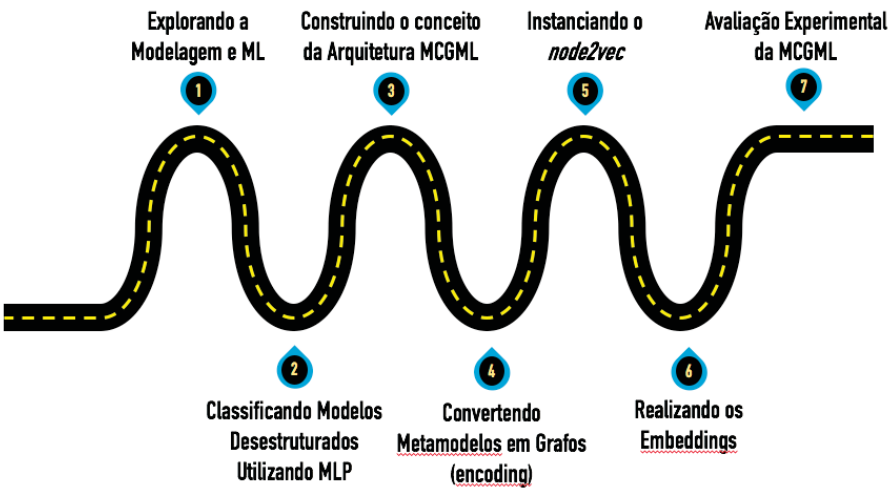



Figura 1.1: As fases da pesquisa - o *roadmap* da tese.

da rede. Além disso, desenvolvemos a arquitetura MCGML, tendo como *framework driver* o *Stanford Network Analysis* - SNAP e o algoritmo *node2vec* (Grover and Leskovec, 2016), demonstrando através dos experimentos realizados e dos resultados obtidos que a arquitetura proposta pode contribuir para mitigar o problema das análises de similaridades e inferências em diferentes metamodelos, ou mesmo em repositórios de metamodelos, melhorando o processo de classificação e clusterização de modelos, ou até mesmo de documentos desestruturados. Dessa forma colaboramos com a comunidade científica da área MDSE na busca pela expansão do uso de soluções baseadas em modelos não apenas na implementação de um classificador de modelos desestruturados; mas na integração com outras abordagens já existentes.

## 1.7 ESTRUTURA DO DOCUMENTO

Esta tese tem como objetivo apresentar os estudos realizados junto ao Programa de Pós-graduação em Informática (PPGInf) da Universidade Federal do Paraná (UFPR) visando a obtenção do grau de doutor em Ciência da Computação.

A organização do documento está estruturada em mais 6 capítulos. Após o Capítulo 1, que apresenta a introdução e a proposta da pesquisa, temos o Capítulo 2: fundamentação teórica, onde fizemos uma revisão bibliográfica em relação ao estado da arte com os conceitos que abordam os aspectos necessários para esclarecer e embasar a pesquisa que foi realizada durante os estudos. O Capítulo 3 apresenta trabalhos relacionados às áreas de redes neurais, clusterização de metamodelos, técnicas de ML aplicadas à classificação de metamodelos, representação do aprendizado em grafos, e técnicas de clusterização aplicadas à detecção de similaridades. Já o Capítulo 4 traz as abordagens desenvolvidas pelo o autor desta tese (Couto. et al., 2020) para extração e classificação de modelos e análises de similaridades, com destaque à apresentação da arquitetura MCGML com as atividades desenvolvidas. No Capítulo 5 apresentamos nossas avaliações experimentais da arquitetura MCGML, demonstrando o *benchmark* aplicado, as métricas utilizadas, e os resultados obtidos. No Capítulo 6 apresentamos as discussões gerais e as contribuições de pesquisa, com as análises das avaliações experimentais, e as referências a trabalhos relacionados nas áreas da MDE, classificação e categorização de metamodelos, e aplicações envolvendo redes neurais, apresentando também as publicações obtidas. E por último, no Capítulo 7 temos as conclusões e as considerações finais para o desenvolvimento da tese e os caminhos a serem seguidos para a expansão da arquitetura MCGML proposta.

# Parte I

# Fundamentação Teórica

# 2 INTELIGÊNCIA ARTIFICIAL

Neste capítulo apresentamos uma introdução aos conceitos das abordagens chave para a nossa metodologia relacionadas ao universo das Redes Neurais Artificiais (Seção 2.1), ao Aprendizado Profundo - *Deep Learning* (Seção 2.2), às Redes Neurais Convolucionais e Recorrentes (Seções 2.3 e 2.4), além de conceitos sobre Modelagem (Seção 2.5), sobre os Princípios da *Model-Driven Software Engineering* - MDSE (Seção 2.6), e sobre Metamodelagem (Seção 2.7) e Linguagens de Modelagem (Seção 2.8).

## 2.1 REDES NEURAIS ARTIFICIAIS

Na área de aprendizado de máquina, Redes Neurais Artificiais (RNA's) são bastante difundidas e com vasta aplicação. Um neurônio artificial (perceptron - Rosenblatt (1958)) é constituído pelo seus pesos e *bias*, além da função de ativação que determina se o neurônio está ativo ou não para um dado conjunto de características fornecido como entrada, funcionando assim como um classificador binário linear, conforme ilustra a Figura 2.1. Seguindo esse raciocínio, uma RNA ou "Perceptron Multi-Camadas" (*Multi-Layer Perceptron - MLP*), distribui uma certa quantidade de neurônios artificias interconectados ao longo de uma rede, formando um fluxo unidirecional (*feed-forward*) entre as camadas (Riedmiller, 1994). As camadas intermediárias, também conhecidas como camadas "escondidas", são responsáveis por mapear os dados de um vetor de entrada em um vetor para a camada de saída, através da ativação ou não de neurônios, funcionando assim como extratores de características implícitos. Os parâmetros de cada neurônio (pesos, *bias* e ativação) nas camadas intermediárias, são atualizados por um algoritmo de retro-propagação (*Back Propagation*), em função do erro calculado na camada de saída.

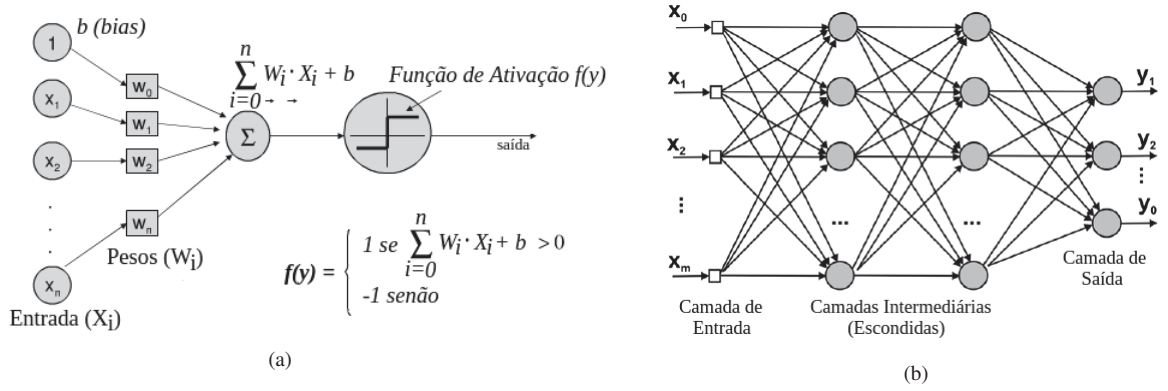



Figura 2.1: (a) Neurônio Artificial (Perceptron) e (b) Rede Neural Artificial ou Perceptron Multi-Camadas (Riedmiller, 1994).

## 2.2 APRENDIZADO PROFUNDO (*DEEP LEARNING*)

Um processo de aprendizado de máquina consiste em definir um conjunto de características para representar um determinado problema, e posteriormente, aplicar um modelo capaz de classificá-las. Deste modo, a representação adequada dos dados definem uma importante etapa deste processo. Basicamente, os métodos tradicionais consistem em predefinir descritores de características desenvolvidos por um humano, ou seja, algoritmos computacionais transformam

características de percepção humana em compreensão de máquina. Assim, esforços têm sido realizados para que técnicas de extração de características representem qualitativamente os dados do problema (*Feature Engineering*), aumentando assim as taxas de reconhecimento dos modelos (Bengio et al., 2013). No entanto, esse tipo de abordagem têm limitado o nível de abstração devido a complexidade dos dados em alguns problemas, tornando pobre a representação para a máquina, e consequentemente, estagnando o avanço das taxas de reconhecimento. Além disso, esse processo é empírico e custoso. Para contornar estes problemas, viu-se a necessidade de criar modelos capazes de extrair as características automaticamente, ao compreendimento da máquina, transformando a descrição pré-definida de características em modelos de descritores treináveis (*Feature Learning*). A Figura 2.2 ilustra a diferença entre as metodologias.

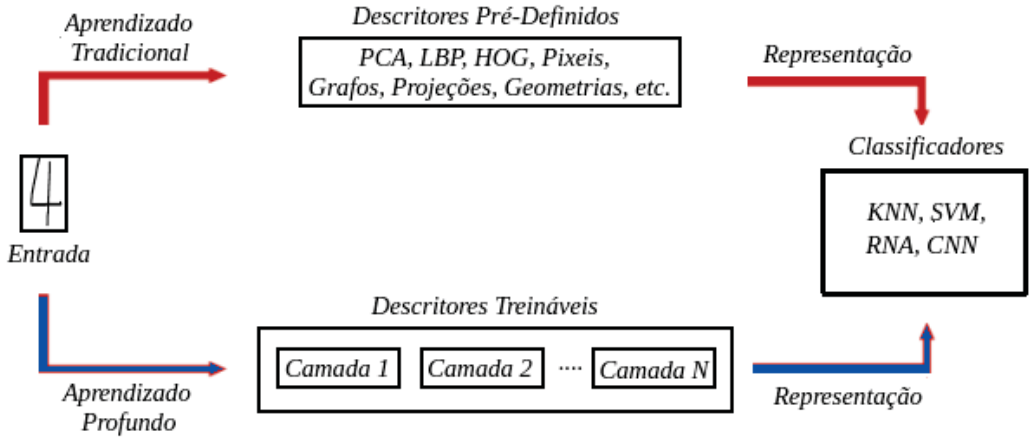



Figura 2.2: Processo de Aprendizado Tradicional e Profundo (Bengio et al., 2013).

Sendo assim, o conceito de aprendizado profundo (Lecun et al., 2015; Schmidhuber, 2015) define arquiteturas densas, treináveis, compostas por várias camadas (Figura 2.3), de forma que a própria máquina consiga extrair características particulares do problema. O dado é repassado entre as camadas, e as camadas mais profundas da arquitetura tendem a representá-lo de forma mais abstrata, ou seja, uma representação mais próxima da máquina do que a apresentada na primeira camada. Alguns exemplos dessas arquiteturas (Bengio, 2009; Deng and Yu, 2014) são: (a) Rede Neurais e Redes Neurais Convolucionais, utilizadas em Aprendizado Supervisionado e, (b) Máquinas de Boltzmann, Rede de Confiança Profunda (*Deep Belief Network*) e Auto-Codificadores (*Auto-Encoder*), aplicadas para mapeamento das características, geralmente aplicadas em Aprendizado Não-Supervisionado. Vale ressaltar que uma abordagem deste tipo, devido a sua densa arquitetura, requer um volume grande de dados para o treinamento eficiente do algoritmo, caso contrário, a solução tenderá a ficar especializada (*overfitting*). É evidente também que essas abordagens têm um custo computacional elevado, devido à quantidade de parâmetros para ajuste e dados processados.

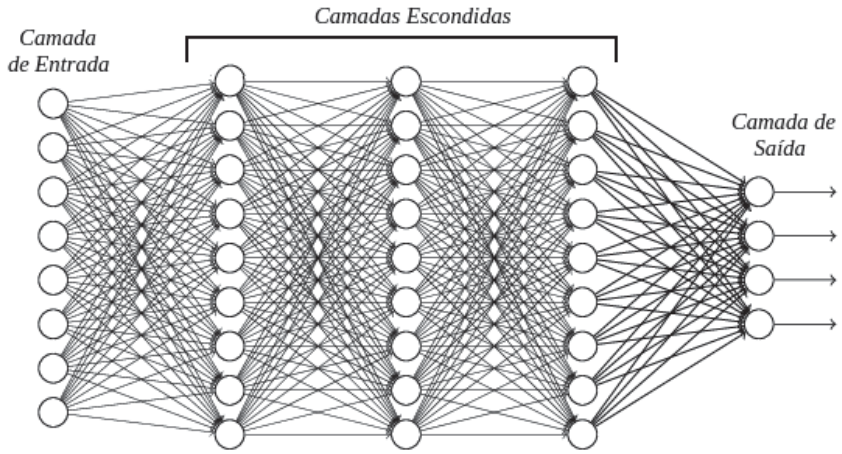



Figura 2.3: Aprendizado Profundo: Estrutura de uma Rede Neural Densa[1] (Schmidhuber, 2015).

## 2.3 REDES NEURAIS CONVOLUCIONAIS (*CONVOLUTIONAL NEURAL NETWORK*)

Como visto na Seção 2.2, uma abordagem baseada em aprendizado profundo (arquiteturas densas) eleva os níveis de abstração dos dados de forma a se aproximar da compreensão pela máquina. Seguindo por este raciocínio, Redes Neurais Convolucionais (*Convolutional Neural Network - CNN*), introduzem o conceito de auto-aprendizagem de filtros, a fim de criar extratores de características para problemas representados de forma matricial, como por exemplo, reconhecimento de padrões em imagens e espectrogramas de áudio (LeCun et al., 2010; Lecun et al., 2015).

Basicamente, existem dois grandes estágios em uma rede convolucional, conforme ilustra a Figura 2.4. O primeiro é responsável por extrair características de uma matriz de dados, comumente uma imagem, fornecida como entrada para a rede. Sua arquitetura é composta de camadas convolucionais (Ciresan et al., 2011), as quais implementam uma sequência de filtros, ativações (ex: ReLU/Sigmoid) e agregações (*pooling's*), mapeando os dados de entrada em novos mapas de características (*feature maps*), criando assim descritores treináveis, abstraindo os dados de acordo com a compreensão da máquina. Ao final desse estágio teremos um modelo que transforma uma matriz de entrada em um vetor de características. A Figura 2.5 apresenta um exemplo de abstração dos dados ao longo da rede.

No segundo estágio, o vetor de características produzido pelas camadas convolucionais é aplicado a um algoritmo de aprendizagem de máquina, como uma RNA ou um *SVM*, determinando assim uma classe para o padrão de entrada (classificação).

Na fase de testes, a imagem de entrada é fornecida à arquitetura da rede neural convolucional, com os pesos e filtros das respectivas camadas calculados durante a fase de treinamento, a qual determinará a classe final do padrão.

Este tipo de abordagem tem elevado os níveis de desempenho do estado da arte, principalmente na última década, antes estagnados pela falta de métodos para representar adequadamente o problema, criando assim um novo marco para as pesquisas nas área de Inteligência Artificial, Reconhecimento de Padrões e Visão Computacional. Dentre as arquiteturas mais conhecidas, citamos os trabalhos de LeCun et al. (1989); Lecun et al. (1998), pioneiros na aplicação de redes convolucionais para reconhecimento de dígitos manuscritos e os recentes trabalhos de Krizhevsky et al. (2012) e Szegedy et al. (2015), trabalhos vencedores nos anos de

[1]Fonte: http://neuralnetworksanddeeplearning.com/chap6.html, acessada em 10/08/2022.

2012 e 2014 do desafio Imagenet (ILSVRC) (Russakovsky et al., 2015), o qual apresenta uma base contendo mais de 1,2 milhões de imagens e 1000 classes.

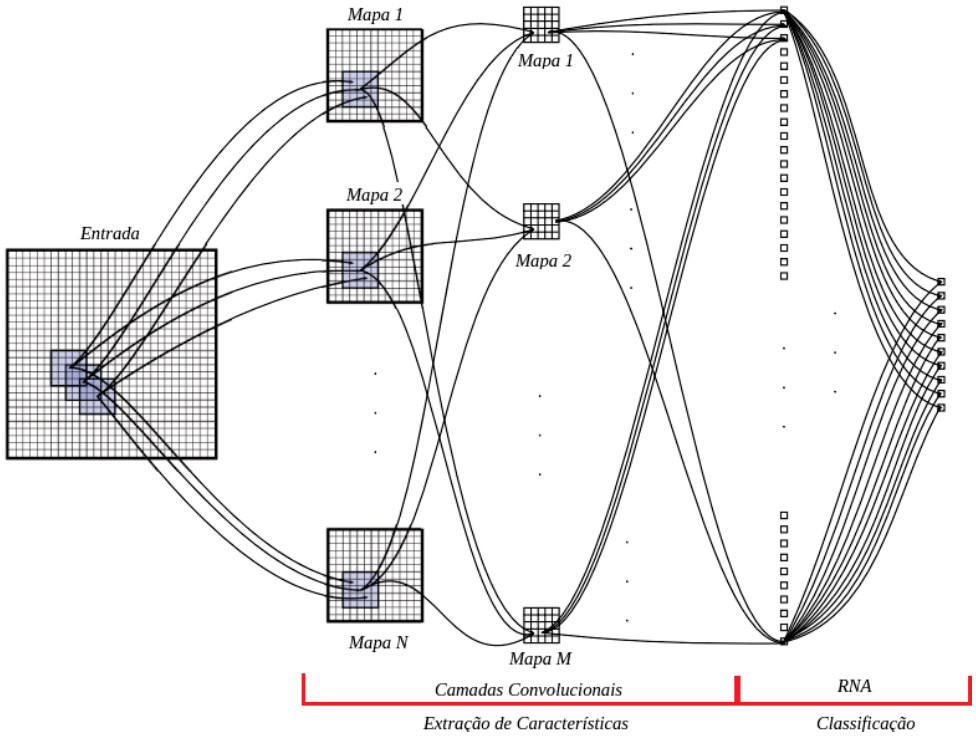



Figura 2.4: Exemplo de arquitetura de uma Rede Neural Convolucional. Os retângulos dentro da matriz representam os filtros/convoluções. (Figura adaptada de (Ciresan et al., 2011))

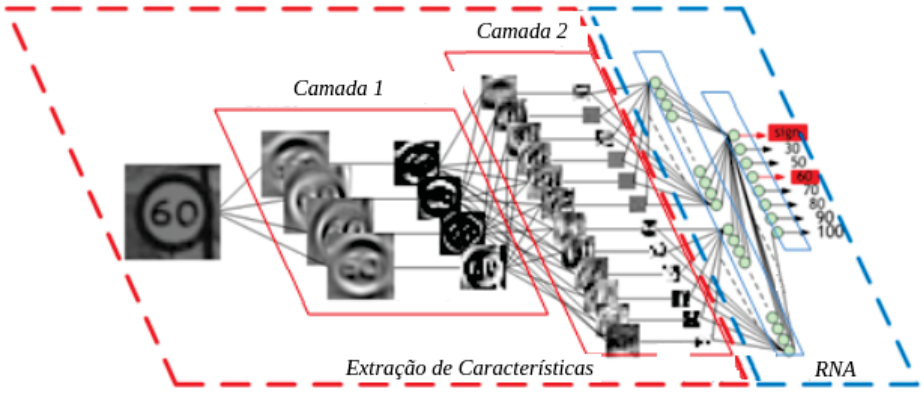



Figura 2.5: Níveis de abstração dos dados em uma rede convolucional de duas camadas[1] (Szegedy et al., 2015).

## 2.4 REDES NEURAIS RECORRENTES (RNN)

Quando se trata de sequências, como por exemplo, processamento de linguagem natural, séries temporais, reconhecimento de voz e sentenças manuscritas, entre outras, a informação sobre o passado é de suma importância para a criação de contexto e classificação. Embora as RNA's sejam capazes de discriminar uma enorme quantidade de classes, a sua mecânica

[1]Fonte: https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-core-concepts/, acessada em 15/07/2022.

não considera a informação produzida em um estado anterior. Uma Rede Neural Recorrente (RNN) (Elman, 1990; Rumelhart et al., 1986; Lipton, 2015), ilustrada na Figura 2.6, resolve esse problema implementando um laço, no qual a informação produzida é re-introduzida no modelo.

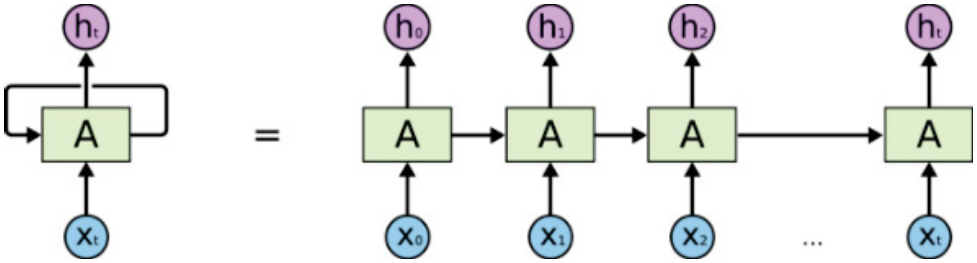



Figura 2.6: RNN: A cada passo a rede ($A$) recebe como entrada a informação nova ($X_t$) e a informação produzida no passo anterior ($h_{t-1}$)[1] (Lipton, 2015).

Seja $A$ uma rede neural, $X_t$ uma sequência qualquer de tamanho $t$, a rede irá produzir $h_t$ estados, considerando sempre a informação produzida no estado $h_{t-1}$. De outro modo, as RNN's podem ser compreendidas como múltiplas cópias da mesma rede, cada uma repassando a informação produzida para a sua sucessora, adicionando uma espécie de memória na rede.

As RNN's se mostraram bastante eficazes no tratamento de sequências, no entanto enfrentam problema quando a informação precisa persistir por um longo tempo. Esse problema é explorado em Bengio et al. (1994); Pascanu et al. (2013), revelando que funções baseadas em minimização do gradiente tendem a perder a informação ao longo das RNN's (*Vanish Gradient Problem*), fazendo com que as RNN's apresentem uma memória curta em relação ao contexto. Para solucionar este problema foram criadas as Redes de Memória Curta e Longa que será descrita na Seção 2.4.1.

### 2.4.1 Redes de Memória Curta e Longa (LSTM)

As Redes de Memória Curta e Longa (*Long-Short Term Memory* - LSTM) (Hochreiter and Schmidhuber, 1997a; Sherstinsky, 2018) são um tipo de RNN capaz de persistir a informação por um longo período. Em uma RNN padrão, os módulos são construídos usando uma simples camada baseada em uma função tangente hiperbólica, tomando como entrada, o padrão atual ($X_t$) e a informação produzida no módulo anterior ($h_{t-1}$). Esse processo é ilustrado na Figura 2.7a.

LSTMs possuem uma estrutura um pouco mais complexa (Figura 2.7b), contendo 4 funções que interagem entre si, além de um segundo fluxo de informação, denominado $C_t$, que representa a memória longa. Esse fluxo de informação transita entre as camadas realizando algumas poucas interações lineares, conforme ilustra a Figura 2.8.

Para determinar a persistência da informação, são utilizados três gatilhos denominados gatilho de Esquecimento ($f_t$), gatilho de Entrada ($i_t$) e gatilho de Saída ($o_t$). Essas funções são detalhadas a seguir e estão representadas na Figura 2.9.

[1]Fonte: http://colah.github.io/posts/2015-08-Understanding-LSTMs/, acessada em 10/09/2022.

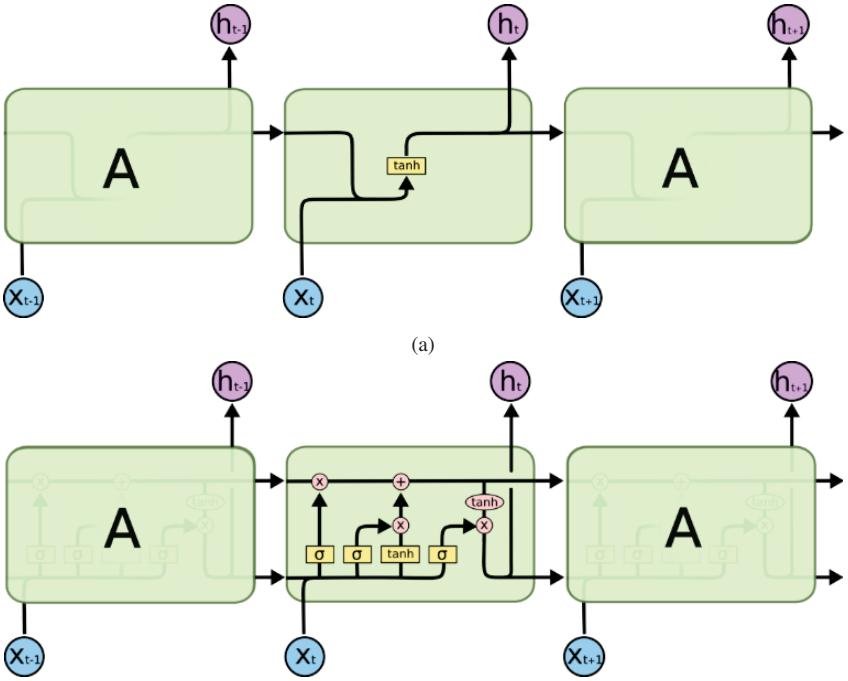



Figura 2.7: RNN x LSTM: (a) RNN apresentando apenas um fluxo de informação, ou memória curta, baseada em seu estado anterior, e (b) LSTM apresentando dois fluxos de informação, ou seja, uma memória curta e outra longa, sendo a longa representado pela linha superior, sendo atualizada pela informação atual e pela memória curta do estado anterior (Sherstinsky, 2018).

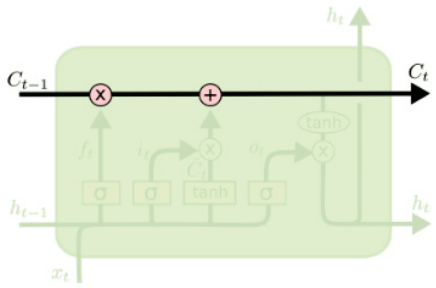


Figura 2.8: Fluxo de Informação $C_t$: A informação transita entre as camadas realizando poucas interações lineares, persistindo a informação por períodos mais longos (Sherstinsky, 2018).

O primeiro passo é decidir qual informação será descartada da memória longa ($C_{t-1}$), o qual é determinado pelo gatilho de esquecimento $f_t$ (Figura 2.9a). Este leva em consideração os dados fornecidos por $h_{t-1}$ e $X_t$, que aplicados a função sigmoide determina um valor entre 0 e 1, no qual, 0 representa "total esquecimento". Posteriormente, o gatilho de entrada $i_t$ (Figura 2.9b) determina o quanto da nova informação será adicionada a $C_t$. Então, $C_t$ é atualizado pelo produto de $f_t$ e a adição dos dados produzidos por $i_t$.

Por fim, o gatilho de saída $o_t$ decide quais dados vão produzir a saída do módulo, baseado nos dados fornecidos por $h_{t-1}$ e $X_t$ e na memória $C_t$ atualizada.

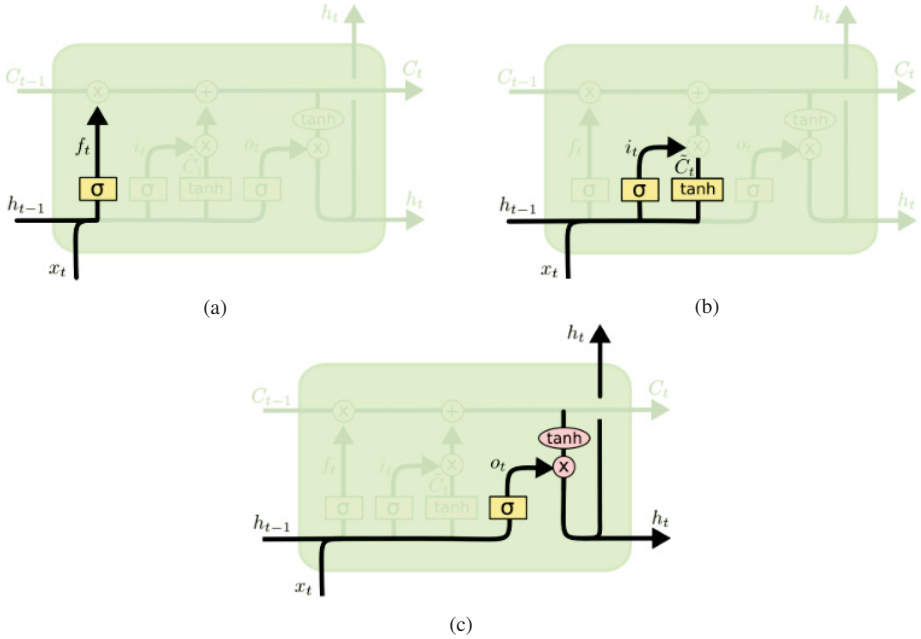



Figura 2.9: Gatilhos LSTM: (a) Esquecimento ($f_t$), (b) Entrada ($i_t$) e (C) Saída ($o_t$) (Sherstinsky, 2018).

## 2.5 MODELAGEM

A mente humana trabalha a realidade continuamente e inadvertidamente aplicando processos cognitivos que alteram a percepção subjetiva da mesma. Entre os vários processos cognitivos aplicados, a abstração é um dos principais. Mas o que é abstração?

Segundo Brambilla et al. (2012), *"abstração consiste na capacidade de encontrar semelhança em observações diferentes, gerando dessa forma uma representação mental da realidade"*, sendo ao mesmo tempo capaz de:

- generalizar características específicas de objetos reais (generalização);
- classificar os objetos dentro de *clusters* coerentes (classificação);
- agregar objetos dentro de outros objetos mais complexos (agregação).

De certa forma, a generalização, a classificação e a agregação representam comportamentos naturais que a mente humana está apta a desenvolver e que os seres humanos praticam todos os dias de suas vidas. Esse processo de abstração é amplamente aplicado nos estudos científicos e na indústria, onde é mais conhecido como modelagem. Informalmente podemos definir um modelo como uma representação simplificada ou parcial da realidade visando a realização de uma tarefa, ou alcançar um entendimento em relação a um determinado problema (Olivé, 2007). Dessa forma, podemos concluir que um modelo nunca descreverá a realidade em sua totalidade (caso contrário não será um modelo).

### 2.5.1 Modelos: finalidades e aplicações

Conforme Batra and M.Marakas (1995) uma enorme ramificação da ciência e da filosofia está baseada em modelos. Pensando em modelos nos níveis abstrato e filosófico, levantamos questões relacionadas à semântica (isto é, a função de representação desenvolvida pelos modelos), à ontologia (isto é, o tipo de coisas que os modelos são), à epistemologia (isto é, como aprender através ou a partir de modelos), e à filosofia.

Já Bézivin (2005) afirma que os modelos são reconhecidos por permitir a implementação de pelos menos dois recursos através da abstração:

1. **Recurso da Redução:** onde os modelos refletem uma seleção relevante das propriedades originais, dando ênfase aos aspectos de interesse;

2. **Recurso do Mapeamento:** nesse caso os modelos são baseados em um artefato original, que é tomado como um protótipo de uma categoria de artefatos e é abstraído e generalizado para um modelo.

Os modelos podem ter diferentes finalidades: eles podem ser usados para fins descritivos (ou seja, para descrever a realidade de um sistema ou de um contexto), para fins prescritivos (ou seja, para determinar o escopo e as fronteiras de estudo de um determinado problema), ou para definir um sistema a ser desenvolvido (Hutchinson et al., 2011).

Dentro do processo produtivo, a modelagem nos permite investigar, verificar, documentar, e discutir propriedades dos produtos antes de eles serem produzidos efetivamente. Em alguns casos, modelos são usados para automatizar diretamente a produção de bens. Diariamente criamos um modelo mental da realidade; de certa forma não podemos evitar a modelagem, isto fica mais evidente quando lidamos com objetos ou sistemas que precisam ser desenvolvidos, onde o desenvolvedor deve ter em mente o modelo dos seus objetivos. Logo, o modelo sempre vai existir, a única opção aos projetistas é com relação a sua forma: podendo ser mental (existindo somente na mente dos projetistas), ou explicitado em forma de artefatos (Brambilla et al., 2012).

### 2.5.2 A Modelagem Aplicada no Desenvolvimento de Software

A área conhecida como Engenharia de Software Orientada a Modelo - *Model-Driven Software Engineering* (MDSE) – explora os processos de modelagem no desenvolvimento de artefatos de software. As práticas da MDSE contribuem para o aumento da eficiência e da eficácia no desenvolvimento de software; podemos dizer que a necessidade de se aplicar a modelagem no desenvolvimento de software está baseada em 4 fatores principais (Acerbis et al., 2007):

1. Artefatos de software estão se tornando cada vez mais complexos e cada vez mais eles precisam ser discutidos em diferentes níveis de abstração (dependendo do perfil dos *stakeholders* envolvidos);

2. Os softwares estão cada vez mais presentes na vida das pessoas, e a expectativa é a de que a necessidade por novos artefatos de software ou a evolução dos artefatos existentes seja aumentada continuamente;

3. O mercado de trabalho sofre escassez contínua de competências envolvidas nos processos de desenvolvimento de software em relação a colocação profissional;

4. O desenvolvimento de software não é uma atividade auto-suficiente: muitas vezes é necessário interações com pessoas que não são da área de desenvolvimento (ex: clientes, gerentes, interessados no negócio, etc.) os quais precisam de alguma mediação na descrição dos aspectos técnicos do desenvolvimento.

Através da modelagem é possível direcionar todas essas necessidades, e isso faz com que as técnicas da MDSE estejam em evidência e cada vez mais colocadas em práticas por profissionais da área, e apoiadas por diferentes agentes de negócios: vendedores, pesquisadores, desenvolvedores de software empresariais e também pelos analistas de negócio Hailpern and Tarr

(2006); haja vista que em cenários onde demandam mais abordagens abstratas do que codificadas - como é o caso das arquiteturas orientadas a serviços (*Service-Oriented Architectures* - SOAs) e das arquiteturas baseadas em nuvem (todas essas integradas ao gerenciamento de processos de negócio) - as técnicas da MDSE contribuem diretamente para um melhor entendimento, trazendo enormes benefícios às empresas, diminuindo as deficiências entre os requisitos de negócio e as implementações da área de TI (Liu and Bourey, 2011).

## 2.6 PRINCÍPIOS DA *MODEL-DRIVEN SOFTWARE ENGINEERING* - MDSE

A MDSE envolve amplos conceitos pertinentes a área da Engenharia Orientada a Modelo - *Model Driven Engineering* (MDE). Segundo Bézivin (2005), a MDE refere-se à sistemática do uso de modelos como artefatos primários ao longo de todo o ciclo de vida do processo de engenharia. Como vimos na seção anterior, modelos são considerados entidades de primeira classe no processo de compreensão de problemas do mundo real, e dessa forma os desenvolvedores são forçados a focar no domínio do problema ao invés de se preocuparem com plataformas de soluções. A complexidade das plataformas de soluções é melhor compreendida através das abordagens orientadas a modelos (Schmidt, 2006).

A MDSE pode ser definida como uma metodologia para aplicar as vantagens da modelagem nas atividades de engenharia de software. Geralmente uma metodologia abrange os seguintes aspectos:

- conceitos;
- notações;
- processos;
- regras;
- e ferramentas.

Os conceitos mais relevantes no contexto da MDSE são: modelos e as transformações (que correspondem às operações de manipulação dos modelos). Lembrando que tanto modelos, quanto as transformações precisam ser expressos através de notações, as quais na MDSE chamamos de linguagens de modelagem (J. Miller and J. Mukerji, 2003). A definição de uma linguagem de modelagem pode ser vista como um modelo: a MDSE refere-se a este processo como metamodelagem (isto é, a modelagem de um modelo, ou melhor, a modelagem de uma linguagem de modelagem, ou a modelagem de todos os modelos possíveis que podem ser representados através da linguagem). É importante ressaltar que os processos de modelagem e metamodelagem podem ser recursivos, ou seja, modelando um metamodelo ou descrevendo todos os metamodelos significa definir um metametamodelo (Kolovos et al., 2013).

A MDSE trabalha com qualquer artefato de software que deva ser expresso através de modelos, pois está relacionada com a concepção e a especificação das linguagens de modelagem: mudando o foco da codificação para a modelagem no processo de desenvolvimento de software (Fondement and Silaghi, 2004). Esses artefatos, dependendo da complexidade dos mesmos, precisam ser representados em diferentes níveis de abstração, utilizando a generalização (para representar o todo), ou a modularização (para restringir partes do problema).

Dessa forma, a MDSE define o projeto de software como uma abordagem de modelagem ao invés da visão simples dos desenho de diagramas. Os desenhos (ou descrições textuais), no processo de modelagem, possuem semânticas (significados) implícitas, as quais permitem

trocas de informações de maneira confiável. A modelagem possibilita uma série de vantagens, incluindo: validação sintática, verificação do modelo, simulação do modelo, transformações do modelo, execução do modelo (através da geração de código ou da interpretação do modelo), e a depuração do modelo. Como os modelos podem ser interpretados por computadores, ferramentas computacionais podem auxiliar na automatização dessas tarefas, como por exemplo: no refinamento de modelos (dos abstratos aos concretos), nas transformações ou na geração completa de código (Gasevic et al., 2009; Kent, 2002).

### 2.6.1 O Universo de Abordagens na MDSE

É muito comum, ao iniciarmos os estudos sobre a MDSE, nos depararmos com várias denominações referentes às diferentes abordagens estudadas na área. A Figura 2.10 abaixo nos dá um entendimento das relações existentes entre os diferentes termos usados, observe:

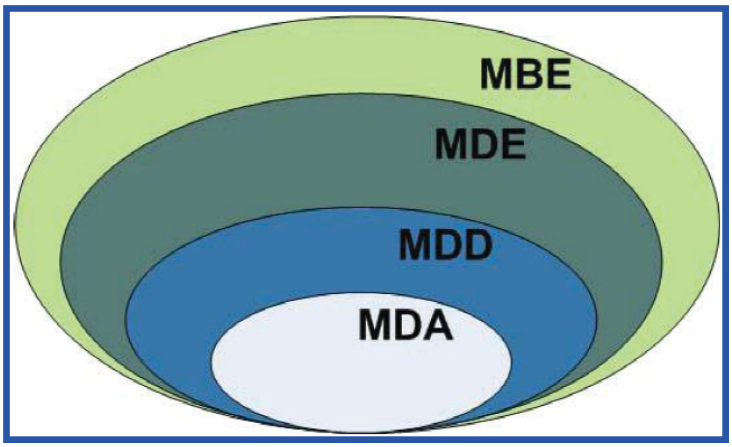



Figura 2.10: As diferentes abordagens na MDSE (Brambilla et al., 2012)

Apresentando as diferentes abordagens em uma escala hierárquica ascendente temos:

1. **Arquitetura Orientada a Modelo - *Model-Driven Architecture* (MDA):** pode ser entendida como uma visão particular do *Model-Driven Development* (MDD) proposta e padronizada pelo *Object Management Group* (OMG), por isso a MDA pode ser considerada um subconjunto do MDD, onde as linguagens de modelagem e transformação são padronizadas pela OMG;

2. **Desenvolvimento Orientado a Modelo - *Model-Driven Development* (MDD):** é considerado um paradigma de desenvolvimento que usa modelos como artefatos primários do processo de desenvolvimento de software. No MDD a implementação geralmente é semi-automatizada a partir dos modelos;

3. **Engenharia Orientada a Modelo - *Model-Driven Engineering* (MDE):** podemos considerar o MDD sendo um subconjunto da MDE porque, como o próprio **E** da MD**E** sugere, a MDE vai além das atividades de desenvolvimento, e engloba outras atividades baseadas em modelos (ex: a evolução do sistema baseada em modelo, ou a engenharia reversa orientada a modelo de um sistema em produção) (Brambilla et al., 2012);

4. **Engenharia Baseada em Modelo ou Desenvolvimento Baseado em Modelo - *Model-Based Engineering* (MBE):** nos referimos à MBE como uma versão *light* da MDE. Ou seja, os processos da MBE são processos nos quais modelos de software desempenham um papel importante, sendo usados mais como *blueprints* ou *sketches* do sistema.

Ressalta-se que essa é apenas uma demonstração para melhor compreender as diferentes abordagens, sabemos que na literatura existe uma enorme variação de todas essas siglas.

### 2.6.2 Visão Geral da Metodologia MDSE

Segundo Brambilla et al. (2012), a modelagem pode ser aplicada em diferentes níveis de abstração. Na Figura 2.11 abaixo temos um *overview* dos principais aspectos considerados na MDSE, resumindo como os diferentes aspectos envolvidos na MDSE são abordados:

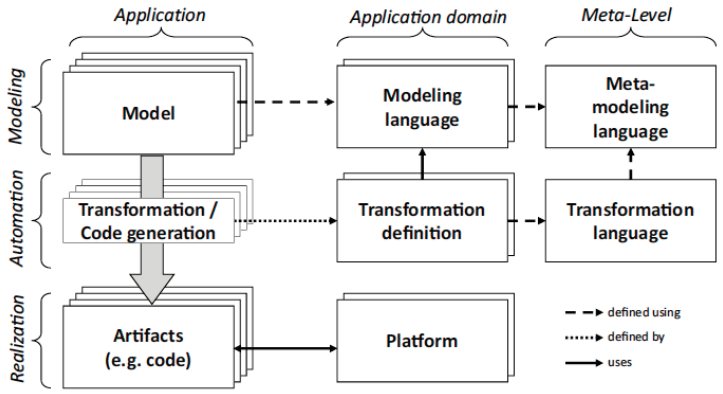



Figura 2.11: *Overview* da metodologia MDSE (Brambilla et al., 2012)

Analisando a Figura 2.11, podemos dizer que as soluções na MDSE são caracterizadas de acordo com as seguintes dimensões ortogonais:

- dimensão **conceitual**: representada pelas colunas na Figura 2.11;
- e a dimensão da **implementação**: representada pelas linhas na Figura 2.11.

A dimensão conceitual está orientada para a definição de modelos conceituais que descrevem a realidade. Essa dimensão pode ser aplicada nos seguintes níveis:

1. **nível de aplicação:** onde os modelos das aplicações são definidos, as regras de transformação são desenvolvidas, e os componentes das funcionalidades são gerados;
2. **nível do domínio da aplicação:** onde a definição da linguagem de modelagem, as transformações, e as plataformas de implementação para um domínio específico são definidas;
3. **nível meta:** onde os conceitos dos modelos e das transformações são definidos.

Na dimensão da implementação lidamos com o mapeamento dos modelos de algum sistema já existente ou de sistemas futuros. Portanto a implementação consiste na definição dos seguintes aspectos:

1. **nível de modelagem**;
2. **nível de realização:** onde as soluções são implementadas através de artefatos que estão, na verdade, em uso nos sistemas em produção;
3. **nível de automação:** onde os mapeamentos dos níveis de modelagem para os níveis de realização são realizados.

Percebe-se que o fluxo de desenvolvimento na MDSE inicia-se nos modelos de aplicação e, a partir de seguidas transformações de modelos, desce-se para a prática (realização dos níveis de execução). É importante destacar que a execução dessas transformações de modelos é definida baseada em um conjunto de regras de transformação, e essas regras são definidas através de uma linguagem de transformação específica. A construção do sistema é disponibilizada por um processo *top-down* dos modelos prescritivos que definem como o escopo está delimitado e como o sistema deve ser implementado. Por outro lado, a abstração é usada da maneira *bottom-up* auxiliando na produção dos modelos prescritivos do sistema (Baudry et al., 2010).

Toda essa dinâmica possibilita o reuso de modelos e a execução de sistemas em diferentes plataformas. Mas para que isso seja possível, os modelos devem ser especificados com base em uma linguagem de modelagem, e esta (por usa vez) deve estar de acordo com uma linguagem de metamodelagem.

## 2.7 METAMODELAGEM

Nesta seção introduzimos as técnicas da metamodelagem. Primeiro abordaremos os conceitos principais, em seguida fazemos uma análise nas linguagens de modelagem, e finalizamos com *Eclipse Modeling Project* (EMP) principal *framework* de metamodelagem para o ambiente Java.

### 2.7.1 Conceitos Principais da Metamodelagem

Segundo Schmidt (2006) a Engenharia Orientada a Modelo *Model-Driven Engineering* (MDE) é uma metodologia de desenvolvimento de software que enfatiza a criação de modelos ao invés de se concentrar nos conceitos computacionais, aumentando a produtividade através da simplificação dos processos de *design* e da comunicação entre os membros da equipe.

Como falamos anteriormente na Seção 2.6, a mais importante iniciativa da MDE é a Arquitetura Orientada a Modelo *Model-Driven Architecture* (MDA) proposta pelo *Object Management Group* (OMG)[1] e esta baseada em vários padrões sugeridos pelo OMG.

Bézivin (2005) sugeriu uma mudança de paradigma no campo da engenharia de software: considerando que na Modelagem Orientada a Objeto (*Object-Oriented Modeling* - OOM) o princípio básico era *"qualquer coisa é um objeto"*, a MDE trouxe um "novo"princípio - *"qualquer coisa é um modelo"*. As relações básicas da OOM são *instanceOf* ("instânciaDe") e *inheritsFrom* ("heraçaDe") conforme a Figura 2.12: um objeto pode ser uma instância de uma classe e uma classe pode herdar o comportamento de outras classes. A MDE aborda outras relações: uma visão particular do sistema é *representedBy* ("representadaPor") um modelo, e cada modelo está *conformsTo* ("emConformidadeCom") o seu metamodelo.

A metamodelagem oferece algumas vantagens:

- fornece definições concisas sobre os conceitos da linguagem;
- fornece um formato de intercâmbio de dados uniforme;
- permite validar a correlação entre modelos;
- a administração de modelos pode ser simplificada usando repositórios de modelos.

[1]OMG: http://www.omg.org/

Um metamodelo define uma linguagem para modelos, para que este esteja em conformidade com aquele. Da mesma forma que um metamodelo está em conformidade com o seu metametamodelo, isso leva a um modelo hierárquico que consiste em 4 (0 a 3) camadas (Figuras 2.13 e 2.14) (Schmidt, 2006; Brambilla et al., 2012).

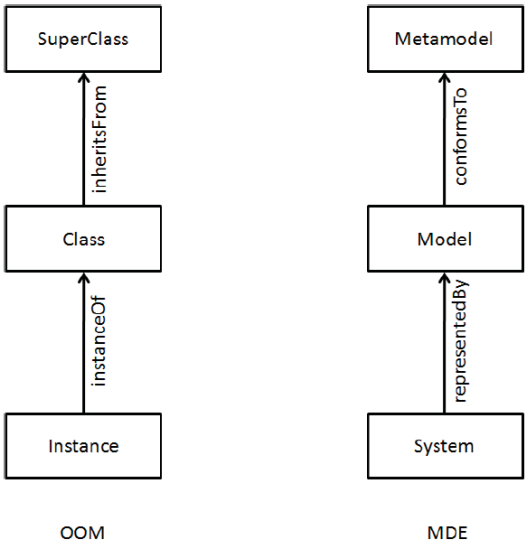



Figura 2.12: Notações básicas na OOM e na MDE (Bézivin, 2005)

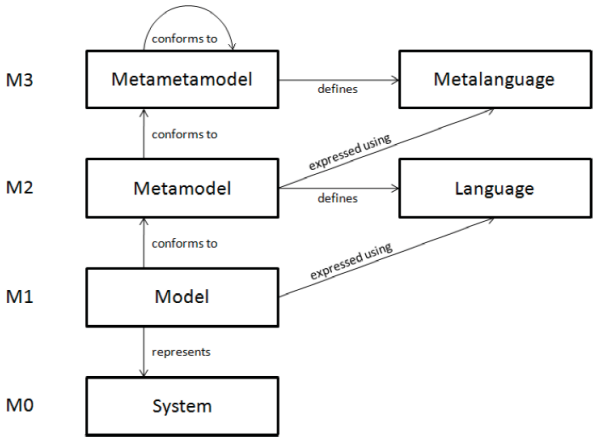



Figura 2.13: Níveis de abstração na metamodelagem (Schmidt, 2006)

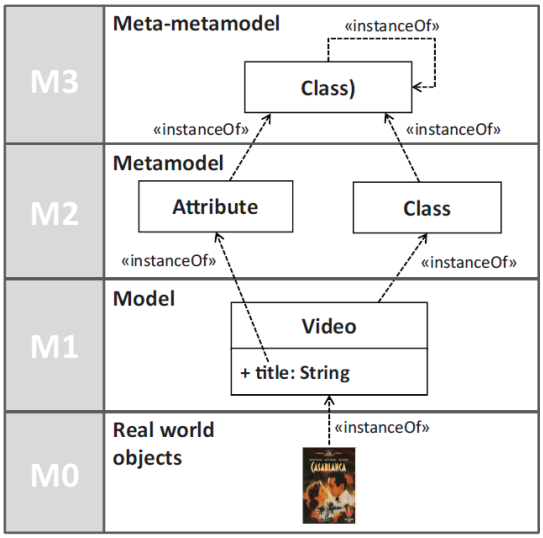



Figura 2.14: Modelos, metamodelos e metametamodelos (Brambilla et al., 2012)

Pegando a Figura 2.13 como referência, a camada mais abaixo (M0) corresponde ao sistema real. Este sistema é *representedBy* ("representadoPor") um modelo na camada acima (M1). Já o modelo está *conformsTo* ("emConformidadeCom") o seu metamodelo na camada superior (M2). E por último, este metamodelo está *conformsTo* ("emConformidadeCom") como o seu metametamodelo na camada mais acima (M3).

É importante não misturar as notações da OOM e da MDE. A Figura 2.15 abaixo destaca a diferença de foco da metamodelagem e da metaprogramação (Schmidt, 2006):

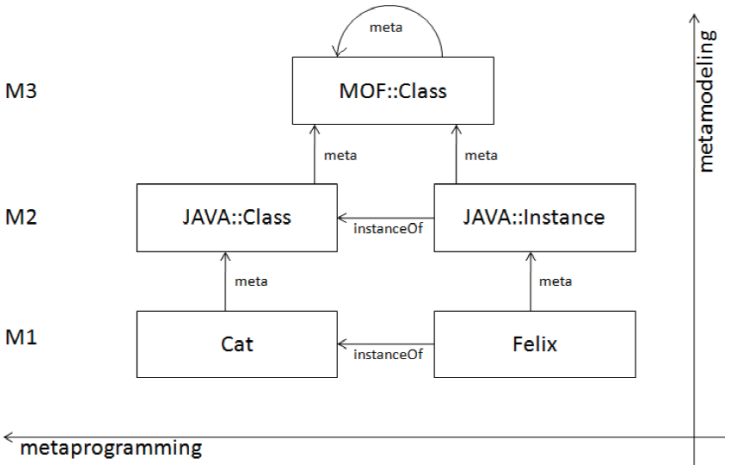



Figura 2.15: Metamodelagem x metaprogramação (Schmidt, 2006)

O eixo horizontal destaca os aspectos da metaprogramação, onde "Felix"é uma instância da classe Java "Cat"; já o eixo vertical destaca os aspectos da metamodelagem, onde "Felix"seria uma instância Java. De acordo com Atkinson and Kühne (2002), as relações também podem ser referenciadas como metamodelo "ontológico"(lidado à ontologia) (metamodelagem), e metamodelo linguístico (metaprogramação).

### 2.7.2 *Meta Object Facility* (MOF)

De acordo com Overbeek (2006), o *Meta Object Facility* (MOF) é o padrão de metametamodelagem para a MDA que foi estabelecido pelo OMG. Este padrão define conceitos

de linguagem para modelar estruturas orientadas a objetos. Estes conceitos de linguagem são reflexivos, ou seja, o próprio MOF foi descrito nele mesmo. O MOF está dividido em duas partes: o *essential* MOF (EMOF) e o *complete* MOF (CMOF).

A Figura 2.16 descreve os construtores chaves no EMOF, que são: as classes, os atributos, as operações e os parâmetros. Objetos são definidos como classes generalizadas (superclasse - *superclass*). As classes possuem propriedades intrínsecas (*Property*). Os relacionamentos entre as classes são definidos como propriedades tipo.

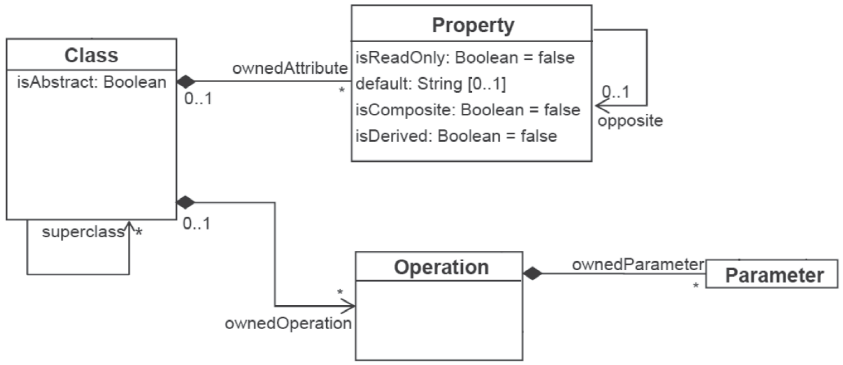



Figura 2.16: Núcleo EMOF (Overbeek, 2006)

### 2.7.3 *XML Metadata Interchange Format* (XMI)

O padrão XMI definido pelo OMG permite o armazenamento baseado em XML (*Extensible Markup Language*), e o intercâmbio de modelos. Através do XMI é possível definir um conjunto de regras para transformar metamodelos MOF em XML *Schema* ou XML DTD (*Document Type Definition*), e transformar metamodelos MOF em documentos XML (extensão padrão *.xmi) (Auth and Maur, 2002).

Com essa dinâmica de arquivos XMI é possível intercambiar modelos entre diferentes ferramentas de modelagem UML (*Unified Modeling Language*). Em complemento ao XMI, o padrão *XMI Diagram Interchange* (XMI[DI]) permite descrever informações de *layout* dos modelos.

## 2.8 LINGUAGENS DE MODELAGEM

As linguagens de modelagem permitem expressar modelos a partir de um metamodelo de origem. Devido as suas finalidades específicas, as linguagens de modelagem também são conhecidas como Linguagens de Domínio Específico (*Domain Specific Languages* - DSLs). As linguagens de modelagem podem ser textuais ou visuais (diagramáticas), porém os princípios teóricos que definem seus comportamentos são quase sempre os mesmos (Harel and Rumpe, 2000). As linguagens de modelagem visuais também são citadas como Linguagem Visual de Domínio Específico (*Domain Specific Visual Language* - DSVL).

As linguagens de modelagem textuais são formadas por cadeias de caracteres *string* (palavras, expressões, etc.) que alcançam desde linguagens naturais (como o Inglês), até linguagens altamente formalizadas (como o XML). Já as linguagens visuais consistem basicamente de elementos gráficos, mas podem utilizar elementos textuais (ex: uma classe em um diagrama de classes apresenta seu nome no topo do elemento gráfico que representa essa classe).

O importante é saber distinguir entre a sintaxe (notação) e a semântica (significado) de uma linguagem, uma vez que elas possuem diferentes propostas, estilos e formas de uso (observe a Figura 2.17):

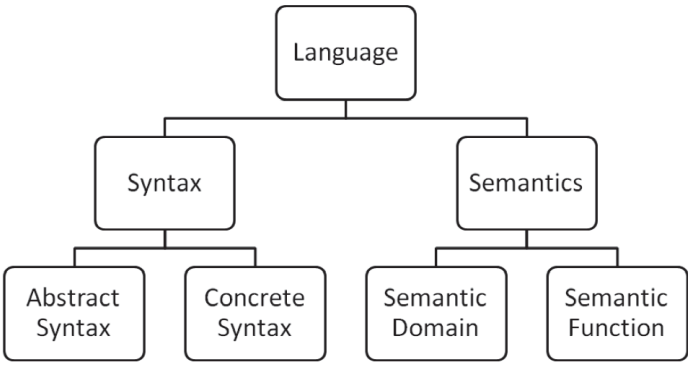



Figura 2.17: Componentes das linguagens de modelagem (Harel and Rumpe, 2000)

A sintaxe se preocupa apenas com os aspectos notacionais da linguagem, independentemente do seu significado. Uma sintaxe descritiva consiste em um conjunto de elementos (chamados expressões) que são usados na comunicação. Um linguagem textual define as palavras, e como essas palavras podem ser combinadas em uma sentença. Da mesma forma, uma linguagem visual define os elementos gráficos, e como esses elementos se relacionam entre si (Brambilla et al., 2012).

Já a semântica descreve o significado de uma notação, e pode ser dividida em duas partes:

1. **domínio semântico:** descreve o domínio de aplicação, e seus elementos descrevem as propriedades do sistema;

2. **mapeamento semântico:** relaciona cada construtor sintático a um construtor do domínio semântico.

Tudo que vemos na tela do computador ou no papel é uma representação sintática e também a representação interna do software; logo a sintaxe abstrata (*abstract syntax*) ou metamodelo nada mais é do que uma representação sintática. Analogamente, tanto a representação visual, quanto a textual são chamadas de sintaxe concreta (*abstract syntax*). Podemos dizer o mesmo para os relacionamentos entre a sintaxe abstrata e a concreta, dessa forma uma sintaxe concreta pode descrever a mesma semântica com diferentes sintaxes abstratas (veja a Figura 2.18). Por exemplo: a semântica do diagrama de classes UML é conhecida, mas existem diferentes implementações que utilizam diferentes estruturas de dados internas (Overbeek, 2006).

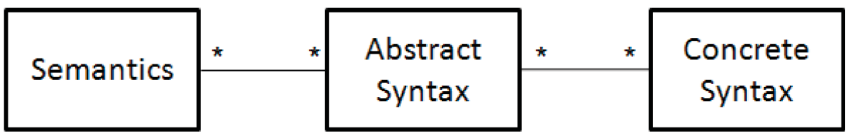



Figura 2.18: Componentes das linguagens de modelagem (Harel and Rumpe, 2000)

Na seção seguinte abordaremos um exemplo de *framework* de desenvolvimento de software (o *Eclipse Modeling Project* - EMP) que utiliza os recursos de uma linguagem de modelagem, facilitando a compreensão de suas sintaxes abstrata e concreta.

### 2.8.1 *Eclipse Modeling Project* (EMP)

O *Eclipse Modeling Project* (EMP) ficou conhecido como a tecnologia de modelagem para a linguagem de programação Java. O EMP oferece recursos de geração de código para a construção de aplicações Java baseadas em definições de modelos. Seu componente principal, o *Eclipse Modeling Framework* (EMF), combina Java, XML e UML. Através desse *framework* os desenvolvedores podem especificar modelos usando notações de classes em Java, definições de XML *schema*, ou via ferramentas UML. Com base na definição de um deles, o EMF é capaz de gerar todos os outros (Figura 2.19). Enquanto o próprio EMF é um *plug-in* Eclipse, ele também oferece interfaces simples para a construção de *views* EMF e editores para os modelos descritos. A arquitetura de *plug-in* do Eclipse usa este conceito para compartilhar dados entre vários *plug-ins* (Gronback, 2009).

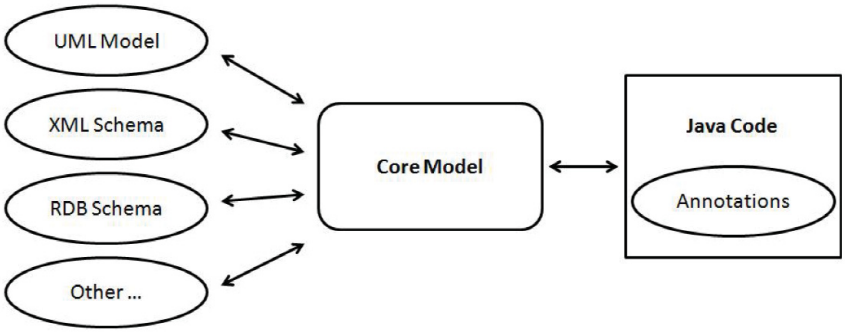



Figura 2.19: Esquema de criação de modelos via EMF (Gronback, 2009)

#### *2.8.1.1 Sintaxe Abstrata do EMF*

O Ecore, o metamodelo do EMF, está baseado no padrão OMG-EMOF. Ele oferece transformações extremamente simples para os elementos da linguagem Java e para XMI. A Figura 2.20 mostra a hierarquia de classes do Ecore: todos os elementos do Ecore herdam do `EObject` o qual é equivalente ao `java.lang.object` da linguagem Java. Para o melhor entendimento dessa hierarquia, é importante destacar que existem dois ramos: o `EClassifier` e o `ETypedElement`. O `EClassifier` reúne todos os tipos (classes, tipos de dados e enumerações). Já o `ETypedElement` abrange todos os elementos que possuem um tipo (`EClassifier`), por exemplo: atributos (`EAttribute`), referências (`EReference`), e operações (`EOperations`) (Steinberg et al., 2009).

A Figura 2.21 mostra as classes que formam o *kernel* do Ecore. As classes fazem referências aos conceitos básicos da orientação a objetos (classes, atributos, relacionamentos, herança, etc.):

- A **herança** de classes (`EClass`) é definida através de uma associação (`eSuperTypes`);
- Os **atributos** (`EAttribute`) e **referências** (`EReference`) são agregados pelas classes `EClass`;
- As **referências** (`EReference`) descrevem as cardinalidades (multiplicidades), composições e associações;
- As **auto-associações** são descritas por duas referências (`EReference`) associadas uma a outra (`eOpposite`);

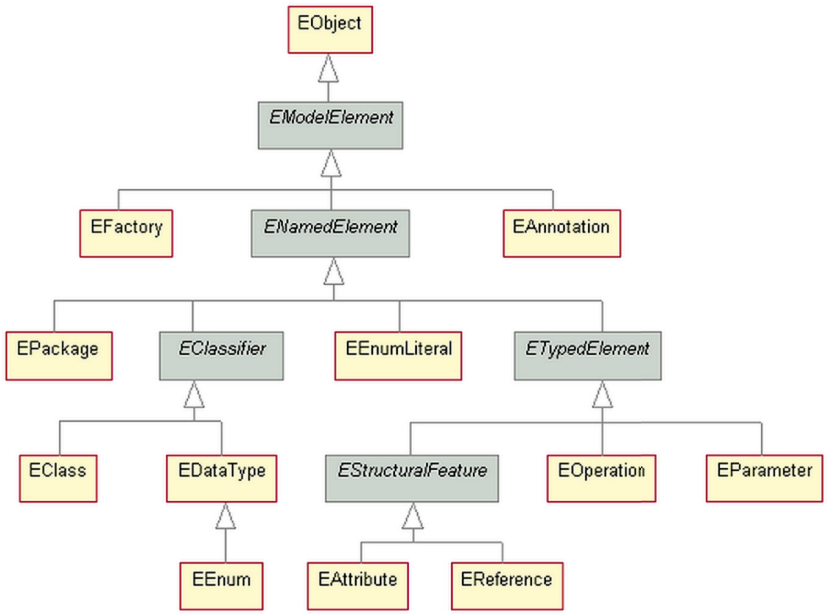



Figura 2.20: Hierarquia de classes Ecore (Steinberg et al., 2009)

- E os **tipos de dados** (*data types*) (`EDataType`) descrevem os tipos de dados base da linguagem Java.

Vale lembrar que as estruturas do Ecore do MOF são similares, e elas influenciam uma a outra no processo evolutivo de ambas.

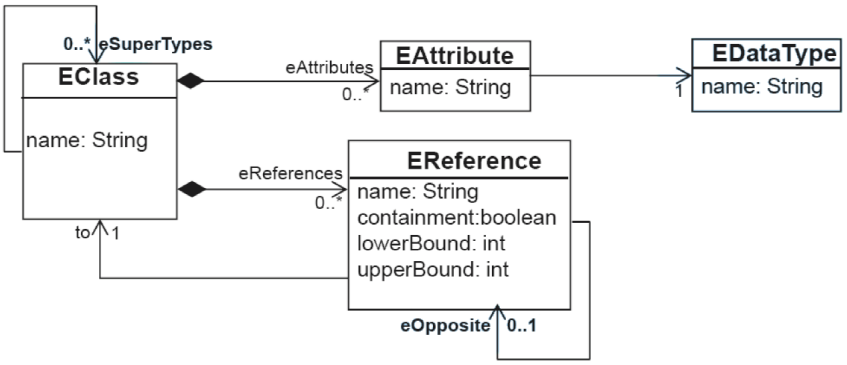



Figura 2.21: Classes núcleo do Ecore (Steinberg et al., 2009)

### *2.8.1.2 Sintaxe Concreta do EMF*

Construída sobre o Ecore, uma sintaxe abstrata baseada no EMF pode definir uma sintaxe concreta textual ou visual. O *Graphical Modeling Framework* (GMF) possibilita a definição de uma sintaxe concreta visual, e dessa forma utiliza a biblioteca de interface gráfica *Graphical Editing Framework* (GEF). Além isso, o GMF oferece uma infraestrutura e um ambiente em tempo de execução para a manipulação de diagramas. Analogamente, o *Textual Modeling Framework* (TMF) permite definir uma sintaxe concreta textual (Steinberg et al., 2009).

## 2.9 CONCLUSÃO

Neste capítulo apresentamos brevemente os principais conceitos das áreas de inteligência artificial e modelagem. No que diz respeito à inteligência artificial abordamos as redes neurais artificiais, aprendizado profundo *deep learning*, as redes neurais convolucionais e as recorrentes (LSTM). Exploramos os principais conceitos da área de modelagem, abordando os princípios da MDSE, a metamodelagem, e as linguagens de modelagem. No próximo capítulo apresentaremos alguns trabalhos relacionados.

# 3 TRABALHOS RELACIONADOS

Modelos e metamodelos criados usando abordagens baseadas em modelos possuem uma relação de conformidade restrita. Porém temos visto um aumento de formatos de dados semiestruturados ou livres de *schema*, tais como as representações orientadas a documentos, que normalmente são criadas como documentos JSON. Mesmo não sendo um *schema* de metamodelos explícito, esses documentos poderiam ser categorizados com o objetivo de descobrir os domínios os quais pertencem e, de certa forma, deixá-los em conformidade com um determinado metamodelo. Para isso, abordagens recentes sugerem tratar a extração de informações ou inter-relacionar a área da modelagem com a área da cognificação.

Nas seções seguintes veremos diversas técnicas inteligentes de inferência de conhecimento que utilizam diferentes abordagens para extração de informações úteis a partir de dados desestruturados, e veremos também como essas técnicas são aplicadas na área da Engenharia Orientada a Modelos (*Model Driven Engineering* - MDE).

## 3.1 REDE NEURAL DE MEMÓRIA CURTA E LONGA (LSTM) APLICADA À TRANSFORMAÇÃO DE MODELOS

Grace et al. (2017) afirmou em sua pesquisa que "nos próximos dez anos, a Inteligência Artificial (IA) superará os humanos em várias atividades, tais como: na tradução de línguas, na realização de trabalhos de pesquisa do colégio, ou mesmo realizando atividades de um cirurgião". E dentro deste contexto da IA, Xie (2018) introduz conceitos de Softwares Inteligentes e Cabot et al. (2017) introduz o conceito da Engenharia de Software Cognificada.

O trabalho de Burgueño (2019) propõe utilizar as vantagens da Inteligência Artificial, em particular das Redes Neurais de Memória Curta e Longa (LSTM), para automatizar o processo de inferência de uma transformação de modelo a partir de um conjunto de pares de modelos de entrada e de saída. Dessa forma, uma vez que o mapeamento da transformação foi aprendido pela rede LSTM, o sistema da rede neural estaria apto a transformar automaticamente novos modelos de entrada em seus respectivos modelos de saída, sem a necessidade de escrever nenhuma linha de código específica para realizar a transformação correspondente.

Com base nas técnicas de Aprendizado de Máquina (*Machine Learning* - ML), mais precisamente nos métodos de aprendizado supervisionado, os quais possibilitam que máquinas aprendam padrões a partir de fontes de dados existentes, e dessa forma, essas máquinas são capazes de realizar predições relacionadas a novos dados. Ou seja, com base em um determinado conjunto de dados de entrada e saída, um algoritmo de aprendizado de máquina seria capaz de aprender o mapeamento a partir de exemplos de entrada e sua respectiva saída, e dessa forma, esse algoritmo seria capaz de predizer, a partir de uma nova entrada, qual seria sua saída correspondente. Essa técnica de aprendizado já vem sendo utilizada por empresas, como a Google, para gerar traduções de idiomas usando Redes Neurais Artificiais (Wu et al., 2016).

Burgueño (2019) utiliza Redes Neurais Recorrentes (*Recurrent Neural Networks* - RNN) onde os neurônios que compõem a rede são organizados em camadas com conexões para frente (*forward connections*), ou seja, eles estão conectados aos neurônios da próxima camada, bem como estão organizados em camadas com conexões propagadas para atrás (*back propagation connections*), isto é, os neurônios também estão conectados a outros neurônios da mesma camada ou de camadas anteriores. Esse mecanismo de propagação regressa faz com que as saídas dos neurônios em uma determinada camada sejam um *feedback* para a própria rede neural,

fazendo com que a rede neural guarde algumas informações dos passos anteriores relacionadas ao processo de aprendizado. Esses tipos de neurônios são chamados de células de memória (*memory cells*) e ajudam a rede neural a entender o seu contexto. Por exemplo: no problema das Transformações de Modelos, as células de memória ajudam a lembrar os nomes das variáveis que foram previamente declarados.

As Redes Neurais de Memória Curta e Longa (LSTM) são tipos específicos de Redes Neurais Recorrentes (RNN) que possuem uma "memória" longa e são capazes de lembrar seu contexto através de diferentes entradas (Hochreiter and Schmidhuber, 1997b). E para o problema das Transformações de Modelos, as redes neurais LSTM mostram-se propícias, pois neste cenário precisamos utilizar esta "memória" longa para lembrar mapeamentos prévios como parte de um padrão de mapeamento mais complexo. A Figura 3.1 mostra a arquitetura de Transformações de Modelos usando Redes Neurais LSTM proposta por Burgueño (2019).

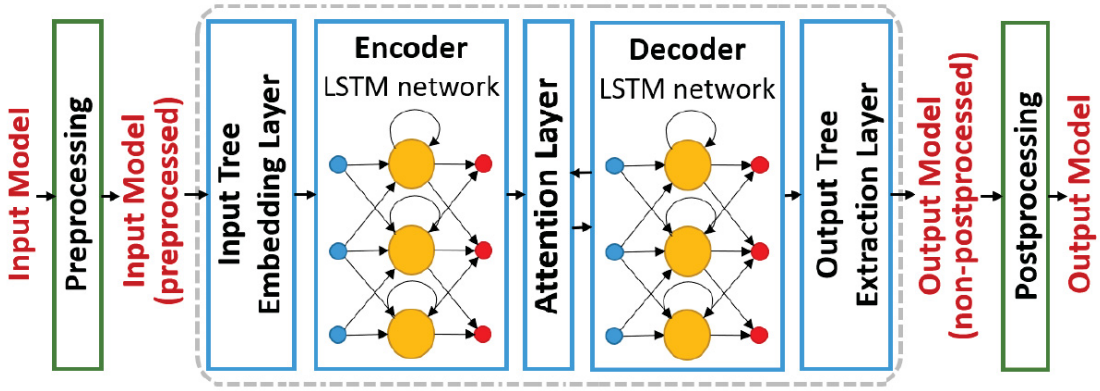



Figura 3.1: Arquitetura de Transformação de Modelos usando Redes Neurais LSTM (Burgueño, 2019)

## 3.2 CLUSTERIZAÇÃO AUTOMATIZADA DE REPOSITÓRIOS DE METAMODELOS

Ao longo dos últimos anos vários repositórios de metamodelos surgiram em resposta às necessidades da comunidade da Engenharia Orientada a Modelos (*Model-Driven Engineering* - MDE) por sistemas avançados de apoio ao reuso de artefatos de modelagem. Os profissionais da área de modelagem costumam interagir com os repositórios da MDE com diferentes propósitos, desde a mera navegação pelos repositórios, até a busca específica por artefatos que satisfazem um requisito de forma precisa. A navegação e a organização, facilidades fornecidas pelos repositórios atuais, são limitadas, uma vez que os repositórios não possuem uma visão geral estruturada dos artefatos que eles possuem, e os mecanismos de categorização (quando possuem) são dependentes de atividades manuais.

Fazer a gestão de um grande número de artefatos de modelagem é um trabalho que exige grande esforço no gerenciamento e no reuso de artefatos armazenados em repositórios de modelos. Afinal localizar informações relevantes em um repositório extenso é uma tarefa complicada, pois exige especialistas em domínio para analisar manualmente todos os metamodelos do repositório com metadados precisos, atividades estas que podem ser demoradas e propensas a erros e omissões (Bislimovska et al., 2014).

A clusterização de software é uma área de pesquisa que possui inúmeras aplicações nos problemas de engenharia reversa e manutenção de software. Ela consiste basicamente em promover a categorização automatizada dos artefatos de software (tais como: classes, funções ou arquivos) dentro de estruturas de alto nível baseadas em suas similaridades (Beck and Diehl,

2013). No contexto da Mineração de Dados (*Data Mining*), Classificação e Clusterização são operações importantes utilizadas para obter conhecimento detalhado sobre dados disponíveis, e para identificar padrões repetitivos. A Classificação é uma técnica de aprendizado supervisionado que depende da existência de classes de objetos predefinidas com o objetivo de entender a qual classe um novo item pertence (Kotsiantis, 2007). Já a Clusterização é uma técnica de aprendizado não supervisionado que busca agrupar um conjunto de objetos em diferentes *clusters* respeitando alguma função de similaridade (Jain et al., 1999).

Pensando nisso, o trabalho realizado por Basciani et al. (2016) propõe a aplicação de técnicas de clusterização para organizar automaticamente metamodelos armazenados, e com isso fornecer aos usuários uma visão geral dos domínios de aplicação cobertos pelos metamodelos disponíveis, onde metamodelos mutuamente similares são agrupados com base em uma medida de proximidade cuja definição pode se dar de acordo com uma pesquisa específica e requisitos de navegação. Para isso, foi utilizado um algoritmo de clusterização hierárquica aglomerativa, muito usado na área de clusterização de dados (*Data Clustering*) (Jain et al., 1999), este algoritmo detecta similaridades nos metamodelos de acordo com determinadas métricas de similaridades. Os experimentos de validação do trabalho de Basciani et al. (2016) demonstraram que as similaridades léxicas e estruturais podem ajudar a classificar metamodelos de acordo com o domínio de aplicação que eles formalizam. Essa abordagem foi implementada na forma de extensão para a plataforma MDEForge (Figura 3.2).

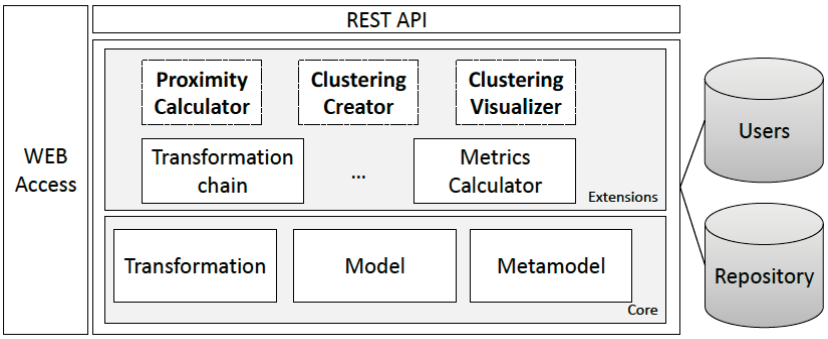



Figura 3.2: Arquitetura da Plataforma MDEForge (Basciani et al., 2016)

## 3.3 TÉCNICAS DE APRENDIZADO DE MÁQUINA (*MACHINE LEARNING - ML*) APLICADAS À CLASSIFICAÇÃO AUTOMÁTICA DE METAMODELOS

Dando continuidade aos avanços das possíveis soluções relacionadas à classificação automatizada de metamodelos, alguns dos pesquisadores que participaram do trabalho de Basciani et al. (2016) recentemente propuseram aplicar técnicas de *Machine Learning* para classificar automaticamente metamodelos presentes em repositórios de metamodelos (Nguyen et al., 2019). Os algoritmos de ML tentam simular as atividades do aprendizado humano com o objetivo de adquirir conhecimento sobre o mundo real de maneira autônoma (Portugal et al., 2015). Dessa forma, os sistemas de ML são capazes de realizar generalizações a partir de exemplos concretos, sem a necessidade de serem programados manualmente (Wuest et al., 2016). Com base nessas características dos recursos de ML, Nguyen et al. (2019) criaram a ferramenta AURORA (um Classificador Automatizado de Repositórios de Metamodelos usando Redes Neurais) que implementa uma rede neural *feed-forward* capaz de classificar metamodelos a partir de um conjunto de treinamento com metamodelos identificados. E após este processo de treinamento, o classificador AURORA é capaz de classificar metamodelos não identificados.

Para atestar a eficácia do classificador AURORA, foram realizadas validações empíricas da ferramenta com um conjunto de 555 metamodelos usando 10 diretórios de validações cruzadas que demonstraram a capacidade do classificador em predizer com alta precisão a identificação de um metamodelo aleatório.

A arquitetura da ferramenta AURORA está dividida em duas fases: construção (*building*) e desenvolvimento (*deployment*), conforme demonstra a Figura 3.3.

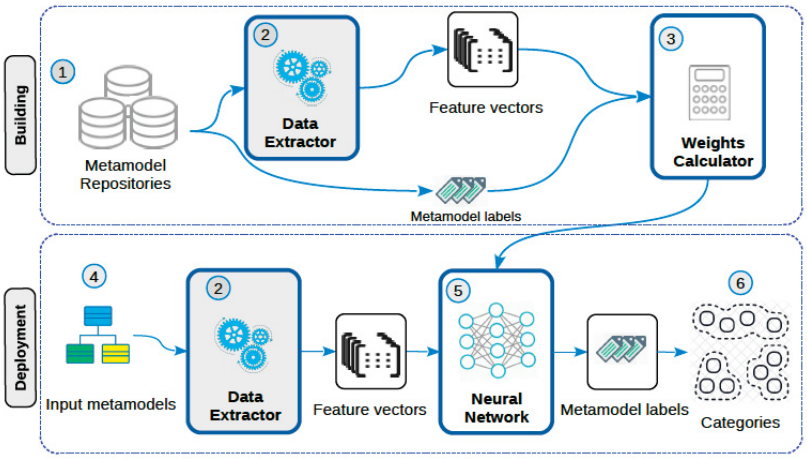



Figura 3.3: Arquitetura da Ferramenta AURORA (Nguyen et al., 2019)

O primeiro passo na fase de construção **(1)** é a obtenção dos metamodelos identificados a partir de um determinado repositório de metamodelos. Cada metamodelo identificado passa por um processo de extração de dados (*Data Extractor*) **(2)**, o processo de *Data Extractor* coloca o metamodelo no formato que permita que o mesmo seja serializado dentro de um vetor de características, e este vetor de características é usado no processo de treinamento da rede neural que faz parte do classificador AURORA auxiliando a calculadora de pesos (*Weights Calculator*) **(3)**. Dessa forma, os pesos e os *bias* obtidos são usados pela rede neural em **(5)** para classificar qualquer metamodelo recebido na fase de desenvolvimento (*deployment*). Quando um metamodelo não identificado é submetido ao processo de classificação pela ferramenta AURORA **(4)**, este metamodelo não identificado passa pelo processo de extração de características **(2)** onde é gerado um vetor com as características do metamodelo em questão, e este vetor servirá como entrada para que a rede neural *feed-forward* **(5)** realize a classificação do metamodelo automaticamente identificando-o, sendo possível categorizá-lo corretamente no respectivo repositório **(6)**.

**O Processo de Extração de Dados - *Data Extractor* (2)**: basicamente este processo consiste em extrair todas as características do metamodelo para um vetor, seguindo as técnicas sugeridas por Leopold and Kindermann (2002). A forma como o extrator de dados (*Data Extractor*), passo **(2)** na Figura 3.3, funciona na ferramenta AURORA é apresentada na Figura 3.4, onde o Extrator de Termos (*Terms Extractor*) **(E)** realiza o *parser* dos termos extraídos do metamodelo de entrada (*Metamodels Dataset*). Esses termos crus extraídos passam por um processo de normalização através de um Normalizador NLP (*NLP Normalizator*) **(N)** onde são realizas tarefas relacionadas ao Processamento de Linguagem Natural (*Natural Language Processing* - NLP), tais como: *(i) stemming*, *(ii) lemmatization*, e *(iii) stop words removal*.

Três esquemas de codificação (*encoding schemes*) (*uni-gram, bi-gram, n-gram*) são aplicados para representar informações de diferentes granularidades referentes aos termos crus extraídos de todas as instâncias dos metamodelos identificadas:

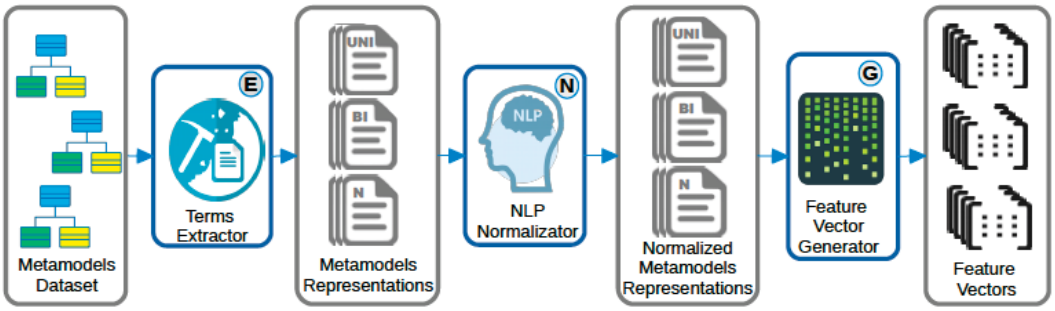



Figura 3.4: Processo de Extração de Dados na Ferramenta AURORA (Nguyen et al., 2019)

- ***uni-gram*:** corresponde a uma coleção de termos simples que não reflete nenhuma estrutura do metamodelo;
- ***bi-gram*:** representa parcialmente a estrutura do metamodelo, considerando as relações de contenção entre os elementos identificados (ex: classes vs. recursos estruturais, pacotes vs. classes, etc.);
- ***n-gram*:** representa a estrutura do metamodelo enriquecendo o *bi-gram* com propriedades de atributos (ex: informações sobre tipagem de dados, cardinalidades, multiplicidades, etc.).

Para exemplificar como funcionam os esquemas de codificação no processo de extração de dados, vamos considerar o metamodelo *Family* (Figura 3.5). A Figura 3.6 mostra o resultado sumarizado do processo de extração de dados, apresentando todos os termos léxicos do metamodelo *Family*, e a codificação correspondente tanto para os termos crus, quanto para o vetor de características normalizado.

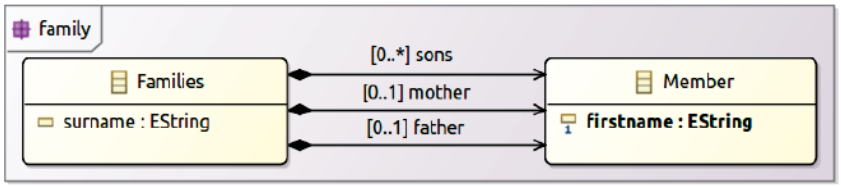



Figura 3.5: O metamodelo *Family* (Nguyen et al., 2019)

| Metamodel element | | | Raw feature vector | | | Normalized feature vector | | |
|---|---|---|---|---|---|---|---|---|
| element type | id | element name | uni-gram | bi-gram | n-gram | uni-gram | bi-gram | n-gram |
| *package* | #1 | family | family | – | – | famili | – | – |
| *metaclass* | #2 | Families | Families | family.Families | – | | famili.famili | – |
| *attribute* | #3 | surname | surname | Families.surname | Families.surname.Int.0.1 | surnam | famili.surnam | famili.surnam.Int.0.1 |
| *reference* | #4 | mother | mother | Families.mother | Families.mother.0.1.TRUE | moth | famili.moth | famili.moth.0.1.TRUE |
| | #5 | father | father | Families.father | Families.father.0.1.TRUE | fath | famili.fath | famili.fath.0.1.TRUE |
| | #6 | sons | sons | Families.sons | Famili.sons.0.*.TRUE | son | famili.son | famili.son.0.*.TRUE |
| *metaclass* | #7 | Member | Member | family.Member | – | memb | famili.memb | – |
| *attribute* | #8 | firstname | firstname | Member.firstname | Member.firstname.0.1.Int | firstnam | memb.firstnam | memb.firstnam.0.1.Int |

Figura 3.6: Extração de dados do metamodelo *Family* (Nguyen et al., 2019)

De acordo com o processo de extração ilustrado na Figura 3.4, os termos léxicos do metamodelo são extraídos e codificados seguindo o esquema descrito na Figura 3.4, onde

cada nome de elemento no metamodelo é fornecido com um identificador e com o nome do metamodelo ao qual pertence. Por exemplo, o pacote (*package*) *family* está: *(i)* individualmente codificado em *uni-grams*; *(ii)* está codificado em *bi-grams* como parte dos termos das metaclasses *#2* e *#7*; e *(iii)* está codificado em *n-grams* como parte dos termos dos recursos estruturais *#3*, *#4*, *#5*, *#6*, e *#8* conforme a tabela ilustrada na Figura 3.6. As codificações relacionadas ao processo de normalização do vetor de características requerem que os termos passem pelas etapas de *(i) stemming*, *(ii) lemmatization*, e *(iii) stop words removal*. Dessa forma, por exemplo, o termo *family* é traduzido para *famili* e compreende tanto o pacote *family* quanto a metaclasse *Families*. Além disso, informações adicionais sobre cardinalidades e sobre se o nó está sujeito a contenções são anexadas (Nguyen et al., 2019).

Com o objetivo de filtrar aqueles termos que aparecem poucas vezes em um determinado metamodelo, Nguyen et al. (2019) criaram valores de corte (*cut-off*) que definem o menor número de ocorrências de um termo em todos os metamodelos, pois termos com menor frequência de ocorrência apresentam singularidades que devem ser evitadas; e por outro lado, quando um valor de corte for muito alto isso gera problemas de precisão na classificação. Sendo assim, o vetor de características *Feature Vector Generator* **(G)** é gerado com aqueles termos que possuem uma frequência de ocorrência maior do que o valor de corte. Os vetores de características resultantes são agrupados na forma de uma matriz, e passam a ser chamados de *Term Document Matrix* (TDM), uma matriz onde cada linha representa um metamodelo dentro do repositório, e as colunas representam os termos extraídos, onde uma entrada $(i, j)$ representa a frequência do termo $j$ no metamodelo $i$.

**Avaliando o Classificador AURORA**: o processo de análise de desempenho e precisão do classificador AURORA foi conduzido através de uma validação baseada em um conjunto de dados (*dataset*) e algumas métricas de qualidade. O *dataset* foi obtido a partir de um repositório GitHub com 555 metamodelos[1], onde cada metamodelo foi inspecionado manualmente e classificado em um dos 9 diferentes grupos de metamodelos definidos previamente (cada um desses grupos possuindo um identificador único), onde para cada grupo de metamodelos são identificados:

- A quantidade de metamodelos que pertencem a esse grupo;
- A quantidade média de metaclasses presentes nesse grupo;
- A quantidade média de recursos estruturais em cada metamodelo.

Foram aplicados valores de corte para reduzir a dimensionalidade do conjunto de documentos (Shafiei et al., 2007), e dessa forma reduzir a quantidade necessária de neurônios (*perceptrons*) na camada de entrada para representar o metamodelo, levando em consideração o tipo de codificação (*uni-gram*, *bi-gram* ou *n-gram*), onde a complexidade computacional varia de acordo com a técnica de codificação escolhida.

## 3.4 REPRESENTAÇÃO DO APRENDIZADO EM GRAFOS: MÉTODOS E APLICAÇÕES

Grafo é uma estrutura de dados onipresente amplamente aplicada na ciência da computação e em outros campos relacionados, tais como: redes sociais, estruturas moleculares, redes biológicas de proteínas, sistemas de recomendações, entre outros. Todos esses domínios de

[1]A labeled Ecore metamodel dataset for domain clustering. Disponível em: https://doi.org/10.5281/zenodo.2585456, acessada em 02/03/2022

aplicação podem ser facilmente modelados como grafos, possibilitando a captura de interações entre unidades de informações individuais por meio de conexões entre arestas e vértices. Como consequência dessas facilidades, os grafos vêm se tornando a espinha dorsal de vários sistemas computacionais, permitindo que o conhecimento relacional entre entidades que interagem entre si seja eficientemente armazenado e acessado (Angles and Gutierrez, 2008). Grafos não são úteis apenas como repositórios de conhecimentos estruturados, eles também desempenham um papel importante na área de *Machine Learning*.

*Machine Learning* (ML) em grafos é uma área de pesquisa presente em diversos tipos de aplicações que variam desde projetos na indústria farmacêutica a recomendações de amizades nas redes sociais. O principal desafio nesta área é encontrar uma forma de representar ou codificar (*encoder*) informações sob a forma de estruturas em grafos que possam ser facilmente exploradas pelos modelos de *machine learning*, pois não há uma forma direta para codificar as informações que estão em uma estrutura em grafo para um vetor de características que possa ser facilmente utilizado pelos algoritmos de ML. Originariamente, as abordagens de aprendizado de máquina dependiam de heurísticas definidas pelo usuário para extrair características, codificando essas características sob a forma de informações estruturadas em grafos, por exemplo: graus estatísticos (*degree statistics*) ou funções kernel (*kernel functions*). Porém, nos últimos anos vêm surgindo abordagens que aprendem automaticamente a codificar estruturas em grafos em combinações com dimensão baixa (*low-dimensional*), utilizando técnicas baseadas em *deep learning* e redução de dimensionalidade não linear (Hamilton et al., 2017b).

A ideia por trás dessas abordagens de representação do aprendizado é aprender uma forma de mapeamento que represente vértices ou subgrafos inteiros como pontos em um espaço vetorial de dimensão baixa $\mathbb{R}^d$. O objetivo é otimizar este mapeamento para que os relacionamentos representados no espaço vetorial possam refletir a estrutura do grafo original. Após otimizar este mapeamento, o aprendizado obtido pode ser usado como recurso de entrada para tarefas de *downstream* em ML. A principal diferença entre as abordagens sobre representação de aprendizado está na forma como elas tratam o problema da representação da estrutura do grafo. Abordagens pioneiras nesse sentido lidavam com este problema como uma etapa de pré-processamento, usando estatísticas elaboradas manualmente para extrair informações estruturais. Por outro lado, as abordagens de representação do aprendizado tratam este problema como uma tarefa própria de *machine learning*, utilizando uma abordagem orientada a dados para aprender o processo que codifica a estrutura do grafo e representar essa estrutura no espaço vetorial.

**Algumas premissas importantes**: em regra, a principal entrada para um algoritmo de representação do aprendizado é um grafo não direcionado $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ associado a uma matriz de adjacência binária $\mathbf{A}$. É importante destacar que os métodos de representação do aprendizado podem fazer uso de uma matriz de valores reais dos atributos dos vértices $\mathbf{X} \in \mathbb{R}^{m \times |\mathcal{V}|}$ (ex: representando texto ou metadados associados aos vértices). O objetivo é combinar as informações contidas em $\mathbf{A}$ e $\mathbf{X}$ para mapear cada vértice ou subgrafo em um vetor $\mathbf{z} \in \mathbb{R}^d$ onde $d << |\mathcal{V}|$.

A maioria dos métodos de representação do aprendizado em grafos otimiza esse mapeamento entre a matriz de adjacência binária $\mathbf{A}$ e a matriz de valores reais dos atributos dos vértices $\mathbf{X}$ de uma maneira não supervisionada, sem a necessidade de fazer uso de tarefas de *downstream* de *machine learning*. Mas também existem abordagens de representação do aprendizado supervisionada, onde os modelos fazem uso de rótulos de classificação e de regressão para otimizar o mapeamento entre as matrizes $\mathbf{A}$ e $\mathbf{X}$. Esses rótulos de classificação podem ser associados a um vértice individualmente, ou a um subgrafo inteiro, e são os alvos de predição nas tarefas de *downstream* de *machine learning*.

A Figura 3.7 apresenta uma visão geral sobre essa abordagem envolvendo os processos de *encoder-decoder*, onde primeiramente o processo de *encoder* mapeia o vértice $v_i$ para um

vetor de baixa dimensionalidade $z_i$ baseado nas posições dos vértices no grafo, na sua estrutura de vizinhança e/ou nos seus atributos. Em seguida, o processo de *decoder* extrai as informações especificadas pelo usuário a partir do vetor $z_i$, podendo ser: informações relacionadas ao grafo de vizinhança do vértice $v_i$ (ex: a identidade dos seus vizinhos), ou um rótulo de classificação associado ao vértice $v_i$.

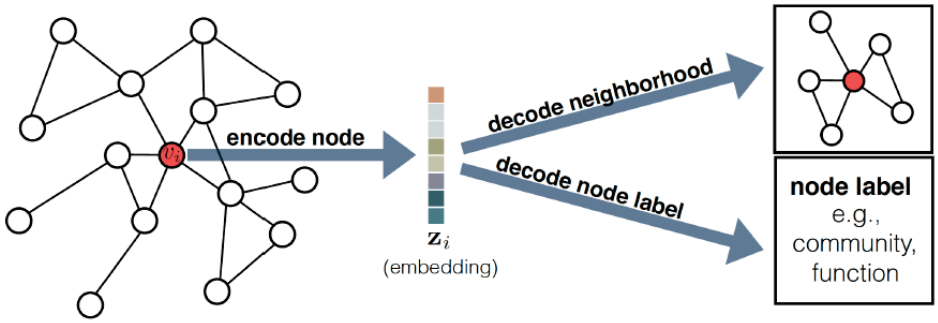



Figura 3.7: Visão Geral do Processo *encoder-decoder* (Hamilton et al., 2017b)

Basicamente o objetivo dos métodos para representar os vértices do grafo é codificar (*encoder*) os vértices em vetores de baixa dimensionalidade, os quais resumem a posição no grafo desses vértices e a estrutura de vizinhança desses vértices. O processo de representação em baixa dimensionalidade pode ser visto como um processo de codificar ou projetar os vértices dentro de um espaço latente, onde as relações geométricas nesse espaço latente correspondem às conexões (ex: arestas) no grafo original (Hoff et al., 2002). A ideia por trás dos processos de *encoder-decoder* é a seguinte: se podemos aprender a decodificar informações de grafos em alta dimensionalidade a partir de representações codificadas em baixa dimensionalidade (ex: as posições globais dos vértices no grafo, ou a estrutura de vizinhança de um vértice no grafo), então, a princípio, essas representações poderiam conter todas as informações necessárias para as tarefas de *downstream* de *machine learning*. Por exemplo, o processo de *decoder* poderia predizer a existência de arestas entre vértices a partir das representações deles (Ahmed et al., 2013; Lerique et al., 2020), ou poderia predizer a comunidade à qual pertence um determinado vértice no grafo (Hamilton et al., 2017a).

A maioria dos trabalhos utiliza a decodificação par-a-par para mapear pares de vértices representados por uma medida de similaridade com valor real que quantifica a similaridade de dois vértices no grafo original. Então quando aplicamos a decodificação par-a-par a um par de representação no vetor $(z_i, z_j)$ conseguimos uma *reconstrução* da similaridade entre os vértices $v_i$ e $v_j$ no grafo original, e a meta é otimizar os mapeamentos *encoder-decoder* para minimizar os erros ou perdas neste processo de reconstrução. Conseguindo essa otimização do sistema *encoder-decoder*, podemos usar o processo de *encoder* treinado para gerar representação dos vértices, e dessa forma usarmos isso como recurso de entrada para as tarefas de *downstream* de *machine learning*. Por exemplo, um *encoder* poderia alimentar um classificador de regressão logística com as representações dos vértices aprendidas, e com isso ter a capacidade de predizer a qual comunidade um determinado vértice pertence. Um outro exemplo seria usar as distâncias entre os vértices para recomendar amizades em uma rede social (Grover and Leskovec, 2016).

A maioria dos métodos de representação de vértices em grafos se desenvolvem seguindo basicamente quatro componentes metodológicos, e a principal diferença metodológica entre as várias abordagens de representação de vértices está em como elas definem justamente esses quatro componentes a seguir:

- **Função de similaridades par-a-par ($\mathcal{SG}$):** definida sobre o grafo $\mathcal{G}$, esta função mede a similaridade entre vértices em $\mathcal{G}$;
- **Função *encoder* (ENC):** gera a representação dos vértices, esta função contém o número de parâmetros treináveis que são otimizados durante a fase de treinamento;
- **Função *decoder* (DEC):** responsável por reconstruir os valores de similaridades par-a-par a partir das representações geradas. Normalmente esta função não contém parâmetros treináveis;
- **Função *loss* ($\ell$):** responsável por determinar como a qualidade das reconstruções par-a-par será avaliada para treinar o modelo, ou seja, como DEC($z_i$, $z_j$) será comparada aos valores reais em $\mathcal{SG}(v_i, v_j)$.

### 3.4.1 node2vec: Aprendizado Escalável de Características em Redes de Conectividade

Em virtude das dificuldades das abordagens de representação do aprendizado em capturar a diversidade dos padrões de conectividade nas redes, Grover and Leskovec (2016) propuseram *node2vec*, um *framework* algorítmico para aprender de forma contínua a representar características dos vértices (nodos) de um grafo. Este *framework* realiza o mapeamento dos vértices em um espaço vetorial com características de dimensionalidade baixa capaz de maximizar a probabilidade de preservar as vizinhanças dos vértices do grafo através da definição de uma noção flexível para essas vizinhanças, e um procedimento de caminhada aleatória tendenciosa (*biased random walk*) capaz de explorar de maneira eficiente diversas vizinhanças, e assim obter melhores representações do aprendizado.

Nas tarefas de predição de arestas (conexões entre vértices - *link prediction*), *node2vec* tenta predizer se um par de vértices em uma rede deveria ter uma aresta conectando esse par de vértices (Liben-Nowell and Kleinberg, 2007). Nos problemas de predições em redes, qualquer algoritmo de *machine learning* supervisionado necessita de um conjunto de características informativas, discriminadas e independentes. Isso significa que temos o desafio de construir uma representação vetorial com as características dos vértices e das arestas. Em virtude disso, sugiram abordagens alternativas para aprender a representar características de maneira otimizada, conforme vimos na seção anterior. O principal desafio em representar o aprendizado das características concentra-se em definir uma função objetiva capaz de obter equilíbrio entre a eficiência computacional e a precisão na predição, independente da tarefa de *downstream* de predição, de tal maneira que a representação possa ser aprendida de uma forma puramente não-supervisionada (autônoma). Para compreender melhor este desafio, observe a Figura 3.8 abaixo:

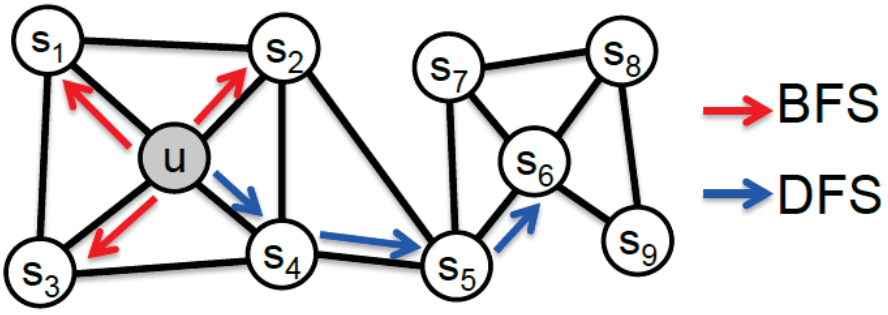



Figura 3.8: Estratégias de pesquisa nos vértices (Grover and Leskovec, 2016)

Na Figura 3.8 temos os vértices $u$ e $s_1$ que estão muito próximos e pertencem a mesma comunidade de vértices, enquanto que os vértices $u$ e $s_6$ estão em comunidades distintas, mas compartilham a mesma estrutura através do papel desempenhado por vértices *hub* ($s_4$ e $s_5$), e essa dinâmica é facilmente observada nas redes do mundo real. Por isso é fundamental para um algoritmo flexível que ele seja capaz de aprender a representar os vértices obedecendo a dois princípios: capacidade de aprender representações que agrupam vértices que estejam muito próximos na mesma vizinhança, bem como aprender representações onde os vértices que compartilham papéis similares sejam agrupados conjuntamente. Dessa forma, obteríamos um algoritmo de representação do aprendizado capaz de realizar tarefas de predição para uma variedade de domínios.

*Node2vec* é um algoritmo semi-supervisionado capaz de aprender a representar características de maneira escalável. Inspirado nas técnicas de representação de palavras no espaço vetorial (Zahran et al., 2015), *node2vec* consegue representar o aprendizado sobre os vértices através dos papéis que eles desempenham no grafo, ou mesmo nas comunidades as quais pertencem. Isso é possível graças às caminhadas aleatórias tendenciosas (*biased random walk*) que exploram diversas vizinhanças de um determinado vértice. É importante destacar a flexibilidade do algoritmo que permite ter o controle sobre o espaço de pesquisa através de parâmetros configuráveis. São esses parâmetros que orientam a estratégia de pesquisa com interpretações intuitivas e caminhadas tendenciosas, explorando de diferentes maneiras o grafo. As mesmas estratégias para representar características utilizadas nos vértices individualmente podem ser estendidas para representar as arestas (pares de vértices).

**Estratégias de pesquisa utilizadas**: visualizando o problema de amostragem de vizinhança de um vértice fonte como um problema de pesquisa local, a Figura 3.8 mostra um grafo onde dado um vértice fonte $u$ qualquer, precisamos gerar amostragens dos vértices que compõem sua vizinhança $N_s(u)$. Nesse sentido, é importante comparar diferentes estratégias de pesquisa que geram estas amostragens de vizinhança. Basicamente existem duas importantes estratégias que geram amostragens de vizinhança $N_s$ de $k$ vértices distintos, são elas:

- **Amostragem em Lagura - *Breadth-first Sampling (BFS)*:** nessa estratégia, a vizinhança $N_s$ fica restrita aos vértices que são vizinhos imediatos do vértice fonte $u$. Por exemplo: na Figura 3.8, para uma vizinhança de tamanho 3 ($k = 3$), os vértices da amostragem BFS seriam $s_1$, $s_2$, $s_3$;

- **Amostragem em Profundidade - *Depth-first Sampling (DFS)*:** já na amostragem em profundidade, a vizinhança é composta pelos vértices que compõem uma amostra sequencial em distâncias crescentes a partir do vértice fonte $u$. No exemplo da Figura 3.8, para uma vizinhança de tamanho 3 ($k = 3$) a partir do vértice fonte $u$ teríamos como resultado da amostragem em profundidade os vértices $s_4$, $s_5$, $s_6$.

Para compreender como essas estratégias auxiliam o processo de predição de vértices no grafo é importante destacar que a tarefa de predição de vértices geralmente oscila entre dois tipos de similaridades: homofilia e equivalência estrutural (Hoff et al., 2002). Nas hipóteses de homofilia, os vértices que estão altamente interconectados e pertencem a mesma comunidade ou *cluster* de rede devem ser representados muito próximos (ex: na Figura 3.8 os vértices $s_1$ e $u$ pertencem a mesma comunidade no grafo). Por outro lado, nas hipóteses de equivalência estrutural, aqueles vértices que possuem papéis similares na rede devem ser representados muito próximos (ex: na Figura 3.8 os vértices $u$ e $s_6$ desempenham um papel de vértice *hub* em suas respectivas comunidades). Note que ao contrário da homofilia, a equivalência estrutural não enfatiza a conectividade, nesse caso os vértices poderiam estar distantes na rede e ainda assim estarem sob as mesmas regras estruturais.

As estratégias de pesquisa BFS e DFS desempenham um papel chave na produção de representações que refletem tipos de similaridades: ou a homofilia, ou a equivalência estrutural. A estratégia BFS produz uma amostra de vizinhança com representações correspondentes à similaridade homofilia. Já a estratégia DFS pode explorar partes maiores do grafo, pois é capaz de ir além da fronteira do vértice fonte $u$, além de que esta estratégia produz amostras de vértices que refletem com mais precisão uma visão macro da vizinhança, sendo importante para inferir comunidades baseadas no tipo de similaridade equivalência estrutural. A estratégia DFS é importante não apenas para inferir quais as dependências vértice-a-vértice existentes no grafo, mas também para caracterizar a natureza exata dessas dependências. E essa não é uma tarefa fácil, uma vez que temos uma restrição no tamanho da amostra e uma vizinhança grande para explorar, tudo isso resultando em uma alta variância.

*Node2vec* foi desenvolvido levando em consideração essas limitações, projetando uma estratégia flexível de amostras de vizinhança, combinando as estratégias BFS e DFS através de um procedimento de caminhada aleatória tendenciosa que explora as vizinhanças, tanto utilizando a estratégia BFS, quanto a estratégia DFS. A Figura 3.9 ilustra o procedimento de caminhada aleatória tendenciosa no *node2vec*, onde no exemplo a caminhada realizou a transição do vértice $t$ para o vértice $v$, e está avaliando o próximo passo ao sair do vértice $v$. Já os rótulos nas arestas indicam os *bias* $\alpha$ de pesquisa.

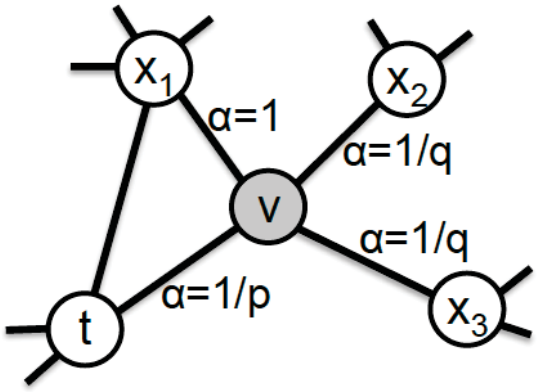



Figura 3.9: Processo de Caminhada Aleatória Tendenciosa (Grover and Leskovec, 2016)

Para melhor configurar os *bias* $\alpha$ de pesquisa, *node2vec* define uma caminhada aleatória de 2ª ordem com dois parâmetros $p$ e $q$ que são os responsáveis por guiar a caminhada aleatória. Na Figura 3.9 a caminha aleatória cruzou a aresta $(t, v)$ e estacionou no vértice $v$. A caminhada aleatória vai decidir o próximo passo avaliando as probabilidades de transição entre as arestas que partem a partir do vértice $v$. A princípio, essa probabilidade de transição é definida de forma desnormalizada baseada no caminho mais curto entre os vértices $t$ e $x_n$. Intuitivamente os parâmetros $p$ e $q$ controlam o quão rápido a caminhada aleatória explora uma vizinhança partindo de um vértice $u$ qualquer, e permitem combinar ambas as estratégias de pesquisa BFS e DFS para inferir uma afinidade por diferentes noções de equivalências entre os vértices.

**Aprendizado de características das arestas**: o algoritmo *node2vec* fornece um método semi-supervisionado de aprendizado para representar características dos vértices em um grafo. Porém, dentro desse contexto de aprendizado, é importante ter a capacidade de realizar tarefas de predição envolvendo pares de vértices (arestas) ao invés de vértices individuais. Por exemplo, na tarefa de predição de *links*, o desafio está em conseguir predizer se existe um link entre dois vértices quaisquer no grafo. Como as caminhadas aleatórias são baseadas naturalmente na estrutura de conectividade entre os vértices no grafo de adjacências, *node2vec* consegue estender

esta mesma dinâmica para pares de vértices usando uma abordagem de reprocessamento (uma espécie de *bootstrapping*) sobre a representação das características dos vértices individuais. Para isso, suponha dois vértices quaisquer $u$ e $v$, *node2vec* utiliza um operador binário sobre o vetor de características desses vértices $f(u)$ e $f(v)$ capaz de gerar uma representação $g(u, v)$ que dependendo do tipo de operador aplicado vai resultar em uma definição capaz de aumentar a probabilidade de predizer se existe ou não uma aresta entre os vértices $u$ e $v$.

## 3.5 TÉCNICAS DE CLUSTERIZAÇÃO APLICADAS À DETECÇÃO DE SIMILARIDADES

Em relação às pesquisas empíricas na engenharia de software, muitos pesquisadores concentram seus esforços na análise de sistemas de software FOSS (*Free and open-source software*), explorando características distintivas, diversidade, contexto, exclusividade e domínio de aplicação desses sistemas de software. São poucos os trabalhos que focam em buscar a identificação clara de similaridades ou diferenças entre sistemas de software.

Baseado nas suposições de que um sistema de software específico pode apresentar similaridades em determinado grau com outros sistemas de software, e que existem diferentes abordagens na tentativa de se identificar essas similaridades, o recente trabalho de Capiluppi et al. (2020) aplica técnicas de clusterização onde uma amostra de sistemas de software pode ser dividida em subconjuntos (ou *clusters*), onde cada um desses *clusters* conteria sistemas similares, exibindo as diferenças existentes com outros *clusters*. O entendimento das similaridades existentes entre diferentes projetos de software auxilia no reuso, na prototipagem, ou mesmo na escolha de implementações alternativas, ajudando desenvolvedores a acelerar o processo de desenvolvimento, e a melhorar a qualidade do software (Zhang et al., 2017).

Capiluppi et al. (2020) agruparam sistemas similares em *clusters* utilizando diferentes técnicas de clusterização:

- **Detecção de similaridades utilizando o algoritmo CrossSim:** nessa técnica, os sistemas são considerados similares ou conectados se eles possuem uma *distância* limitada (Nguyen et al., 2018);
- **Clusterização de projetos de software aplicada ao github:** técnica de clusterização utilizada em repositórios github extraídas dos trabalhos de Borges and Valente (2018);
- **Alocação Latente de Dirichlet - *Latent Dirichlet Allocation* (LDA):** técnica para extrair automaticamente as descrições de um projeto, onde os sistemas são clusterizados baseados nessas extrações.

O objetivo da pesquisa de Capiluppi et al. (2020) é identificar se as métricas orientadas a objetos (OO), que serviram de parâmetro para validar o agrupamento em um *cluster*, são sensitivas ao contexto de seus respectivos *clusters*, ou seja, se *clusters* diferentes exibem diferentes métricas OO. O conceito de similaridade é fundamental na construção de qualquer técnica de clusterização. Um entendimento comum relacionado à similaridade de software é que o grau de similaridade entre dois software distintos está relacionado ao modo como eles são iguais. Porém ainda não temos uma definição de consenso, pois, dependendo do método usado na comparação de artefatos, vários tipos de similaridades podem ser identificadas.

Um algoritmo de clusterização tenta distribuir objetos dentro de grupos de objetos semelhantes, de modo que a similaridade entre um par de objetos em um *cluster* é maior do que entre um desses objetos com qualquer outro objeto em um *cluster* diferente (Berkhin,

2006). Várias técnicas de clusterização compartilham a propriedade de que a clusterização pode ser aplicada quando é possível especificar uma medida de proximidade (ou distância) que permite avaliar se os elementos a serem clusterizados (agrupados) são mutuamente semelhantes ou diferentes. A ideia básica é que o nível de similaridade entre dois elementos quaisquer é inversamente proporcional à distância entre eles, ou seja, quanto maior o nível de similaridade, menor é a distância entre dois elementos quaisquer. Dessa forma, a definição da medida de similaridade é uma questão crucial em quase todas as técnicas de clusterização; porém isso não é uma tarefa simples, pois definir uma medida de similaridade depende de alguns fatores, tais como: o domínio de aplicação considerado, os dados disponíveis e os objetivos. A partir do momento que temos essa medida de similaridade definida, é possível produzir uma matriz de similaridade para os objetos relacionados. Suponha que existem $n$ objetos a serem clusterizados (agrupados), então uma matriz de similaridade $n \times n$ pode ser gerada contendo todos os pares de similaridades ou dissimilaridades entre os objetos considerados (Capiluppi et al., 2020). Basicamente existem dois importantes tipos de técnicas de similaridade de software: a primeira, chamada de similaridade de baixo nível (*low-level*), é calculada considerando dados de baixo nível, tais como: código fonte, *byte code*, chamada de funções, referências de API, etc.; enquanto que a segunda, chamada de similaridade de alto nível (*high-level*), é baseada nos metadados dos projetos analisados, ou seja, similaridades nos arquivos *readme*, descrições textuais, etc. A seguir apresentamos as diferentes técnicas de clusterização exploradas no trabalho de Capiluppi et al. (2020).

### 3.5.1 Análise de Similaridades Pareadas via Algoritmo CrossSim

Nguyen et al. (2018) projetaram e implementaram o CrossSim (*Cross Project Relationships for Computing Open Source Software Similarity*), um algoritmo que realiza cruzamentos de relacionamentos de projetos de software buscando identificar similaridades entre projetos de software open source (OSS - *Open Source Software*) baseadas na lógica de dados conectados (*Linked Data*). O núcleo central do método de representação *Linked Data* é um grafo RDF feito de diversos vértices e arestas (conexões) orientadas para representar relacionamentos semânticos entre vários artefatos. O algoritmo CrossSim considera a analogia das aplicações típicas dos grafos RDF, e explora os grafos para representar diferentes tipos de relacionamentos em ecossistemas de OSS. Com a adoção dessa representação em grafos, o CrossSim consegue transformar os relacionamentos existentes entre artefatos de software em um formato matemático, facilitando assim vários tipos de técnicas computacionais.

Essa técnica realiza a importação dos dados dos projetos de software existentes em repositórios OSS e realiza uma representação baseada em grafo. Dependendo do repositório considerado e das informações disponíveis para cada projeto OSS, a estrutura do grafo para ser gerada precisa ser configurada apropriadamente. No caso do GitHub, por exemplo, precisamos definir configurações específicas que permitam representar no grafo os artefatos atribuídos a cada projeto. A Figura 3.10 descreve a arquitetura do algoritmo CrossSim, onde os retângulos representam os artefatos, enquanto os círculos ovais representam as atividades que são desempenhadas automaticamente pelo CrossSim.

O módulo *Graph similarity* implementa o algoritmo de similaridade que, aplicado à representação baseada em grafo do ecossistema de entrada, gera as matrizes contendo os valores de similaridades de cada par de projetos contido no ecossistema de entrada. Observe a Figura 3.11, ela demonstra as dependências de um par de projetos OSS `project#1` e `project#2`. Para construir o grafo, o CrossSim utiliza as informações de dependências extraídas do código fonte e do metadado correspondente, representando no grafo todas as conexões (arestas) existentes entre os artefatos de software presentes no par de projetos OSS.

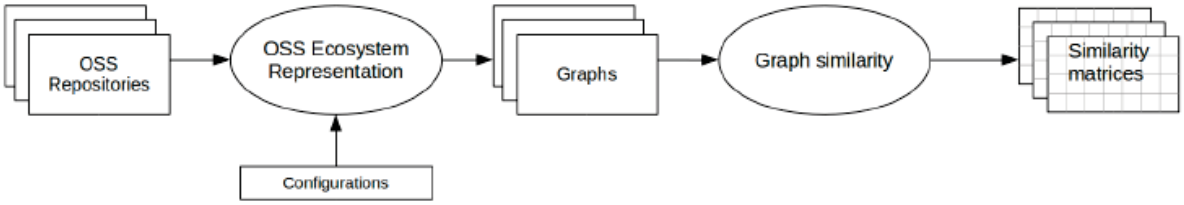



Figura 3.10: Overview da Abordagem do Algoritmo CrossSim (Nguyen et al., 2018)

Com base na estrutura do grafo, é possível explorar os vértices, as arestas (conexões), e os relacionamentos mútuos para calcular a similaridade usando algoritmos de similaridade baseados em grafos que utilizam métricas para o cálculo dessas similaridades. Por exemplo, o CrossSim adota o SimRank como mecanismo para calcular similaridade entre os vértices do grafo de um ecossistema OSS (considerando relacionamentos mútuos entre os vértices no grafo) (Jeh and Widom, 2002). Na Figura 3.11 podemos calcular as similaridades entre o `project#1` e o `project#2` levando em consideração os caminhos semânticos entre eles, por exemplo, podemos utilizar as arestas de valores semânticos `hasSourceCode` e `implements` e realizar um caminho de dois passos, ou realizar um caminho de apenas um passo através da `API#1` com a aresta `isUsedBy`. Em relação à aresta (conexão) `isUsedBy` podemos considerar os projetos similares, pois ambos têm origem na `API#1` e visam criar funcionalidades comuns usando bibliotecas comuns.

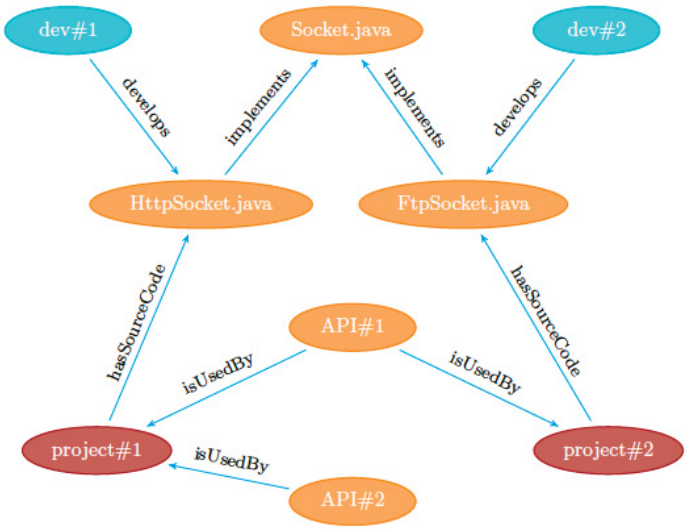



Figura 3.11: Representação Baseada em Grafo de um Ecossistema OSS (Nguyen et al., 2018)

### 3.5.2 Análise de Similaridades Baseada em Descrições de Projetos

Neste tipo de análise o processo de clusterização é baseado em coleções de sistemas de software que compartilham o mesmo domínio de aplicação. A relevância do domínio de aplicação como um fator direcionador para especificar abordagens de desenvolvimento tem sido reconhecida por deixar claro que as técnicas e as ferramentas independentes da aplicação devem ser suplementadas com uma abordagem específica de aplicação.

Borges and Valente (2018) classificaram manualmente o domínio de aplicação de 5.000 repositórios GitHub, onde cada repositório foi classificado em um dos seguintes domínios de aplicação de acordo com suas características:

- **Software aplicação:** sistemas que fornecem funcionalidades aos usuários finais, como *browsers* e editores de texto;
- **Software sistema:** sistemas que fornecem serviços e infraestrutura para outros sistemas, como os sistemas operacionais, *middleware*, e banco de dados;
- **Bibliotecas Web e *frameworks***;
- **Bibliotecas Não-Web e *frameworks***;
- **Ferramentas de software:** sistemas que apoiam as tarefas de desenvolvimento, como IDEs, gerenciadores de pacotes, e compiladores;
- **Documentação:** nesse domínio ficariam aqueles repositórios que tratam de documentação, tutoriais, exemplos de código fonte, etc.

Mesmo Borges and Valente (2018) conseguindo realizar essa classificação dos repositórios, categorizando-os manualmente em 6 diferentes domínios, realizar essa classificação de maneira automática ainda é um desafio de pesquisa.

### 3.5.3 Análise de Similaridades via Alocação Latente de Dirichlet

Esta técnica de análise de similaridades aplicada ao processo de clusterização é similiar à técnica baseada na descrição de projetos de software apresentada na seção anterior, porém acrescentando etapas automatizadas para extrair os tópicos contidos nos projetos de sistemas de software. Para isso, Capiluppi et al. (2020) extraíram o conteúdo léxico (o corpo) de cada classe Java de duas formas: *(i)* considerando o nome das classes; e *(ii)* analisando o seu código e considerando os nomes das variáveis, comentários e palavras-chave.

Para cada sistema de software, todas as classes Java são reduzidas a um corpo de termos. Todos esses termos são então considerados para criar um modelo de implementação do algoritmo Alocação Latente de Dirichlet - *Latent Dirichlet Allocation* (LDA) que utiliza técnicas de modelagem de tópicos (Koltcov et al., 2014).

## 3.6 APRENDIZADO DE MÁQUINA EM GRAFOS NATIVOS (*GRAPH-NATIVE MACHINE LEARNING*)

Algumas ferramentas de *Machine Learning* e banco de dados em grafos vêm democratizando técnicas avançadas de aprendizado de máquina (ML) baseadas em grafos, aproveitando o aprendizado profundo e as redes neurais convolucionais em grafos, permitindo alavancar os *embeddings* em grafos. Essa técnica de Aprendizado de Máquina em Grafos Nativos (*Graph-Native Machine Learning*) calcula o formato de uma rede de vizinhança para cada fragmento de dados dentro de um grafo, permitindo assim realizar tarefas de predições através de um aprendizado de máquina com mais precisão. A área da ciência de dados em grafos (*Graph Data Science*) vem mudando a forma de fazer predições em diversos cenários, desde a detecção de fraudes na tarefa de rastreamento de atividades suspeitas, até a descoberta de medicamentos com a construção de um grafo de conhecimento a respeito (Negro, 2021).

Segundo Hodler and Needham (2022), com o aprendizado de máquina em grafos nativos somos capazes de generalizar o aprendizado, predizer características a partir de dados em grafos, facilitando muito as atividades que envolvem este processo, pois uma das principais dificuldades é saber como representar dados conectados para uso em modelos de aprendizado de máquina. Através dos algoritmos de *embedding* de grafos é possível aprender a estrutura do grafo, ao invés de depender de fórmulas predeterminadas para calcular características específicas, como pontuações de centralidade, possibilitando assim que qualquer pessoa utilize o aprendizado de máquina em grafos para explorar toda a capacidade da análise preditiva.

Por exemplo: dado um grafo qualquer, o algoritmo *node2vec* (seção 3.4.1) é capaz de aprender a representar características de forma contínua (um vetor de números) para cada nó do grafo, e esse vetor de características pode ser usado para várias tarefas de aprendizado de máquina, tais como recomendações de conteúdo. E por meio deste processo aprendemos a criar a infraestrutura de dados necessária que dará suporte ao processo de treinamento e implantação de um modelo de ML (que são processos demorados) (Venuti, 2021).

### 3.6.1 Realizando *Embeddings* em Grafos

Os algoritmos de *embedding* em grafo são o ponto chave dos processos de aprendizado de máquina em grafos nativos. Esses algoritmos são usados para transformar a topologia e as características de um determinado grafo para um vetor de tamanho fixo (processo conhecido como *embedding*) que representa unicamente cada nó do grafo. Esse processo de *embedding* do grafo é poderoso porque através dele podemos preservar as características chaves do grafo enquanto reduzimos a dimensionalidade de uma forma que podemos decodificá-la. Isso significa que podemos capturar a estrutura e a complexidade do grafo e transformá-la para usá-la em diversas tarefas de predições em ML (Hodler and Needham, 2022). A Figura 3.12 demonstra o processo de *embedding* de um grafo que captura os detalhes (as características) do grafo de uma forma que pode ser usada para realizar predições ou visualizações em baixa dimensionalidade.

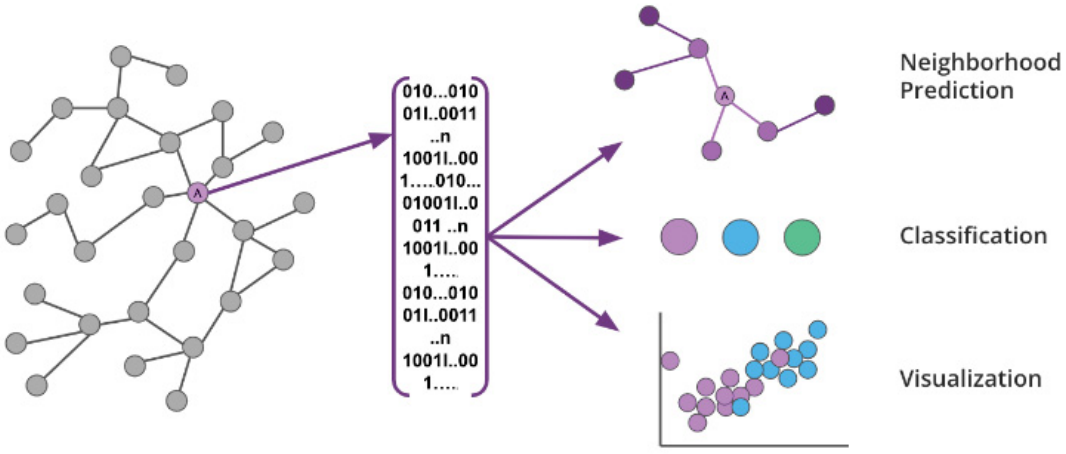



Figura 3.12: Processo de *embedding* de um grafo (Hodler and Needham, 2022)

Aprender uma métrica de similaridade adequada em um espaço vetorial de características pode determinar significativamente o desempenho dos métodos de aprendizagem de máquina (ML). Aprender tais métricas automaticamente a partir de dados é o objetivo principal da aprendizagem de similaridades. O processo de aprendizagem de Métricas/Similaridades resume-se em aprender uma função para medir a distância ou medir a similaridade entre objetos, e isso é justamente uma etapa crítica para vários problemas de *Machine Learning* (ML), tais

como classificação, clusterização (agrupamento), ranking, etc. Por exemplo, na classificação do k-vizinho mais próximo (*k-Nearest Neighbor* - kNN), uma métrica é necessária para medir a distância entre pontos de dados e identificar os vizinhos mais próximos; nos diversos algoritmos de clusterização (agrupamento), as medidas de similaridades entre os pontos de dados são usadas para determinar os *clusters*. Embora existam algumas métricas de propósito geral, como a distância Euclidiana, que pode ser usada para obter medidas de similaridade entre objetos representados como vetores, essas métricas geralmente não conseguem capturar características específicas dos dados que estão sendo estudados, especialmente em dados estruturados. Portanto, é essencial encontrar ou aprender uma métrica para medir a similaridade dos pontos de dados envolvidos em uma tarefa específica. Os *embeddings* de grafos geralmente aprendem representações de grafos em um estágio isolado e as representações aprendidas são usadas para a tarefa alvo (Ma et al., 2019).

Basicamente os trabalhos existentes sobre aprendizado de similaridades em grafos são baseados nas diferentes estratégias de aprendizagem de representação dos grafos, e como essas estratégias são utilizadas nas tarefas de aprendizado de similaridades em grafos. Podemos categorizar os modelos de aprendizagem de similaridades em grafos em três grupos: métodos baseados em *embeddings* de grafos, métodos baseados em Redes Neurais em Grafos (*Graph Neural Networks* - GNNs), e métodos baseados em kernel de grafos (*graph kernel*). A arquitetura proposta nesta tese utiliza métodos baseados em *embeddings* de grafos, e a seguir vamos ilustrar como esses métodos baseados em *embeddings* de grafos abordam o problema de aprendizagem por similaridades em grafos, e também discutiremos as funções de perda usadas nas tarefas de aprendizagem de similaridades em grafos.

Para compreender como funciona o aprendizado de similaridades em grafos (*Graph Similarity Learning*) precisamos entender algumas notações preliminares. Dada uma representação de um grafo $G = (V, E, \mathbf{A})$, onde $V$ é um conjunto de nós (nodos), $E \subseteq V \times V$ é um conjunto de arestas, e $\mathbf{A} \in \mathbb{R}^{|V \times V|}$ é uma matriz de adjacências do grafo. Esta é uma notação de propósito geral utilizada para descrever diferentes tipos de grafos, incluindo grafos balanceados e desbalanceados, grafos direcionados e não-direcionados, e grafos com atributos e sem atributos. Devemos considerar também um conjunto de grafos $\mathcal{G} = \{G_1, G_2, ..., G_n\}$ como entrada, onde a meta é medir/modelar as similaridades entre eles aos pares. Isso está relacionado ao clássico problema de isomorfismo em grafos e suas variantes. Podemos dizer que dois grafos $G = (V_G, E_G)$ e $H = (V_H, E_H)$ são isomórficos se existe uma função de mapeamento $\pi : V_G \rightarrow V_H$, tal que as arestas $(u, v) \in E_G$ se $(\pi(u), \pi(v)) \in E_H$. O problema de isomorfismo em grafos é um problema NP, e não há nenhum algoritmo eficiente para tratá-lo. Isomorfismo em subgrafos é uma generalização do problema de isomorfismo em grafos. No problema de isomorfismo em subgrafos, o objetivo é responder a partir de dois grafos de entrada $G$ e $H$ se existe um subgrafo de $G$ ($G' \subset G$) tal que $G'$ é isomórfico em relação à $H$ (ou seja, $G' \cong H$). Isso é apropriado em um cenário onde os dois grafos possuem tamanhos diferentes. Foi provado que o problema de isomorfismo em subgrafos é NP-completo (ao contrário do problema de isomorfismo em grafo). O problema do máximo subgrafo comum é outra medida menos restritiva de similaridade em grafos, no qual a similaridade entre dois grafos é definida baseada no tamanho do maior subgrafo comum entre dois grafos de entrada. Entretanto este também é um problema NP-completo (Hodler and Needham, 2022).

**Aprendizagem de Similaridades em Grafos (*Graph Similarity Learning*)** Seja $\mathcal{G}$ um conjunto de grafos $\mathcal{G} = \{G_1, G_2, ..., G_n\}$, onde $G_i = (V_i, E_i, \mathbf{A}_i)$. Sendo $\mathcal{S}$ uma função de aprendizado de similaridades, tal que $\mathcal{S} : (G_i, G_j) \rightarrow \mathbb{R}^{m \times m}$, para qualquer par de grafos $G_i$, $G_j$ $\in \mathcal{G}$, onde $\mathbb{R}^{m \times m}$ é o espaço Euclidiano $m-$dimensional. Assumimos que $s_{ij} \in \mathbb{R}^{m \times m}$ denota o score de similaridade calculado usando a função $\mathcal{S}$ entre pares de grafos $G_i$ e $G_j$. Então a função

$\mathcal{S}$ é simétrica se e somente se $s_{ij} = s_{ji}$ para qualquer par de grafos $G_i, G_j \in \mathcal{G}$. A função de aprendizado de similaridades $\mathcal{S}$ deve satisfazer a propriedade que $s_{ii} \geqslant s_{ij}$ para qualquer par dos grafos $G_i, G_j \in \mathcal{G}$. Além disso, $s_{ij}$ é mínimo se $G_i$ é o complemento de $G_j$, ou seja, $G_i = G_j$ para qualquer grafo $G_j \in \mathcal{G}$ (Ma et al., 2019).

Fica claro que isomorfismo em grafos e suas variantes (ex: isomorfismo em subgrafos, máximo subgrafos comuns, etc.) focam em medir a equivalência topológica dos grafos, dando origem a uma medida de similaridade binária que gera 1 se dois grafos são isomórficos e 0 caso contrário. Embora esses métodos possam parecer intuitivos, eles são, na verdade, mais restritivos e difíceis de calcular para grafos maiores. Nesta tese, utilizamos na arquitetura MCGML uma noção de similaridades em grafos menos complexa que pode ser calculada usando modelos de aprendizado de máquina, onde o objetivo é aprender um modelo que quantifique o grau de similaridade estrutural e o grau de parentesco entre dois grafos. Este processo é similar ao trabalho realizado na modelagem da similaridade estrutural entre nós (nodos) do mesmo grafo.

### 3.6.2 Métodos de Aprendizado de Similaridades em Grafos

A principal taxionomia encontrada na literatura sobre aprendizado de similaridades em grafos está baseada no modelo de arquitetura. Dentro desta taxionomia baseada no modelo de arquitetura, existem três categorias de métodos de aprendizado de similaridades em grafos: (1) métodos baseados em *embedding* de grafos, os quais aplicam técnicas de *embedding* para obter representações no nível de grafo (*graph-level representations*) ou no nível de nó (nodo) (*node-level representations*) além de usar as representações para o aprendizado de similaridades (Tixier et al., 2019); (2) modelos baseados em redes neurais em grafos (*graph neural network* (GNN)) que utilizam as GNNs para o aprendizado de similaridades, incluindo GNN-CNNs, GNNs Siamesas (*Siamese GNNs*) e redes de *matching* em grafos baseadas em GNN (Zhang et al., 2021); (3) métodos de baseados em *kernels* de grafos que primeiro mapeiam grafos para um novo espaço de características, onde funções *kernel* são definidas através do aprendizado de similaridades em pares de grafos, incluindo sub-estruturas baseadas em *kernels* e redes neurais baseadas em *kernel* (Du et al., 2019). Vale lembrar que métodos diferentes podem usar diferentes tipos de recursos no processo de aprendizagem.

***Embedding* de Grafos baseado em Aprendizado de Similaridades em Grafos** Segundo Shi et al. (2020), os *embeddings* de grafos têm recebido uma atenção especial nos últimos anos, e alguns modelos de *embeddings* de grafos foram apresentados, por exemplo o modelo *DeepWalk* proposto por Perozzi et al. (2014) e o modelo *node2vec* proposto por Grover and Leskovec (2016). Os métodos de aprendizado de similaridades baseados em *embeddings* de grafos utilizam representações no nível de grafos ou no nível de nós (nodos), representações essas que são aprendidas por essas técnicas de *embeddings* de grafos para definir funções de similaridades ou predizer scores (pontuações) de similaridades. Dada uma coleção de grafos, esses métodos de aprendizado de similaridades baseados em *embeddings* de grafos primeiro convertem cada grafo $G$ para um espaço $d-$dimensional ($d \ll \|V\|$), onde o grafo é representado ou como um conjunto de vetores $d-$dimensional onde cada vetor representa o *embedding* de um nó (ou seja, um *embedding* no nível de nó (nodo) (*node-level representations*)), ou como um vetor $d-$dimensional para o grafo inteiro como um *embedding* no nível de grafo (*graph-level representations*) (Grover and Leskovec, 2016). Os *embeddings* de grafos são geralmente aprendidos de uma maneira não supervisionada em estágio separado antes do estágio de aprendizado de similaridades, onde os *embeddings* de grafos obtidos são usados para estimar ou predizer o score de similaridade entre cada par de grafos.

**Métodos Baseados em *Embedding* no Nível de Nó (Nodo)** Esses métodos comparam os grafos usando representações no nível de nó aprendidas a partir dos grafos. Os scores de

similaridades obtidos por esses métodos capturam principalmente as similaridades entre nós (nodos) correspondentes em dois grafos distintos. Assim esses métodos focam nas informações no nível de nó em grafos durante o processo de aprendizado. Nesta tese utilizamos o *node2vec* - método baseados em *embedding* no nível de nó (visto na seção 3.4.1).

**Métodos Baseados em *Embedding* no Nível de Grafo** O objetivo desses métodos é aprender uma representação vetorial para cada grafo e então aprender a calcular os scores de similaridades entre grafos com base nessa representação vetorial. O *graph2vec* é um dos métodos de *embedding* no nível de grafo que aprende representações distribuídas de grafos em um processo de linguagem natural (Narayanan et al., 2017). No *graph2vec* cada grafo é visualizado como um documento e os subgrafos roteados ao redor de cada nó no grafo são visualizados como palavras que compõem o documento. Existem dois componentes principais no método *graph2vec*: o primeiro, um procedimento para extrair os subgrafos roteados ao redor de cada nó em um determinado grafo seguindo o processo de rotulagem de Weisfeiler-Lehman, o segundo, um procedimento para aprender os *embeddings* de determinados grafos por *skip-gram* com amostragem negativa. O algoritmo de rotulagem de Weisfeiler-Lehman pega o nó raiz de um determinado grafo e o grau do subgrafo pretendido como entradas, e retorna o subgrafo pretendido. Na fase de amostragem negativa, dado um grafo qualquer e um conjunto de subgrafos roteados em seu contexto, um conjunto de subgrafos escolhidos aleatoriamente são selecionados como amostras negativas e apenas os *embeddings* das amostras negativas são atualizados no processo de treinamento. Depois que os *embeddings* dos grafos são obtidos, a similaridade ou distância entre os grafos são calculadas no espaço de *embedding* para as tarefas de *downstream* de predição (exemplos: classificação de grafos, clusterização, etc.) (Narayanan et al., 2017).

**Aprendizado de Similaridades em Grafos baseado em GNN** Os métodos de aprendizado de similaridades baseados em redes neurais em grafos - *Graph Neural Networks* (GNNs) - buscam aprender as representações dos grafos através de uma GNN enquanto realizam a tarefa de aprendizagem de similaridades de ponta a ponta. A Figura 3.13 ilustra o modelo de aprendizagem de similaridades em grafos baseado nas GNNs.

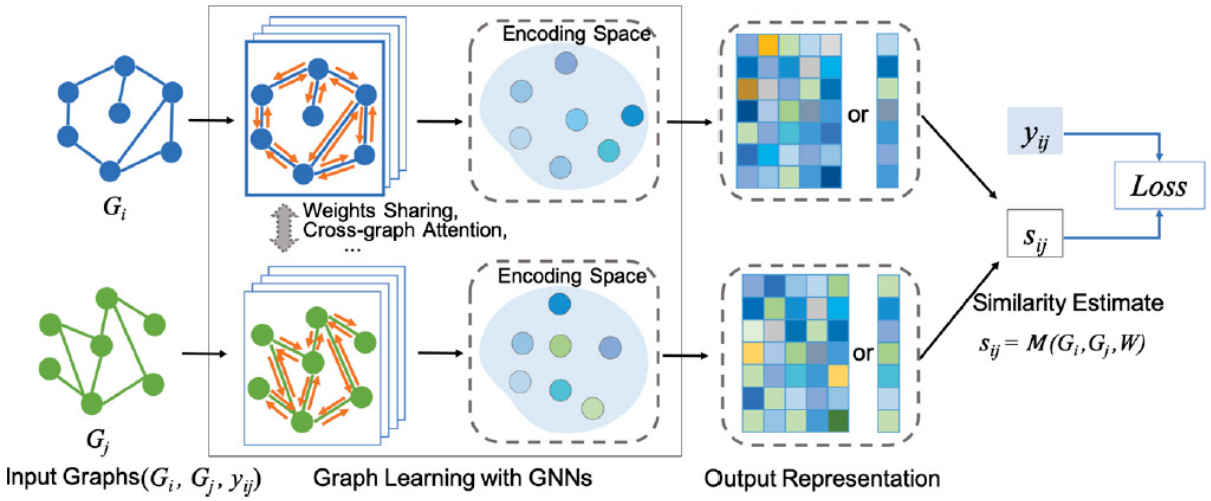



Figura 3.13: Modelo de aprendizagem de similaridades em grafos baseado nas GNNs (Ma et al., 2019)

Dado um par de grafos de entrada $<G_i, G_j, y_{ij}>$, onde $y_{ij}$ denota o rótulo de similaridade de verdade-absoluta ou score do $<G_i, G_j>$, o método de aprendizado de similaridades baseados em redes neurais em grafos primeiro aplica uma GNN multicamadas com pesos $W$ para aprender as representações de $G_i$ e $G_j$ no espaço de codificação, onde o aprendizado de cada grafo em pares poderia influenciar uns aos outros através de alguns mecanismos, tais como o compartilhamento

de pesos e interações cruzadas de grafos entre as GNNs produzidas para dois grafos. Uma matriz ou um vetor de representações será produzido para cada grafo submetido às camadas das GNN, após as quais uma camada em específico ou camadas totalmente conectadas podem ser adicionadas para produzir ou predizer os scores de similaridades entre dois grafos. E por último, as similaridades estimadas para todos os pares de grafos e os seus rótulos de verdades-absolutas são usados em uma função de perda (*loss function*) para treinar o modelo $M$ como $W$ parâmetros.

Para o desenvolvimento da arquitetura MCGML proposta, a função de similaridade é um modelo de aprendizado de máquina em grafos (*graph machine learning*) que pode ser obtido a partir de métodos baseados em *embedding* no nível de nó (nodo), como é o caso do *node2vec*. No próximo capítulo descrevemos e detalhamos a arquitetura MCGML para classificação e análises de similaridades entre metamodelos em grafos.

## 3.7 CONCLUSÃO

Neste capítulo apresentamos os principais trabalhos relacionados que serviram de base e inspirações para esta tese. A seguir entraremos da Parte II desta tese apresentando nossa proposta começando pelas abordagens para extração e classificação de modelos e análises de similaridades, em seguida demonstraremos nossos primeiros experimentos na tentativa de classificarmos modelos desestruturados em metamodelos a partir de uma rede neural *multi-layer perceptron*, culminando na apresentação da arquitetura MCGML.

# Parte II

# Tese Proposta

# 4 ABORDAGENS PARA EXTRAÇÃO E CLASSIFICAÇÃO DE MODELOS E ANÁLISES DE SIMILARIDADES

No capítulo anterior apresentamos diversas técnicas inteligentes de inferência de conhecimento que utilizam diferentes abordagens para extração de informações úteis a partir de dados desestruturados, e vimos também como essas técnicas são aplicadas na área da Engenharia Orientada a Modelos (*Model Driven Engineering* - MDE). Entretanto, ainda há uma carência de abordagens que tratam a classificação de modelos desestruturados e a possibilidade de realizar análises de similaridades entre modelos desestruturados, apesar de existirem estudos que abordam a classificação de estruturas complexas, tais como as soluções de classificação utilizando grafos (Zhang et al., 2018), mas esses trabalhos não são focados em modelos desestruturados, isso significa que ainda há um campo de pesquisa aberto a ser explorado.

Neste capítulo descrevemos todos os trabalhos realizados que culminaram com a proposta da arquitetura MCGML (*Model Classification using Graph Machine Learning*) de manipulação de metamodelos em grafos (Figura 4.1 abaixo), começando por alguns métodos desenvolvidos pelo autor desta tese juntamente com seus orientadores, Couto. et al. (2020), para analisar e classificar documentos JSON tendo como base metamodelos existentes. A seção 4.1 descreve a abordagem onde extraímos metamodelos existentes, utilizando uma solução *One-hot encoding*, para uma rede neural *Multi-Layer Perceptron* (MLP), traduzindo os elementos do metamodelo em neurônios de entrada da rede. Essa rede neural foi treinada e utilizada para classificar documentos JSON, que são traduzidos em dados de entrada para serem classificados. A partir da seção 4.2 descrevemos propriamente a arquitetura MCGML (*Model Classification using Graph Machine Learning*) de manipulação de metamodelos em grafos, capaz de realizar análises de similaridades entre diferentes metamodelos e a classificação de modelos desestruturados, utilizando como *framework driver* o *Stanford Network Analysis* - SNAP e o algoritmo *node2vec* (Grover and Leskovec, 2016).

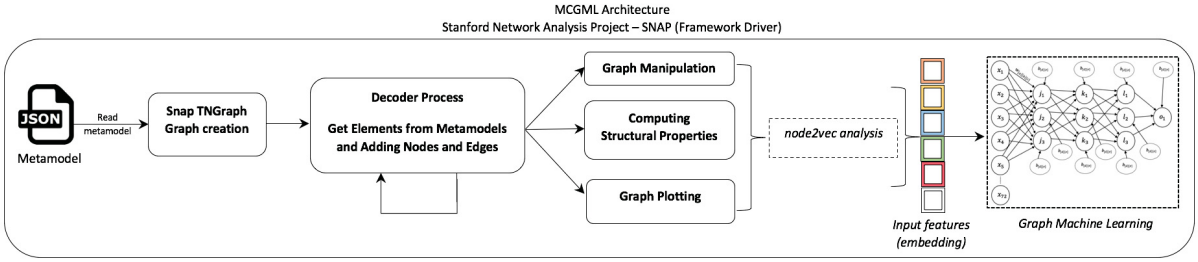



Figura 4.1: Visão geral da arquitetura MCGML de manipulação de metamodelos em grafos

## 4.1 CLASSIFICANDO MODELOS DESESTRUTURADOS EM METAMODELOS UTILIZANDO MULTI-LAYER PERCEPTRON

Modelos e metamodelos criados a partir de abordagens baseadas em modelos possuem relações de conformidade restritas, isso significa que cada elemento de um determinado modelo deve estar em conformidade com um elemento do seu metamodelo correspondente. Por exemplo, um elemento `Aluno` em um modelo pode estar em conformidade com o elemento `Class` de um metamodelo Java. Relacionamentos similares, com diferentes terminologias, estão presentes

em outros modelos de dados, tais como nas tuplas de um banco de dados, nas definições de suas tabelas/colunas (campos), nos documentos XML, e nos seus *schemas* correspondentes.

Esses relacionamentos muitas vezes são restritos e não podem ser aplicados a qualquer tipo de modelo de dados, principalmente quando estão baseados em representações semiestruturadas ou livre de *schema* (*schema-free*), onde não há relação explícita de conformidade. Tais tipos de dados são bons para o desenvolvimento rápido de aplicações, acompanhado de baixo custo por serem livremente tipados.

O formato semiestruturado mais popular são os documentos armazenados, que normalmente são representados usando arquivos no formato JSON. Os documentos JSON são utilizados na interoperabilidade, no armazenamento de dados da aplicação, onde a flexibilidade é importante, e por isso tornaram-se um padrão nas implementações de APIs RESTful. Apesar de não terem um metamodelo *schema* definido, existem iniciativas, tais como o JSON *schema* como tentativa de solução para fornecer documentos JSON tipados.

Quando *schemas* JSON não são definidos, ou seja, para documentos não-tipados, torna-se importante classificar documentos JSON, na tentativa de descobrir se esses documentos poderiam ser categorizados (tarefa de clusterização) dentro de um determinado domínio para verificar se estão parcialmente em conformidade com um metamodelo ou mesmo com um JSON *schema*. Recentes abordagens surgiram com o objetivo de extrair informações, ou tentando associar metamodelagem com a área da cognificação, ou seja, tentar extrair conhecimento a partir de modelos e metamodelos aplicando técnicas de *Machine Learning* (ML) para obter informações úteis (Cabot et al., 2017; Perini et al., 2013). O trabalho realizado por Burgueño (2019), visto na seção 3.1, apresenta uma abordagem que utiliza redes neurais de memória de termos curtos e longos (*Long Short-Term Memory Neural Networks* - LSTM) para inferir transformações de modelos de maneira automática, a partir de conjuntos de pares de modelos de entrada e saída. Entretanto, ainda há uma carência de abordagens que trate a classificação de modelos desestruturados. Já existem estudos que abordam a classificação de estruturas complexas, tais como as soluções de classificação utilizando grafos (Zhang et al., 2017), porém esses trabalhos não são focados em modelos desestruturados, isso significa que ainda há um campo de pesquisa aberto a ser explorado.

Com o objetivo de contribuir para a classificação de modelos desestruturados e explorar esse campo aberto de pesquisa, o autor desta tese (Couto. et al., 2020) apresentou uma metodologia para analisar e classificar documentos JSON tendo como base metamodelos existentes. Para isso extraímos metamodelos existentes, utilizando uma solução *One-hot encoding*, para uma rede neural *Multi-Layer Perceptron* (MLP), traduzindo os elementos do metamodelo em neurônios de entrada da rede. Essa rede neural foi treinada e então utilizada para classificar documentos JSON de entrada, os quais são também traduzidos em dados de entrada para serem classificados. Conduzimos uma série de experimentos utilizando MLPs com quantidades diferentes de camadas intermediárias, demonstrando que esta abordagem se mostra efetiva na classificação de documentos.

### 4.1.1 Classificando Documentos Armazenados em Conformidade com Metamodelos

A primeira etapa da abordagem proposta pelo autor Couto. et al. (2020) consiste em extrair metamodelos como recurso de entrada para uma rede MLP, para isso, consideramos $\mathcal{M}$ como sendo o metamodelo de entrada que define as informações estruturadas utilizadas como recursos de entrada para treinar a rede MLP. Um metamodelo é composto de classes, atributos e referências, que são traduzidos em uma coleção de pares chave/valor (*key/value*) no formato JSON. $\mathcal{E}$ descreve o conjunto de classes, atributos e referências presentes em $\mathcal{M}$, onde $\mathcal{E} \subset \mathcal{M}$.

A Figura 4.2 ilustra a abordagem proposta pelo autor Couto. et al. (2020) demonstrando seu fluxo de execução.

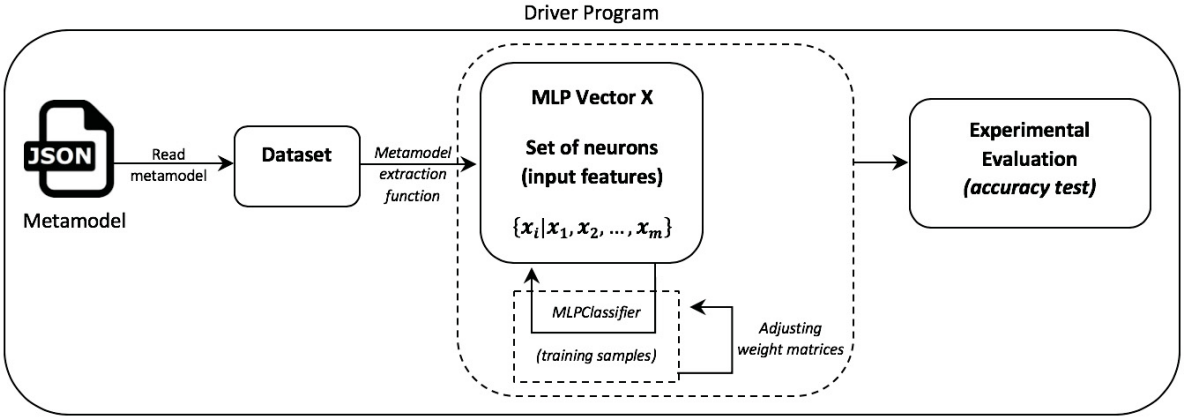



Figura 4.2: Fluxo de execução para a classificação de modelos desestruturados (Couto. et al., 2020)

O *Driver Program* implementa o fluxo de controle e carrega as operações, gerenciando passo a passo o *schema* de execução[1]. O início se dá pela leitura do metamodelo de entrada $\mathcal{M}$ o qual é atribuído a um *dataset* simples de alto nível $D_s$ onde $D_s \leftarrow \mathcal{M}$. É importante destacar que um *dataset* é uma coleção de dados que pode ser dividida em outros *datasets* (Armbrust et al., 2015b), dessa forma o *dataset* $D_s$ é processado usando uma função de extração $f_e(x)$ que seleciona as classes $(c)$, os atributos $(a)$, e as referências $(r)$, onde $\{c, a, r\} \subset \mathcal{E}$, atribuindo cada um desses elementos, seja classe, atributo ou referência, a um *dataset* específico $d_{s(0)}...d_{s(n)}$, onde $n$ representa a quantidade total de elementos. A partir do momento que essa conversão é realizada, não há diferença entre os tipos de elementos a serem codificados (*encode*) como recursos de entrada para a rede MLP. Dessa forma, cada um desses *datasets* $d_{s(0)}...d_{s(n)}$ é convertido em um número binário e empacotado para criar o vetor de características *MLP Vector X: set of input layer neurons* $\{x_i|x_1, x_2, ..., x_m\}$ na camada de entrada da rede.

O conjunto de elementos nos *datasets* de entrada são extraídos e codificados para o *MLP Vector X* aplicando técnica de *One-hot Encoding* (OHE) que é amplamente utilizada para transformar características categorizadas em características numéricas (Ul Haq et al., 2019). Em seguida a rede neural é treinada através do *MLP Classifier* utilizando um conjunto de exemplos de treinamento. Uma vez que a etapa de treinamento é concluída, é possível realizar a classificação de modelos desestruturados.

Para executar a extração do metamodelo, definimos o Algoritmo 1 que começa lendo o metamodelo de entrada $\mathcal{M}$, em seguida aplica uma função de extração $f_e(x)$ atribuindo o resultado a um *dataset* $D_s$ (Couto. et al., 2020). A variável $ItemsAmount$ guarda a quantidade de classes, atributos e referências distintas usada para calcular a quantidade de dígitos binários necessários para representar as classes, os atributos e as referências em um vetor de binários, o qual é usado como recurso de entrada na rede MLP. Para construir este vetor binário, utilizamos a técnica de OHE porque dados categorizados devem ser convertidos para numérico quando estamos lidando com problemas do tipo classificação de sequências no planejamento de uso de uma rede neural (Helmy, 2019).

Detalhando um pouco mais o Algoritmo 1, na linha 4 é criado o $MLPVectorX$ para armazenar todas as classes, atributos e referências distintas presentes em $\mathcal{M}$ como um

[1]O *Driver Program* foi desenvolvido no *Apache Spark*, uma *engine* analítica para processamento de dados (Armbrust et al., 2015a).

vetor de binários. A partir da linha 5 até a linha 7, para cada $distinct(c, a, r \subset \mathcal{E}) \in D_s$ aplicamos uma função de extração $f_e(x)$ separando o *dataset* $D_s$ em classes ($c$), atributos ($a$) ou referências ($r$), e atribuindo cada um desses elementos a outros *datasets* $d_{s(0)}...d_{s(n)}$. É importante destacar que para a rede MLP não há diferença entre classes, atributos, e referências, onde cada um desses elementos é representado por um número binário no vetor $MLPVectorX$. Na linha 9, $binaryDigitsAmount$ recebe o valor inteiro $n$ como resultado da função exponensial que tem $ItemsAmount$ como parâmetro. Dessa forma, da linha 10 até a linha 13, cada *dataset* $d_{s(0)}...d_{s(n)}$ é convertido para uma sequência em binário, tendo a quantidade de dígitos binários $binaryDigitsAmount$ sendo passada como parâmetro para a função *BinaryGenerator* responsável por gerar essa sequência em binário, em seguida a sequência em binário é colocada no vetor $MLPVectorX$ usado como vetor de características na camada de entrada da MLP. A referência entre o $d_{s(n)}.element.name$ e a sua sequência binária correspondente na posição $MLPVectorX.[n]$ é atribuída a um *dataset* especial $sD$ na linha 12, e então esse *dataset* especial $sD$ é gravado no arquivo JSON $ReferenceElementBinary$, que posteriormente é usado para auxiliar a representar modelos contidos em documentos no formato JSON como entrada para a rede MLP. Esse processo mantém um mapeamento entre os elementos do metamodelo e os seus elementos correspondentes na rede.

**Algoritmo 1** Extraindo Metamodelos para MLP:

**Input:** Metamodelo de Entrada $\mathcal{M}$
**Output:** MLP Vector X, ReferenceElementBinary

1: $D_s \leftarrow f_e(\mathcal{M})$
2: $ItemsAmount \leftarrow count(distinct(c, a, r \subset \mathcal{E}) \in D_s)$
3: $binaryDigitsAmount \leftarrow 0$
4: $MLPVectorX \leftarrow empty$

5: **for** $n = 0$ to $ItemsAmount - 1$ **do**
6: $\quad d_{s(n)} \leftarrow f_e(D_s.[n].element.name)$
7: $\quad n \leftarrow n + 1$

8: $n \leftarrow 0$
9: $binaryDigitsAmount \leftarrow toInt(exp(2^n = ItemsAmount))$
10: **for** $d_{s(0)}...d_{s(n)}$ **do**
11: $\quad MLPVectorX.[n] \leftarrow BinaryGenerator(d_{s(n)}, binaryDigitsAmount)$
12: $\quad sD \leftarrow f(d_{s(n)}.element.name, MLPVectorX.[n])$
13: $\quad n \leftarrow n + 1$

14: $SaveFile(sD, ReferenceElementBinary)$

15: **return** $MLPVectorX, ReferenceElementBinary$

Considerando um metamodelo Java simplificado[2] representado por um diagrama de classe UML e apresentado na Figura 4.3, o Algoritmo 1 converte cada classe, cada atributo, cada referência em um par chave/valor no arquivo JSON, criando assim o metamodelo de entrada $\mathcal{M}$.

[2]O metamodelo Java utilizado está disponível em:
https://www.eclipse.org/atl/atlTransformations/UML2Java/ExampleUML2Java[v00.01].pdf

Neste exemplo com o metamodelo Java, a classe *JavaElement* é associada ao *dataset* $d_{s(0)}$, o atributo *name* é associado ao *dataset* $d_{s(1)}$, e assim por diante, e então cada *dataset* do conjunto $d_{s(0)}...d_{s(n)}$ é convertido para uma sequência em binário através da função *BinaryGenerator*, e associado ao vetor $MLPVectorX$. Dessa forma, após a execução do Algoritmo 1, o vetor $MLPVectorX$ terá 20 posições, e sua estrutura pode ser vista na Tabela 4.1.

| **Estrutura do vetor *MLPVectorX* para o metamodelo Java** | | | |
|---|---|---|---|
| **Elemento** | **Tipo** | **Posição *MLPVectorX*** | **Valor** |
| JavaElement | class | 0 | 0000000 |
| name | attribute | 1 | 0000001 |
| Type | class | 2 | 0000010 |
| Modifier | class | 3 | 0000011 |
| isPublic | attribute | 4 | 0000100 |
| isStatic | attribute | 5 | 0000101 |
| isFinal | attribute | 6 | 0000110 |
| PrimitiveType | class | 7 | 0000111 |
| Method | class | 8 | 0001000 |
| isAbstract | attribute | 9 | 0001001 |
| Field | class | 10 | 0001010 |
| type | reference | 11 | 0001011 |
| parameters | reference | 12 | 0001100 |
| field | reference | 13 | 0001101 |
| owner | reference | 14 | 0001110 |
| methods | reference | 15 | 0001111 |
| JavaClass | class | 16 | 0010000 |
| classes | reference | 17 | 0010001 |
| package | reference | 18 | 0010010 |
| Package | class | 19 | 0010011 |

Tabela 4.1: Classes, atributos e referências no $MLPVectorX$ (Couto. et al., 2020)

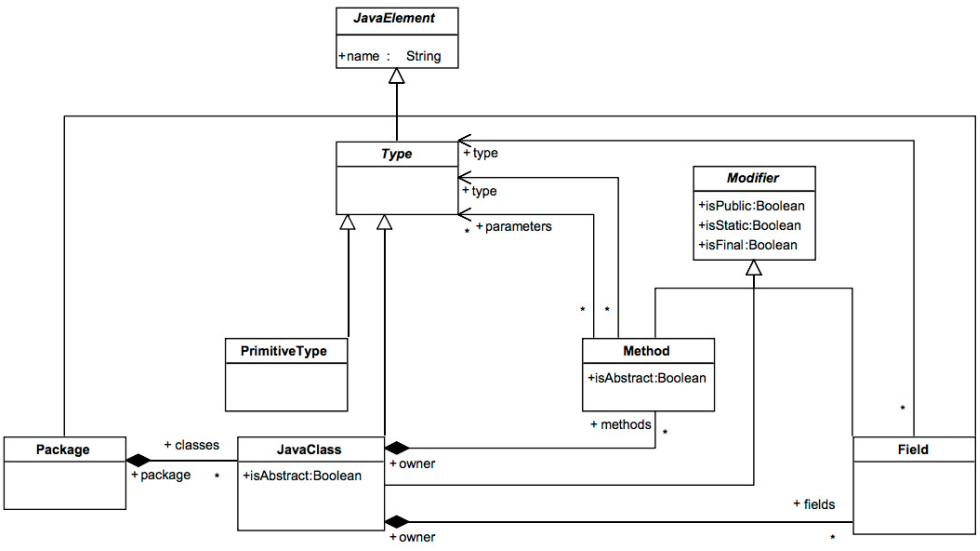



Figura 4.3: Metamodelo Java simplificado (Couto. et al., 2020)

### 4.1.2 Treinando a Rede Neural MLP

Após a extração das características de entrada (conforme descrito na seção anterior 4.1.1), é necessário treinar a rede neural MLP. Para fazer isso, precisamos escolher um conjunto de metamodelos para extrair suas características utilizando o Algoritmo 1 explicado anteriormente. É possível escolher qualquer conjunto de metamodelos. Demonstramos o treinamento da nossa rede MLP através de 4 metamodelos: *MySQL*, *KM3*, *UML* e *Java* (Couto. et al., 2020). Esses metamodelos são metamodelos criados por terceiros, disponíveis no web site ATL Transformations[3]. Para esses 4 metamodelos, o vetor de características $MLPVectorX$ foi representado com 72 posições, ou seja, foram extraídas 72 classes, atributos, e referências distintas. Dessa forma, para cada posição no $MLPVectorX$ foi atribuído uma sequência binária de 7 dígitos, onde $ItemsAmount = 72$ e $2^n = ItemsAmount$ então $n = 7$. Todas essas 72 extrações e conversões binárias podem ser encontradas no github[4]. Foi necessário escolher também a quantidade de camadas ocultas (intermediárias) na MLP, juntamente com a quantidade de neurônios em cada uma dessas camadas, e como não há uma regra exata para determinar a quantidade de camadas ocultas, e o número de neurônios em cada camada, optamos por 3 camadas ocultas (intermediárias), com 3 neurônios cada, e um neurônio na camada de saída, para o qual será atribuído um número binário de 2 dígitos, ou seja, para 4 metamodelos possíveis de saída, temos $2^n = 4$ assim $n = 2$. O *schema* da MLP gerada para este processo de treinamento é descrito na Figura 4.4, optamos por uma rede neural artificial multicamadas com treinamento via retropropagação, e inicialização aleatória convencional, onde cada neurônio em uma determinada camada, por exemplo $x_1$, conecta-se a um determinado peso $w_{[a][b][c]}$ para cada neurônio na camada seguinte, por exemplo $j_1, j_2, j_3$, onde $_{[a]}$ é o número da camada de origem, $_{[b]}$ é o número do neurônio na camada de origem, e $_{[c]}$ é o número do neurônio na camada seguinte (Couto. et al., 2020).

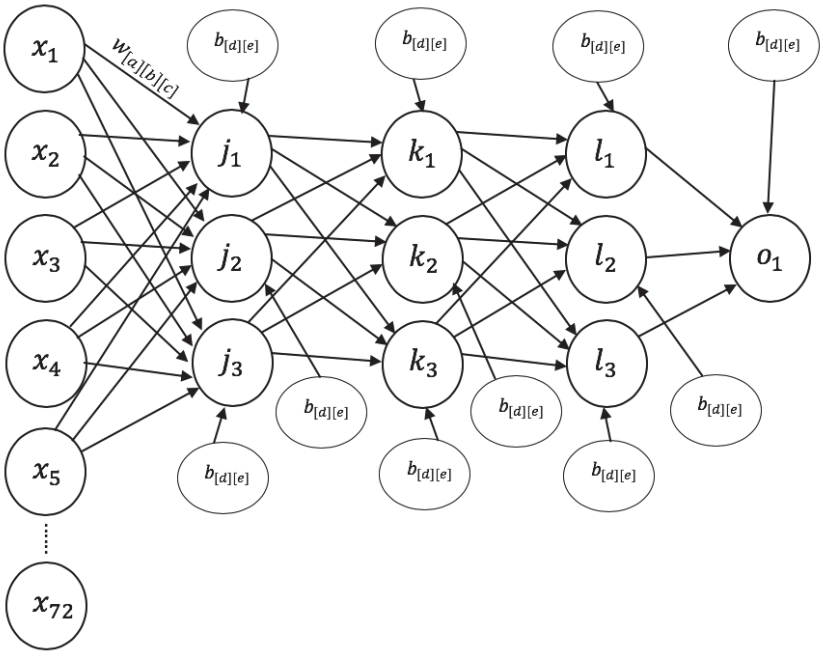



Figura 4.4: *Schema* MLP para classificar documentos (Couto. et al., 2020)

[3]https://www.eclipse.org/atl/atlTransformations/

[4]https://github.com/walmircouto/MLPTraining

Além disso, para cada neurônio nas camadas intermediárias (ocultas) é adicionado um peso *bias* $b_{[d][e]}$, onde $_{[d]}$ é o número da camada intermediária, e $_{[e]}$ é o número do neurônio na camada intermediária. Nós geramos os pesos iniciais aleatoriamente, por exemplo em um intervalo [–1, 1], para cada peso $w_{[a][b][c]}$, foram gerados 237 pesos $w_{[a][b][c]}$ e 10 pesos *bias* $b_{[d][e]}$ no total. É importante destacar que um conjunto de atualizações de todos os pesos para todos os padrões de treinamento é considerado uma época (*epoch*) de treinamento. No processo de treinamento da MLP, definimos 4000 *epoch* de treinamento. Em seguida, implementamos um segundo treinamento da MLP, com a mesma quantidade de neurônios na camada de entrada, a mesma quantidade de épocas de treinamento, e o mesmo número de camadas intermediárias, porém com 5 neurônios em cada camada intermediária, vale ressaltar que para este segundo treinamento foram gerados no total 415 pesos $w_{[a][b][c]}$ e 16 pesos *bias* $b_{[d][e]}$. Todos os pesos $w_{[a][b][c]}$ e $b_{[d][e]}$ estão disponível no github[5] (Couto. et al., 2020).

Durante os treinamentos da MLP, escolhemos a função logística *sigmoid s* como a função de ativação, com $sigmoid(a) = 1/(1 + e^{-a})$. O vetor de saída é obtido com: $o(x) = G(b^{(2)} + W^{(2)}h(x))$. Para treinar a rede é necessário aprender todos os parâmetros dos modelos, onde o conjunto de parâmetros é o conjunto $\theta = \{W^{(2)}, b^{(2)}, W^{(1)}, b^{(1)}\}$. O treinamento da rede neural é finalizado quando o erro, isto é, a diferença entre a saída desejada e a saída esperada está abaixo de algum valor limite, ou a quantidade de iterações ou épocas está acima de algum valor limite; nos treinamentos realizados, o mesmo foi interrompido quando o Erro Médio (*Mean Squared Error* - MSE) é menor que 0.01 (1%) (Couto. et al., 2020).

### 4.1.3 Representando Modelos em Documentos JSON como Entrada para a MLP

Antes de iniciar a etapa de classificação, é necessário traduzir os documentos desestruturados (em JSON) para um formato compatível com a entrada para a MLP. Para isso, utilizamos um processo similar àquele utilizado na extração de metamodelos, como descrito no Algoritmo 1 apresentado na seção 4.1.1. O processo de tradução inicia com a leitura do modelo de entrada $m$, que é formado por classes $(c)$, atributos $(a)$ e referências $(r)$, e em cima do modelo de entrada $m$ é aplicada uma função de extração (*extraction function*) $f_e(x)$ responsável por extrair todos esses elementos (classes, atributos e referências) e atribuí-los a um *dataset* $D_{sm}$ (Couto. et al., 2020). Podemos observar que na linha 3 do Algoritmo 2 é aberto o arquivo JSON $ReferenceElementBinary$, criado durante o processo de extração do metamodelo, e o mesmo é atribuído ao *dataset sd* que é usado para fazer a referência entre o *element.name* e a sua sequência binária correspondente, auxiliando na formação do arquivo texto $MLPTextFile$. Das linhas 6 a 10 no Algoritmo 2, para cada elemento distinto $distinct(c, a, r \subset D_{sm})$ é feita a localização do *element.name* contido em $D_{sm}$ no *dataset sd*, aplicando uma função de seleção na linha 7 para obter a sequência binária correspondente e associá-la ao *dataset* $d_{s(n)}$. Se $d_{s(n)} \neq null$ então temos o início da montagem do arquivo $MLPTextFile$ incluindo a sequência binária contida no *dataset* $d_{s(n)}$. O arquivo $MLPTextFile$ é utilizado como recurso de entrada na rede MLP auxiliando o processo de classificação do modelo de entrada $m$ (Couto. et al., 2020).

[5]https://github.com/walmircouto/MLPTraining

**Algoritmo 2** Representando Modelos em Documentos JSON como Entrada para a MLP:

**Input:** $m$ model, $ReferenceElementBinary$ JSON file
**Output:** MLP text file

1: $D_{sm} \leftarrow f_e(m)$
2: $ItemsAmount \leftarrow count(distinct(c, a, r \subset D_{sm}))$
3: $sd \leftarrow OpenFile(ReferenceElementBinary)$
4: $n \leftarrow 0$
5: $MLPTextFile \leftarrow empty$
6: **for** $n = 0$ to $ItemsAmount - 1$ **do**
7: $\quad d_{s(n)} \leftarrow sd.select(name, binary)\ where\ name = D_{sm}.[n].element.name$
8: $\quad$ **if** $d_{s(n)} \neq null$ **then**
9: $\quad\quad MLPTextFile \leftarrow MLPTextFile + d_{s(n)}.binary$
10: $\quad n \leftarrow n + 1$
11: **return** $MLPTextFile$

### 4.1.4 Validação Experimental da Rede MLP

Realizamos uma série de experimentos para a abordagem, onde o principal objetivo foi avaliar a precisão da rede MLP construída e treinada para classificar documentos JSON desestruturados indicando o seu metamodelo correspondente, ou seja, avaliar qual o percentual de documentos corretamente classificados (Couto. et al., 2020). Todos os experimentos foram realizados em um computador com 8 GB DDR3 RAM, processador de 2,53 GHz Intel Core i5, sistema operacional MacOS High Sierra 10.13.6, onde o framework de execução foi o Spark 2.3.3 com a linguagem Scala 2.12.8, e Java 1.8.0_191. Os experimentos apresentaram as seguintes configurações.

**Configurações da Rede**: como destacado na seção 4.1.2, definimos 2 redes MLP, ambas com 3 camadas intermediárias, onde a primeira possuía 3 neurônios em cada camada intermediária, e a segunda com 5 neurônios em cada camada intermediária. Dessa forma realizamos 2 sessões de treinamentos com 4000 épocas (*epochs*) cada. É importante ressaltar que um dos problemas que ocorre durante o treinamento da rede neural é chamado de *overfitting*. Isso acontece quando o erro em uma série de treinamentos é direcionado a um valor muito pequeno, porém ao apresentar um novo dado à rede, o valor deste erro torna-se alto. A rede memoriza os exemplos apresentados durante o treinamento, mas ela não aprende a generalizar este aprendizado para uma nova situação apresentada. Já esperávamos por isto, uma vez que a avaliação experimental conduzida não aborda o problema de *overfitting* para todos os tipos de metamodelos, pois o domínio alvo é restrito a um cenário específico (Couto. et al., 2020). É possível realizar novas sessões de treinamento com conjuntos de dados de teste de fronteira variando a quantidade de camadas intermediárias visando encontrar uma taxa de *overfitting* baixa.

As duas redes MLP criadas possuem a mesma quantidade de neurônios de entrada, 72. Esses neurônios foram extraídos a partir dos elementos contidos em 4 diferentes metamodelos: MySQL, KM3, UML, e Java. O processo de extração é executado automaticamente através de um *script* escrito na linguagem Scala. Os metamodelos de entrada utilizados são metamodelos de terceiros, disponíveis no web site *ATL transformations*[6]. Os metamodelos são primeiramente

[6]https://www.eclipse.org/atl/atlTransformations/

convertidos para um documento JSON, e em seguida são convertidos em um formato compatível com a rede, conforme demonstramos nas seções 4.1.1 e 4.1.3. O conjunto de exemplos de treinamento é composto por documentos JSON gerados automaticamente, neste caso com nomes de elementos extraídos a partir de um determinado metamodelo único, porém com uma distribuição aleatória dos elementos gerados, ou seja, os documentos JSON podem ter uma quantidade diferente de classes, atributos ou referências. Foram gerados 20 diferentes documentos para cada um dos 4 metamodelos de entrada.

**Documentos de entrada**: criamos um *script* em Scala para gerar os documentos de entrada para serem classificados. Esses documentos gerados são diferentes daqueles utilizados no conjunto de treinamento. Os documentos foram gerados respeitando dois critérios. O primeiro critério foi a quantidade de elementos - foram gerados documentos com 50 e com 100 elementos. O segundo critério diz respeito à variação do grau de conformidade dos documentos produzidos visando avaliar a precisão da rede MLP. Primeiramente foram gerados automaticamente documentos com 50 e 100 elementos, onde todos os nomes de elementos são iguais àqueles existentes nos metamodelos MySQL, KM3, UML ou Java. Isso significa que isso não é uma relação de conformidade restrita, mas apenas para gerar elementos JSON com um determinado nome. Em um primeiro caso, queríamos checar a classificação dos documentos que estavam 100% em conformidade com um metamodelo existente. Em um segundo caso, geramos elementos a partir de classes, atributos e referências misturados entre diferentes metamodelos, utilizando a seguinte proporção: 80%-20%, 60%-40%, e 50%-50%. Ou seja, criamos documentos com 80% dos elementos em conformidade com um determinado metamodelo e 20% em conformidade com um outro metamodelo. Na segunda proporção o percentual de mistura foi de 60% dos elementos em conformidade com um determinado metamodelo e 40% em conformidade com outro metamodelo. E por fim foi utilizada a proporção 50%-50% (Couto. et al., 2020). O objetivo dessas distribuições era verificar a precisão das classificações quando há uma variação na quantidade de elementos em conformidade com um determinado metamodelo. Os resultados das classificações realizadas pela rede MLP podem ser visualizados nas Tabelas 4.2 e 4.3.

| **Validação da MLP com 3 Camadas Intermediárias** | | | | |
|---|---|---|---|---|
| **Modelos com 50 elementos** | | | | |
| **%** | **MySQL** | **KM3** | **UML** | **Java** |
| 100% | 100% | 100% | 100% | 100% |
| | | | | |
| **Modelos misturados** | | | | |
| **%** | **MySQL + KM3** | | **UML + Java** | |
| 80%-20% | 96,3% | | 94,3% | |
| 60%-40% | 84,5% | | 83,2% | |
| 50%-50% | 47,2% | | 45,6% | |
| | | | | |
| **Modelos com 100 elementos** | | | | |
| **%** | **MySQL + KM3** | | **UML + Java** | |
| 80%-20% | 96,1% | | 93,9% | |
| 60%-40% | 82,7% | | 86,4% | |
| 50%-50% | 46,7% | | 45,2% | |

Tabela 4.2: Classificador MLP com 3 camadas intermediárias (Couto. et al., 2020)

| Validação da MLP com 5 Camadas Intermediárias | | | | |
|---|---|---|---|---|
| **Modelos com 50 elementos** | | | | |
| **%** | **MySQL** | **KM3** | **UML** | **Java** |
| 100% | 100% | 100% | 100% | 100% |
| | | | | |
| **Modelos misturados** | | | | |
| **%** | **MySQL + KM3** | | **UML + Java** | |
| 80%-20% | 97,2% | | 96,6% | |
| 60%-40% | 87,3% | | 85,6% | |
| 50%-50% | 48,6% | | 47,3% | |
| | | | | |
| **Modelos com 100 elementos** | | | | |
| **%** | **MySQL + KM3** | | **UML + Java** | |
| 80%-20% | 97,6% | | 95,1% | |
| 60%-40% | 83,8% | | 87,6% | |
| 50%-50% | 47,7% | | 46,8% | |

Tabela 4.3: Classificador MLP com 5 camadas intermediárias (Couto. et al., 2020)

### 4.1.5 Avaliação dos Resultados Obtidos dos Experimentos do Classificador MLP

Nesta seção discutiremos os resultados obtidos em relação à precisão do nosso classificador MLP, à técnica de codificação utilizada, e em relação à variedade de modelos de entrada.

**Precisão do classificador.** O objetivo almejado era testar a precisão e a aplicabilidade do classificador MLP na tarefa de classificar metamodelos. A partir dos resultados apresentados nas Tabelas 4.2 e 4.3, é possível verificar que quando documentos são gerados com os elementos estando 100% em conformidade com o seu respectivo metamodelo, o classificador MLP classifica corretamente todos os documentos. Isto acontece porque a rede MLP foi treinada com todos os elementos presentes no metamodelo, e neste caso, quando um documento está perfeitamente em conformidade com o seu metamodelo, o classificador MLP é preciso. Assim, para ambas as redes, com três camadas intermediárias e com cinco camadas intermediárias, a precisão é de 100%.

Quando os elementos são misturados, por exemplo: quando foram misturados 80% dos elementos do metamodelo MySQL com 20% dos elementos do metamodelo KM3, a precisão do nosso classificador MLP diminui, porém as taxas de acerto permanecem altas, isto é, ela classifica os documentos de acordo com a quantidade de elementos predominante em conformidade com um determinado metamodelo. A rede MLP com três camadas intermediárias obteve 96,3% de precisão de acerto, já a rede MLP com cinco camadas intermediárias obteve 97,2%, melhorando em 0,9%. Isto significa que, mesmo aumentando a quantidade de camadas intermediárias (com duas camadas a mais), o impacto no resultado final é pequeno quando os documentos são muito semelhantes (Couto. et al., 2020).

Quando foi feito o aumento na quantidade de elementos, passando de documentos com 50 elementos para documentos com 100 elementos, a precisão do classificador diminuiu ligeiramente, de 96,3% para 96,1% na rede MLP com três camadas intermediárias; porém a rede MLP com cinco camadas intermediárias melhorou seus resultados, passando de 97,2% para 97,6%, mostrando uma pequena melhora neste caso.

Porém quando misturamos 80% dos elementos do metamodelo UML com 20% dos elementos do metamodelo Java em um documento com 50 elementos, o classificador MLP com três camadas intermediárias teve uma taxa de precisão de 94,3%, e com cinco camadas intermediárias a taxa de precisão aumentou para 96,6%, melhorando 2,3%. O resultado ligeiramente menor em comparação com MySQL e KM3 pode ser explicado devidos aos metamodelos UML e Java possuírem alguns elementos semelhantes que podem se sobrepor. Mas o ganho com o aumento da quantidade de camadas intermediárias, de 3 para 5, é maior, evidenciando uma configuração válida quando os modelos apresentam uma estrutura mais variável (Couto. et al., 2020).

Quando foram misturados 60% dos elementos do metamodelo MySQL com 40% dos elementos do metamodelo KM3, a precisão do classificador MLP diminui, apresentando uma taxa de precisão de 84,5% para uma MLP com três camadas intermediárias, mas a taxa de precisão melhora 2,8%, passando para 87,3%, para uma MLP com cinco camadas intermediárias, o que é uma melhora importante. Isso significa que mesmo com apenas 60% dos elementos do metamodelo MySQL, nosso classificador MLP teve um bom desempenho, demonstrando que pode ser usado em soluções de reconhecimento de modelos ou documentos.

E finalmente quando misturamos 50% dos elementos do metamodelo MySQL com 50% dos elementos do metamodelo KM3, a precisão do classificador MLP ficou próxima de 50%. Este resultado já era esperado porque esta configuração poderia simular um teste aleatório, já que temos 50% dos elementos de um documento $x$ e 50% dos elementos de um documento $y$, dessa forma o classificador MLP poderia classificar ora como sendo o documento $x$, ora como sendo o documento $y$ (Couto. et al., 2020). Podemos expandir esses experimentos variando a quantidade de camadas intermediárias e a quantidade de neurônios em cada camada intermediária na tentativa de melhorar a precisão do classificador MLP.

**Codificação das características.** A decisão de implementar de maneira direta uma técnica de extração de características dos modelos e metamodelos permitiu o desenvolvimento de um extrator simples, sem a necessidade de codificar relacionamentos entre os elementos dos modelos e metamodelos. Normalmente dados no formato numérico possibilitam um melhor desempenho nos algoritmos de clusterização, regressão e classificação. Além disso, optamos por não fazer distinção se um elemento qualquer de um metamodelo é uma classe, é uma referência ou é um atributo, ou seja, apenas definimos os elementos dos modelos e metamodelos como um nome. Isso é importante porque, por exemplo, ao obtermos um metamodelo como entrada, a quantidade de características de entrada permanece gerenciável, e a solução pode ser adaptada ou reutilizada em outros cenários. É importante destacar que a abordagem *One-hot Encoding* (OHE) também apresenta alguns desafios: a dispersão dos dados transformados, onde nem sempre os valores distintos de um atributo são conhecidos com antecedência. Entretanto na solução OHE proposta, mitigamos essas restrições implementando uma extração de características de maneira direta conforme apresentado no Algoritmo 1 (Couto. et al., 2020).

Outras abordagens para classificar documentos poderiam ser usadas, por exemplo, adaptando um *schema matching* ou uma abordagem baseada em clusterização para classificação de metamodelos. No entanto, o esquema simples de codificação que propomos apresentou bons resultados. Se a quantidade de características for elevada, outros esquemas de extração precisariam ser estudados e comparados, como métodos baseados em estruturas. Pois este aspecto pode ser crítico se usarmos, além dos metamodelos, elementos de modelos ou documentos para codificarmos como características de entrada para a rede.

**Variedade de modelos.** Nós avaliamos o Classificador MLP utilizando modelos de diferentes domínios e também misturando elementos de diferentes modelos, porém outros elementos de modelos aleatórios poderiam ser usados nos testes de validação. Optamos por gerar automaticamente metamodelos de entrada com uma variação explícita nas taxas de conformidade

das características de entrada, permitindo assim analisar a solução sob limites distintos (Couto. et al., 2020). Feita esta validação inicial, como trabalhos futuros, pretendemos aplicar a abordagem em cenários do mundo real, como por exemplo, para classificar metamodelos existentes em repositórios Git. Porém não podemos afirmar se apenas esses repositórios seriam suficientes para validar as taxas de conformidade explícitas. Além disso, nesses casos, é possível realizar um pré-processamento adicional dos elementos, ou utilizá-los em conjunto com algoritmos de similaridades em grafos ou string.

Resumindo, os resultados desses experimentos mostraram que podemos utilizar redes neurais como suporte na tarefa de classificação de documentos através de um esquema de codificação simples. Podemos expandir esta abordagem sobre classificação de modelos para outros algoritmos de redes neurais, tais como os algoritmos para *Long Short-Term Memory Neural Networks* (LSTM), e também realizar comparações em relação ao nível de precisão nas classificações em diferentes tipos de redes.

## 4.2 ARQUITETURA MCGML DE MANIPULAÇÃO DE METAMODELOS EM GRAFOS

Nesta seção descreveremos a arquitetura *Model Classification using Graph Machine Learning* (MCGML) de manipulação de metamodelos em grafos, proposta como parte do trabalho de pesquisa desta tese, capaz de realizar análises de similaridades entre diferentes metamodelos e a classificação de modelos desestruturados, utilizando como *framework driver* o *Stanford Network Analysis* - SNAP, e o algoritmo *node2vec* (Grover and Leskovec, 2016) realizando os *embeddings*, ou seja, a construção de um vetor de tamanho fixo que representa unicamente cada nó do grafo. Esses espaços vetoriais de baixa dimensionalidade são utilizados como recursos de entrada (*Input features*) para as tarefas de classificação e análise de similaridades de modelos desestruturados nos algoritmos de ML em Grafos Nativos (*Graph-Native Machine Learning*). Demonstraremos através dos experimentos realizados e dos resultados obtidos que a arquitetura MCGML proposta pode contribuir para mitigar o problema de classificação de modelos desestruturados, realizando análises de similaridades e inferências em diferentes metamodelos, ou mesmo em repositórios de metamodelos.

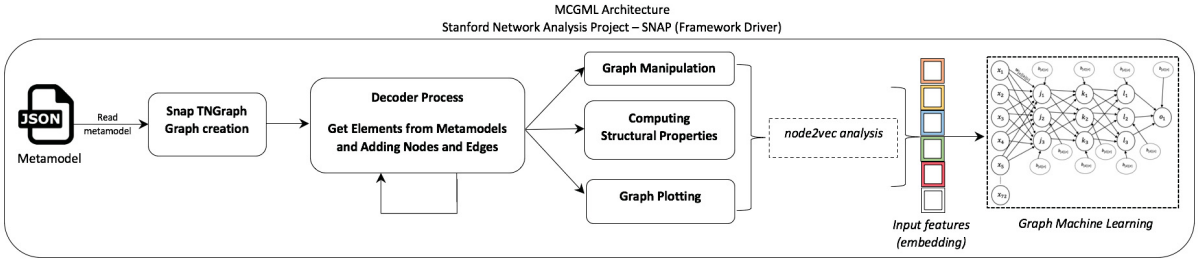



Figura 4.5: Arquitetura MCGML de manipulação de metamodelos em grafos

O primeiro passo da abordagem diz respeito ao processo de conversão de um determinado metamodelo em grafo, para ser utilizado como recurso de entrada para o *framework driver* SNAP[7]. A justificativa para utilizarmos o SNAP está na sua capacidade de obter inferências e analisar similaridades em grafos com milhares de vértices e arestas; além disso, o algoritmo *node2vec* está integrado ao SNAP, e nos permite realizar caminhadas aleatórias através da estrutura do

[7]Stanford Network Analysis Project (SNAP): A General-Purpose Network Analysis and Graph-Mining Library (Leskovec and Sosič, 2016).

grafo em busca de similaridades entre os vértices (os conceitos e as funcionalidades do algoritmo *node2vec* foram apresentados na seção 3.4.1).

O processo de conversão inicia com a leitura do metamodelo $\mathcal{M}$, onde $\mathcal{M}$ é composto de classes, atributos e referências, que são representados em uma coleção de pares chave/valor (*key/value*) no formato JSON. Em seguida é criado o grafo *Snap TNGraph* que utilizamos para realizar o processo de decodificação do metamodelo (*Decoder Process*) onde será feito a leitura de cada par chave/valor do metamodelo JSON, e realizada a adição de vértices e arestas ao grafo *Snap TNGraph*. Após a conclusão do processo de decodificação do metamodelo JSON em grafo, é possível realizar a manipulação do grafo, calcular as propriedades estruturais do grafo com o algoritmo *node2vec*, e realizar análises de similaridades através das caminhadas aleatórias em largura e profundidade. Além disso, os resultados podem ser plotados utilizando as bibliotecas *gnuplot* e *graphviz* integradas ao SNAP. A seguir detalharemos o Algoritmo 3 desenvolvido pelo autor desta tese e seus orientadores para executar o processo de leitura do metamodelo JSON e converter o mesmo em grafo, demonstrando uma simulação de plotagem do grafo utilizando a biblioteca *gnuplot*.

**Algoritmo 3** Conversão do Metamodelo em Grafo:

**Input:** Metamodelo JSON de Entrada $\mathcal{M}$
**Output:** Snap TNGraph, arquivoMetamodeloPlotado

1: $G1 \leftarrow snap.TNGraph.New()$ ▷ Criação do grafo
2: $n \leftarrow 1$ ▷ Variáveis contadoras para criar os vértices e formar as arestas
3: $i \leftarrow 2$
4: $NIdName \leftarrow snap.TIntStrH()$ ▷ Identificando os vértices
5: $data \leftarrow json.load(\mathcal{M})$ ▷ Abertura do metamodelo JSON

6: **for each** $NIdName[i]$ **in** $data[i][n]$ **do**
7: $\quad i \leftarrow i + 1$
8: $\quad G1 \leftarrow AddNode(i)$
9: $\quad G1 \leftarrow AddEdge(i, n)$
10: $\quad n \leftarrow n + 1$

11: $snap.DrawGViz(G1, snap.gvlDot, arquivoMetamodeloPlotado, tituloGrafo,$
12: $NIdName)$ ▷ Plotagem do grafo

13: **return** $SnapTNGraph, arquivoMetamodeloPlotado$

Para executar a conversão do metamodelo em grafo, definimos o Algoritmo 3 que começa com a criação do objeto $G1$ do tipo $SnapTNGraph$ que será o tipo de grafo produzido a partir do processo de decodificação do metamodelo JSON. Nas linhas 2 e 3 são definidas as variáveis contadoras $n$ e $i$ que serão utilizadas para auxiliar na criação dos vértices e na formação das arestas no grafo $G1$. Na linha 4 definimos a variável $NIdName$ utilizada para identificar o tipo de vértice $snap.TIntStrH()$. E encerrando o bloco das definições iniciais, na linha 5 temos a definição do objeto $data$ recebendo a abertura do metamodelo $\mathcal{M}$.

Entre as linhas 6 e 10 temos o laço ***for each*** onde percorremos cada par chave/valor contido em $data$ para criarmos, para cada par, uma combinação de vértices e arestas no grafo $G1$, utilizando o incremento das variáveis contadoras $n$ e $i$ que auxiliam no processo de varredura do conjunto de pares chave/valor do metamodelo contido em $data$. Após a conclusão do processo de

conversão dos pares chave/valor do metamodelo $\mathcal{M}$ para vértices e arestas no grafo $G1$, plotamos o grafo $G1$ resultante utilizando a biblioteca *gnuplot* nas linhas 11 e 12. A Figura 4.6 mostra um pequeno fragmento de um metamodelo Java $\mathcal{M}$ em JSON utilizado como entrada, e a Figura 4.7 exibe o resultado da conversão de parte deste fragmento em grafo após a plotagem.

```
{
        "XMI": {
                "EPackage": [
                        {
                                "eClassifiers": [
                                        {
                                                "eStructuralFeatures": {
                                                        "xsi:type": "ecore:EAttribute",
                                                        "name": "name",
                                                        "ordered": "false",
                                                        "unique": "false",
                                                        "lowerBound": "1",
                                                        "eType": "/1/String"
                                                },
                                                "xsi:type": "ecore:EClass",
                                                "name": "JavaElement",
                                                "abstract": "true"
                                        },
                                        {
                                                "eStructuralFeatures": {
                                                        "xsi:type": "ecore:EAttribute",
                                                        "name": "isFinal",
                                                        "ordered": "false",
                                                        "unique": "false",
                                                        "lowerBound": "1",
                                                        "eType": "/1/Boolean"
                                                },
                                                "xsi:type": "ecore:EClass",
                                                "name": "ClassFeature",
                                                "abstract": "true",
                                                "eSuperTypes": "/0/JavaElement"
                                        },
                                        ...
                                ...
                        ...
                ...
}
```

Figura 4.6: Fragmento do metamodelo Java em JSON

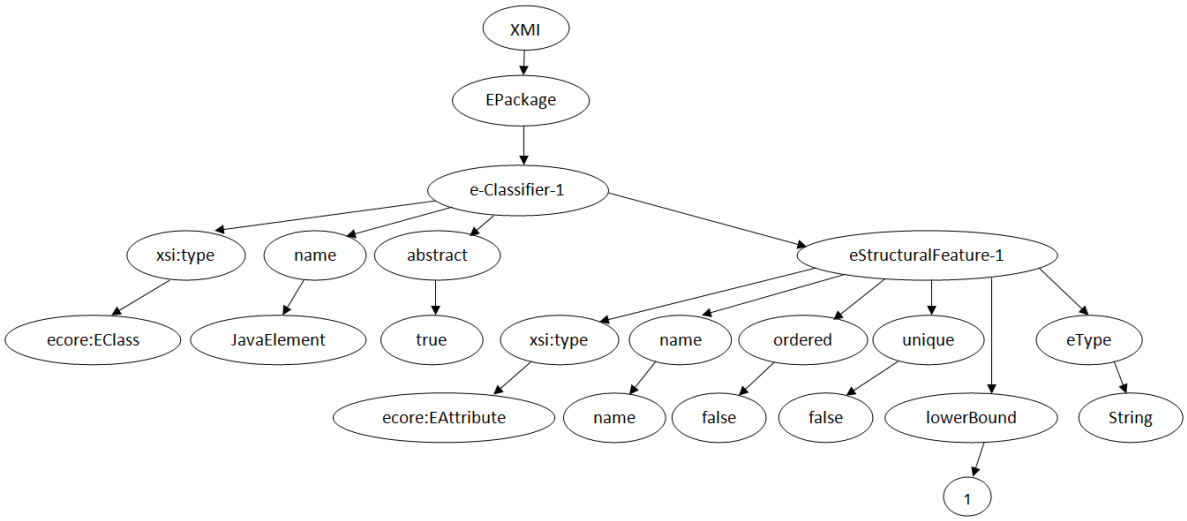



Figura 4.7: Fragmento da plotagem do grafo do metamodelo Java em JSON

A partir do retorno do grafo $SnapTNGraph$, é possível realizar manipulações no grafo, calcular suas propriedades estruturais, obter inferências, e fazer análises de similaridades através das caminhadas aleatórias via algoritmo *node2vec*, e consequentemente realizar os *embeddings* necessários que irão servir como recursos de entrada (vetor de características) para os algoritmos

de ML em grafos nas tarefas de classificação e predição. Na seção a seguir demonstramos nossos primeiros passos explorando os recursos do algoritmo *node2vec* tendo como entrada o metamodelo em grafo. O metamodelo JSON, o algoritmo de conversão em Python desenvolvido, e a imagem do grafo plotada estão disponíveis no github[8].

### 4.2.1 Analisando o Metamodelo em Grafo com o *Node2Vec*: Primeiros Passos

Vimos na seção 3.4 que *Machine Learning* (ML) em grafos é uma área de pesquisa presente em diversos tipos de aplicações que variam desde projetos na indústria farmacêutica a recomendações de amizades nas redes sociais. Porém o principal desafio nesta área é encontrar uma forma de representar ou codificar informações sob a forma de estruturas em grafos que possam ser facilmente exploradas pelos modelos de *machine learning*, pois não há uma forma direta para codificar as informações que estão em uma estrutura em grafo para um vetor de características que possa ser facilmente utilizado pelos algoritmos de ML. Na seção 3.4.1 apresentamos o ***node2vec***: um *framework* algorítmico para aprender de forma contínua a representar características dos vértices (nodos) de um grafo. Este *framework* realiza o mapeamento dos vértices em um espaço vetorial com características de dimensionalidade baixa capaz de maximizar a probabilidade de preservar as vizinhanças dos vértices do grafo através da definição de uma noção flexível para essas vizinhanças, e um procedimento de caminhada aleatória tendenciosa (*biased random walk*) capaz de explorar de maneira eficiente diversas vizinhanças, e assim obter melhores representações do aprendizado. *Node2vec* é um algoritmo semi-supervisionado capaz de aprender a representar características de maneira escalável, onde é possível representar o aprendizado sobre os vértices através dos papéis que eles desempenham no grafo, ou mesmo nas comunidades as quais pertencem. Isso é possível graças às caminhadas aleatórias tendenciosas (*biased random walk*) que exploram diversas vizinhanças de um determinado vértice. Dessa forma, o *node2vec* representa o aprendizado sobre os vértices através dos papéis que eles desempenham no grafo, ou mesmo nas comunidades as quais pertencem, pois as caminhadas aleatórias tendenciosas exploram diversas vizinhanças de um determinado vértice. Podemos controlar o espaço de pesquisa através de parâmetros configuráveis, são esses parâmetros que orientam a estratégia de pesquisa, explorando o grafo de diferentes maneiras.

Vimos também que o *node2vec* combina duas estratégias de pesquisa: a Amostragem em Lagura - *Breadth-first Sampling (BFS)*, onde a vizinhança fica restrita aos vértices que são vizinhos imediatos do vértice fonte; e a Amostragem em Profundidade - *Depth-first Sampling (DFS)*, onde a vizinhança é composta pelos vértices que fazem parte de uma amostra sequencial em distâncias crescentes a partir do vértice fonte. Essas estratégias de pesquisa são fundamentais na tarefa de predição de vértices, onde podemos analisar dois tipos de similaridades: equivalência estrutural e homofilia. Nas hipóteses de similaridade por equivalência estrutural, aqueles vértices que possuem papéis similares no grafo devem ser representados muito próximos; já nas hipóteses de homofilia, os vértices que estão altamente interconectados e pertencem a mesma comunidade ou *cluster* de grafo devem ser representados muito próximos. A estratégia de pesquisa BFS desempenha um papel chave na produção de representações que refletem o tipo de similaridade homofilia, e a estratégia DFS reflete o tipo de similaridade equivalência estrutural.

Feita essa revisão dos conceitos relacionados ao problema das representações de ML em grafos, e o algoritmo *node2vec*, nosso trabalho de pesquisa tem como foco apresentar contribuições na tentativa de mitigar o problema de classificação de modelos desestruturados em metamodelos através das análises de similaridades e inferências em diferentes metamodelos, ou mesmo em repositórios de metamodelos. Iniciamos dando um importante passo com o

[8]https://github.com/walmircouto/GraphMetamodel

trabalho de Classificação de Modelos Desestruturados em Metamodelos Utilizando Multi-Layer Perceptron (apresentado na seção 4.1), a partir daí passamos a explorar o problema ML em grafos na tentativa de preparar o metamodelo como recurso de entrada para os algoritmos de ML e com isso realizar análises de similaridades, predições e classificações de documentos. Na seção 4.2 apresentamos a principal contribuição desta pesquisa: a arquitetura MCGML para manipulação de metamodelos em grafos. Onde a partir do estudo do *framework driver* - SNAP, conseguimos converter metamodelos no formato JSON em grafos, e com isso ter a possibilidade de avançar no desenvolvimento da arquitetura proposta.

Com o processo de extração do metamodelo JSON para grafo concluído, passamos a estudar os recursos do algoritmo *node2vec* com o objetivo de explorar sua capacidade de realizar predições e análises de similaridades entre diferentes vértices em um grafo, visando obter um espaço vetorial de baixa dimensionalidade (*embedding*) que sirva de recurso de entrada para os algoritmos de ML em grafos. Criamos um algoritmo instanciando o *node2vec* para entender as propriedades dele (*node2vec*) tendo como entrada o grafo do metamodelo extraído no processo descrito no Algoritmo 3. A seguir apresentamos o Algoritmo 4 desenvolvido a partir do *node2vec* (instanciando-o) para explorar os seus recursos no grafo do metamodelo, e em seguida descrevemos o mesmo com as devidas análises e críticas.

**Algoritmo 4** Explorando o Metamodelo em Grafo com o *Node2Vec*:

**Input:** Metamodelo em grafo Snap TNGraph $G1$
**Output:** arquivoInferenciasSimilaridades, arquivoModeloInferencias

1: $arquivoInferenciasSimilaridades \leftarrow ./InferSimi.emb$ ▷ Arquivos usados na análise
2: $arquivoModeloInferencias \leftarrow ./ModelInfer.model$

3: $graph \leftarrow SnapTNGraphG1$

▷ Probabilidades predefinidas e geração das caminhadas aleatórias
4: $node2vec \leftarrow Node2Vec(graph, dimensions, walklength, numwalks, workers)$

▷ Realizando as inferências nos vértices
5: $model \leftarrow node2vec.fit(window, mincount, batchwords)$

▷ Procurando por similaridades entre os vértices, passando o grau de similaridade
6: $model.wv.mostsimilar('2')$

▷ Salvando as inferências realizadas, com suas similaridades
7: $model.wv.saveword2vecformat(arquivoInferenciasSimilaridades)$

8: $model.save(arquivoModeloInferencias)$ ▷ Salvando o modelo de inferências

9: **return** $arquivoInferenciasSimilaridades, arquivoModeloInferencias$

Os resultados das análises feitas pelo *node2vec* são armazenados em dois diferentes arquivos: o primeiro guarda as análises de inferências de acordo com as similaridades encontradas ($arquivoInferenciasSimilaridades$) que nada mais é do que o processo de *embedding* e a geração do vetor numérico de baixa dimensionalidade, e o segundo ($arquivoModeloInferencias$) guarda o resultado do modelo de inferência obtido após realizar todas as caminhadas aleatórias

tendenciosas, com os respectivos pesos atribuídos para cada passo na caminhada aleatória, as Figuras 4.8 e 4.9 mostram fragmentos dos respectivos resultados da análise de similaridades e do modelo de inferência.

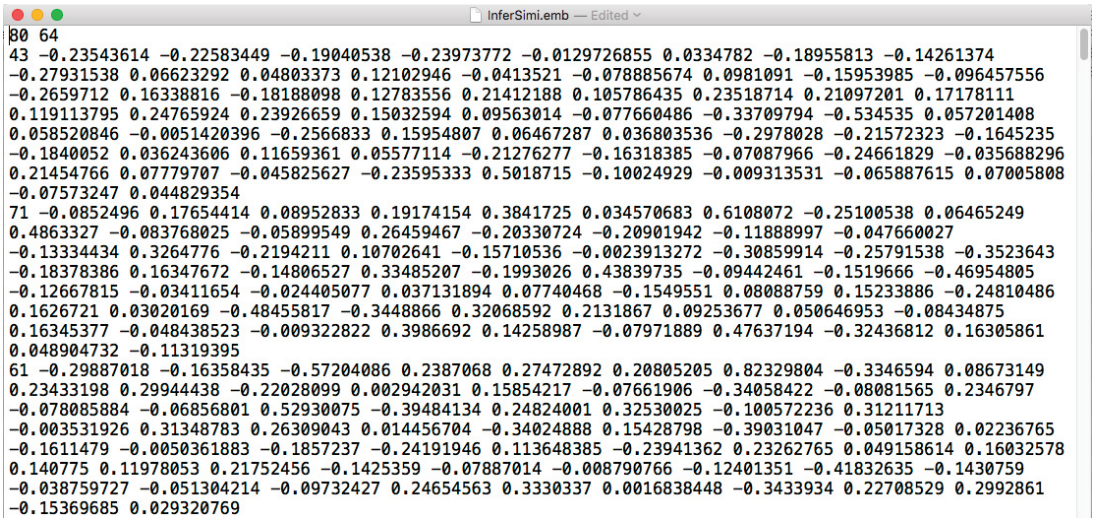



Figura 4.8: Fragmento das análises de inferências a partir das similaridades encontradas no grafo

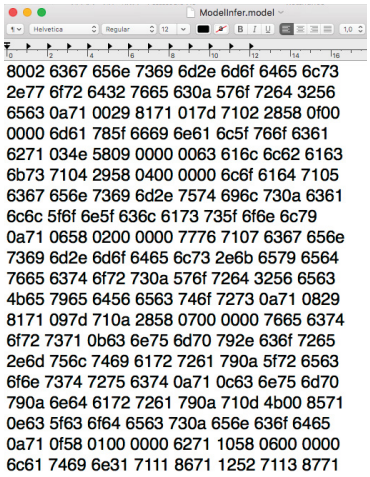



Figura 4.9: Fragmento do modelo de inferência obtido após as caminhadas aleatórias tendenciosas

Nas linhas 1 e 2 do Algoritmo 4 definimos as variáveis $arquivoInferenciasSimilaridades$ e $arquivoModeloInferencias$ para armazenar as análises de inferência de acordo com as similaridades encontradas no arquivo $InferSimi.emb$ (processo de *embedding*), e o modelo de inferência obtido será armazenado no arquivo $ModelInfer.model$. Na linha 3 definimos a variável $graph$ que vai receber o grafo do metamodelo $SnapTNGraphG1$ obtido pela extração e conversão demonstrada no Algoritmo 3.

A partir da linha 4 começamos a explorar os recursos do *node2vec* definindo as probabilidades e os parâmetros que vão guiar as caminhadas aleatórias em $graph$, os parâmetros $dimensions$, $walklength$, $numwalks$, $workers$ são usados para configurar todo o processo de varredura do grafo. Por exemplo: $walklength$ define a profundidade da caminhada, já $numwalks$ vai definir a quantidade de caminhadas aleatórias que serão realizadas. Na linha 5 temos o objeto $model$ que irá guardar o modelo de inferência usado para associar os vértices do grafo, este processo é configurado através dos parâmetros $window$, $mincount$, $batchwords$.

Na linha 6 novamente trabalhamos o objeto *model* através do método *mostsimilar* que nos permite procurar por similaridades entre os vértices do grafo, passando como parâmetro o grau de similaridade para compreendermos o impacto do grau de similaridade no resultado da busca por similaridades. E ao final, nas linhas 7 e 8 salvamos os resultados obtidos com as manipulações no objeto *model* (que guarda as inferências realizadas com as similaridades) e o modelo de inferência respectivamente em $arquivoInferenciasSimilaridades$, e $arquivoModeloInferencias$. O código referente a esses passos com o *node2vec*, os arquivos de inferências e similaridades, e o modelo de inferências estão disponíveis no GitHub[9].

Antes de analisarmos o fragmento das análises de inferências a partir das similaridades encontradas no grafo, é preciso entender que o processo de análise de similaridades em grafo nada mais é do que determinar o grau de similaridade entre vértices de um grafo (um número entre 0 e 1). Intuitivamente se conhecemos os vértices correspondentes, podemos dizer que um mesmo nó em ambos os grafos seria semelhante se seus vizinhos são semelhantes (e sua conectividade a seus vizinhos seria determinada em termos dos pesos das arestas). Este mesmo raciocínio pode ser aplicado para análise de similaridades entre vizinhanças de um mesmo grafo.

Voltando ao fragmento das análises de inferências a partir das similaridades encontradas no grafo de acordo com a Figura 4.8 temos que entre os vértices *80* e *64*, o algoritmo *node2vec* realiza caminhadas aleatórias para cada vértice que compõe a vizinhança entre os nodos *80* e *64*, e para cada vértice encontrado é descrito um conjunto de pesos utilizados para percorrer a vizinhança de acordo com o parâmetro da quantidade de passos que pretendemos realizar, é dessa forma que é feita as análises de similaridades. Já o modelo de inferência obtido após as caminhadas aleatórias tendenciosas (Figura 4.9) pode ser utilizado para a plotagem do grafo de similaridades encontradas, combinando aqueles elementos mais similares na mesma vizinhança, possibilitando redistribuir os elementos do grafo de acordo com grau de similaridade encontrado.

A seguir descrevemos os processos no desenvolvimento da Arquitetura MCGML e os desafios que tivemos na exploração dos recursos para obtermos classificações e análises de similaridades de modelos e documentos desestruturados de acordo com metamodelos existentes.

### 4.2.2 Analisando o Metamodelo em Grafo com o *Node2Vec*: Criando *Embeddings*

Após realizarmos os primeiros passos analisando metamodelos em grafo com o *node2vec*, começamos a vislumbrar potencial para desenvolver ainda mais a arquitetura MCGML apresentada na seção 4.2. A motivação teve início com a construção da nossa abordagem de classificação de modelos desestruturados utilizando redes neurais MLP apresentada na seção 4.1 pois há uma relação de propósito em contribuir para melhorar a precisão e a capacidade de análise nas classificações não só de modelos desestruturados, mas de documentos de acordo com metamodelos existentes, além da possibilidade de explorar os algoritmos de ML em grafos na análise de similaridades entre modelos e metamodelos, e com isso ter a possibilidade de realizar inferências, podendo aplicar essas análises em repositórios de metamodelos existentes. Para isso, aprofundamos e atenuamos o problema de classificação e as análises de similaridades justamente explorando os recursos que o *framework driver* SNAP e o algoritmo *node2vec* oferecem (conforme descrição da arquitetura MCGML proposta na seção 4.2).

Os desafios que tivemos foram principalmente na exploração detalhada e comparativa dos efeitos que as configurações de parâmetros podem gerar no processo de análise de similaridades através das caminhadas aleatórias tendenciosas. Um de nossos desafios de pesquisa foi a exploração do *node2vec* para se chegar a um modelo vetorial (*embedding*) do metamodelo em grafo justamente para alavancar o potencial de classificação de documentos desestruturados

[9]https://github.com/walmircouto/GraphMetamodel

utilizando algoritmos de ML em grafos, e com isso contribuir ainda mais com a área de classificação de modelos e documentos e também na área de ML em grafos.

Através das adaptações que fizemos no algoritmo *node2vec* (conforme demonstrada na seção anterior 4.2.1) foi possível construir os *embeddings* dos metamodelos em grafo, onde transformamos a topologia e as características do metamodelo em grafo para um vetor de tamanho fixo (o famoso *embedding*) responsável por representar unicamente cada nó do grafo. O *node2vec* utiliza as caminhadas aleatórias para explorar o grafo, e uma rede neural para aprender a melhor representação de cada nó no grafo. A partir desses *embeddings* do grafo conseguimos preservar as características principais do grafo enquanto reduzimos sua dimensionalidade de uma forma que podemos decodificá-lo, e com isso conseguimos capturar a estrutura e a complexidade do grafo e transformá-lo para ser usado nas tarefas de classificação e análises de similaridades usando *Machine Learning* em grafos.

### 4.2.3 *Embeddings* de Grafos: Uma Tarefa Específica de Aprendizado

Algumas abordagens supervisionadas de aprendizado de *embeddings* do grafo por inteiro foram propostas recentemente, tal como a proposta por Zhou et al. (2022). Esse tipo de abordagem oferece bom desempenho nas tarefas de aprendizado supervisionado (por exemplo: classificação de grafos), mas apresentam duas limitações críticas que reduzem seu campo de aplicação: (1) Sendo abordagens de aprendizado de representação baseadas em redes neurais, essas abordagens necessitam de um grande volume de dados rotulados para aprender representações significativas. Porém obter tais conjuntos de dados em grande volume é um desafio por si só, pois requer um enorme esforço de rotulagem. (2) As representações aprendidas através dessas abordagens supervisionadas são específicas para uma tarefa de ML em particular e não podem ser usadas ou transferidas para outras tarefas ou problemas. Por exemplo, os *embeddings* do grafo para os compostos químicos obtidos a partir do *dataset* MUTAG (Yanardag and Vishwanathan, 2015) aprendidos através da abordagem proposta por Zhou et al. (2022) são projetados especificamente para predizer se um composto tem ou não tem efeito mutagênico em uma bactéria. Dessa forma, os mesmos *embeddings* não poderiam ser usados para nenhuma outra tarefa, tal como predizer as propriedades dos compostos. Para contornar essas limitações, precisamos de uma abordagem completamente não supervisionada, ou semi-supervisionada que possa capturar sucintamente as características genéricas do grafo inteiro na forma dos seus *embeddings*. Até onde sabemos, não existem muitas técnicas disponíveis que abordem estas formas de realizar o aprendizado dos *embeddings* do grafo inteiro. Por isso, movidos por esta limitação, nesta tese, utilizamos a abordagem *node2vec* para aprender representações agnósticas de grafos de tamanhos arbitrários de uma maneira semi-supervisionada.

Na arquitetura MCGML que propomos visualizamos um grafo inteiro como um documento, e os subgrafos enraizados ao redor de cada nó (nodo) no grafo visualizamos como palavras que compõem o documento e estendemos os *embeddings* do documento como uma rede neural que aprende as representações do grafo por inteiro. Esse tipo de tratamento oferece as seguintes vantagens: (1) Representação do aprendizado de uma maneira semi-supervisionada, onde o aprendizado dos *embeddings* do grafo é realizado sem a completa necessidade de rotular as classes do grafo, permitindo a possibilidade de usarmos facilmente os *embeddings* obtidos em outras aplicações onde dados rotulados são difíceis de obter. (2) *Embeddings* orientados a dados, possibilitando o aprendizado de *embeddings* de grafos de um grande volume de dados em grafo. (3) Captura de equivalências estruturais, onde consideramos na arquitetura MCGML subestruturas não lineares, ou seja, subgrafos roteados para o aprendizado dos *embeddings*, pois as subestruturas não lineares são conhecidas por preservar as equivalências estruturais

assegurando, dessa forma, que o processo de representação do aprendizado produza *embeddings* similares para grafos de estruturas similares.

O objetivo maior da arquitetura MCGML é oferecer um apoio aos desenvolvedores em suas tarefas de metamodelagem. Para fins de ilustração, considere um desenvolvedor que deseja criar novos metamodelos para um domínio de banco de dados, embora essa tarefa possa ser generalizada para qualquer domínio na prática. Então obter uma categoria consistente de metamodelos para trabalhar com bancos de dados seria muito útil. Isto ajudaria o desenvolvedor a abordar o domínio do problema mais rapidamente obtendo um metamodelo mais adequado, diminuindo o escopo de pesquisa considerando apenas os metamodelos com a mesma categoria. De posse da categoria adequada, o desenvolvedor pode revisar e buscar aqueles metamodelos que melhor atendem as suas necessidades. Começar ou se inspirar em metamodelos existentes ajuda a lidar com a complexidade do desenvolvimento de novos artefatos, reduzindo assim o tempo de desenvolvimento e aumentando a eficiência dos processos envolvidos.

### 4.2.4 Considerações Importantes sobre a Arquitetura MCGML

Sabemos que realizar uma notação de metamodelos ou modelos desestruturados é uma tarefa tipicamente tediosa e geralmente inaplicável a uma grande quantidade de dados. Nesse contexto, é fundamental dispor de uma estrutura conveniente para automatizar tais processos. Técnicas de clusterização têm sido exploradas na tentativa de agrupar modelos e metamodelos similares (Basciani et al., 2016). No entanto, o desempenho das técnicas de clusterização depende fortemente da medida de similaridade escolhida. Em outras palavras, a tarefa de clusterização é considerada complicada em virtude de duas desvantagens principais: (i) Tempo de desempenho e a (ii) Identificação do número apropriado de *clusters*.

A área de aprendizado de máquina em grafos (*Graph Machine Learning* - GML) vem ganhando projeção nos últimos anos (Zhou et al., 2022), fomentando assim uma infinidade de aplicações, obtendo um desempenho superior em comparação às abordagens convencionais. Um algoritmo de GML tenta adquirir conhecimento do mundo real simulando autonomamente as atividades de aprendizagem do ser humano (Du et al., 2019). As soluções em GML conseguem automaticamente extrair padrões significativos a partir dos *embeddings* de grafos utilizados na fase de treinamento. Dada as circunstâncias, observamos um grande potencial de uso de GML no contexto da MDE (*Model-driven Engineering*). Curiosamente, embora os algoritmos de aprendizagem têm sido aplicados para resolver diferentes problemas, como regras de refatoração (Mokaddem et al., 2018), ou modelagem de domínio (Hartmann et al., 2019), a presença de aprendizado de máquina em grafos na MDE ainda não é compatível com o seu potencial.

A arquitetura MCGML proposta nesta tese amplia ainda mais nosso trabalho em aplicar as técnicas de aprendizado de máquina em grafos para classificar modelos desestruturados e analisar similaridades entre metamodelos através dos processos de clusterização. Uma vez que existem várias técnicas de aprendizado, e essas técnicas têm sido amplamente utilizadas em diferentes propósitos, na avaliação da arquitetura MCGML que realizamos (e que iremos demonstrar no próximo capítulo através dos experimentos realizados), comparamos os algoritmos de aprendizado de máquina em grafos presentes na arquitetura MCGML com dois algoritmos de classificação bem fundamentados: o gradiente de árvore de decisão robusta (*gradient boosted decision tree* - GBDT) (Friedman, 2000) e o *support vector machines* (SVM) (Steinwart and Christmann, 2008). Tanto o algoritmo GBDT quanto o SVM já foram explorados em uma ampla gama de aplicações, como classificação de padrões (Akbari et al., 2022), reconhecimento facial (Phillips, 1998), ou sistemas de recomendação (Jahrer et al., 2010), para citar alguns. Porém, até onde temos conhecimento, esses algoritmos raramente foram utilizados para classificar modelos

desestruturados ou metamodelos. Isso nos motivou a realizar uma investigação sobre a aplicação dos algoritmos GBDT e SVM na classificação de modelos desestruturados ou metamodelos.

Resumidamente a arquitetura MCGML trás as seguintes contribuições:

- Extração de modelos desestruturados e metamodelos JSON para grafos;
- Implementação de um classificador semi-supervisionado para categorizar metamodelos e classificar modelos desestruturados baseado nas estruturas de aprendizado de máquina em grafos;
- Definição de métricas para análises de similaridades e os algoritmos utilizados nessas atividades;
- Aproveitamento do aprendizado de representação para aprender automaticamente as características e representá-las na forma de *embeddings*;
- Implementação de algumas tarefas de GML: (1) Classificação de nós (*node classification*) para predizer uma propriedade de um nó (por exemplo: categorizar atributos de uma classe); (2) Classificação de grafos (*graph classification*) para categorizar modelos desestruturados e metamodelos representados em diferentes grafos com a possibilidade de, por exemplo, predizer propriedades entre modelos e metamodelos; (3) Clusterização para detectar se um nó (um artefato de modelagem, por exemplo) pertence a uma determinada comunidade;
- Apresentação de alguns fatores que afetam o desempenho do classificador MCGML;
- Validação experimental em um conjunto de metamodelos categorizados que foi coletado no GitHub (Nguyen et al., 2021) e transformados em grafos a partir da aquitetura MCGML;
- Comparação dos algoritmos de GML presentes na arquitetura MCGML com os algoritmos GBDT e SVM.

A Figura 4.10 abaixo dá uma visão geral sobre algumas atividades realizadas com a arquitetura MCGML:

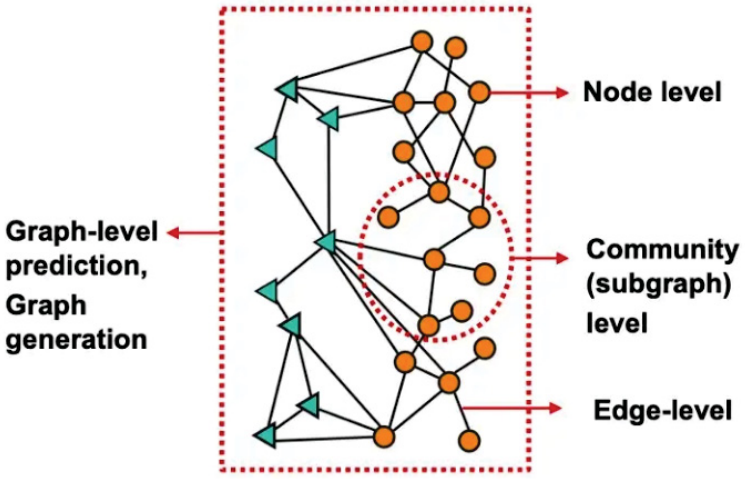



Figura 4.10: Tarefas de GML presentes na arquitetura MCGML (Hamilton et al., 2017b)

## 4.3 CONCLUSÃO

Neste capítulo apresentamos em detalhes nossa proposta de tese, demonstrando as abordagens para extração e classificação de modelos e análises de similaridades, apresentamos também nosso trabalho onde classificamos modelos desestruturados em metamodelos a partir de uma rede neural *multi-layer perceptron*, e por último exploramos em detalhes a arquitetura MCGML proposta. No próximo capítulo apresentamos todos os pontos relacionados à avaliação experimental da arquitetura MCGML, com suas motivações e *background*, demonstrando o processo de obtenção e organização dos datasets envolvidos nos experimentos, as métricas utilizadas, os recursos computacionais utilizados, e as análises dos resultados obtidos.

# 5 EXPERIMENTOS

Neste capítulo discutimos os experimentos e os resultados obtidos com o nosso estudo. Primeiro, descrevemos as bases motivacionais, o *backgound*, e os recursos computacionais utilizados em nossos experimentos. Em seguida detalhamos o fluxo de trabalho que serve de base para entender os experimentos realizados e as características da arquitetura proposta. Métricas chaves que servem de parâmetros de validação são apresentadas e discutidas. Apresentamos também um exemplo prático motivacional para deixar claro a usabilidade e a importância de realizar classificação de metamodelos. Em seguida apresentamos um *background* relacionado às tarefas de classificação e de clusterização de metamodelos em grafos. Além disso apresentamos os resultados dos experimentos de precisão e robustez desenvolvidos, realizando diferentes tarefas utilizando *graph machine learning* (GML).

## 5.1 MOTIVAÇÕES E *BACKGROUND*

Nesta seção descrevemos as motivações e o *background* que serviram de base para o desenvolvimento e validação de nosso estudo. Essa base é importante para compreender a importância de alguns aspectos experimentais aplicados na arquitetura MCGML proposta e desenvolvida durante os trabalhos de pesquisa desta tese.

### 5.1.1 Metamodelos: Usos e Aplicações

Um metamodelo especifica os conceitos abstratos de um domínio onde os conceitos e os relacionamentos entre eles são representados por uma infraestrutura de modelagem. Por exemplo, na Figura 5.1 temos uma representação simplificada do metamodelo *Ecore*. O *Eclipse Modeling Framework* (EMF) permite aos profissionais da área de modelagem projetar, definir, e editar modelos. O EMF contém o metamodelo *Ecore* para permitir a descrição de metamodelos. Todas as metaclasses representadas são nomeadas como elementos. `EPackage`, por exemplo, é composto de sub-pacotes e classes, enquanto que `EClass` é comporto por `EStructuralFeatureS`, isto é, por `EAttributeS` e `EReferenceS`. Além disso, `EClass` pode herdar características estruturais de outras classes. `EAttribute` e `EReference` herdam os atributos `lowerbound` e `upperbound` para definir as cardinalidades das características estruturais. `EReference` está lincada ao contêiner `EClass` pela referência `eReferenceType`. Além disso, `EAttributeS` são tipados como instância de `EDataType` através das referências `eAttributeType` (Benelallam et al., 2014).

### 5.1.2 Repositórios de Modelos: Bases de Exemplos

Repositórios de modelos oferecem recursos valiosos para aqueles usuários que estão interessados em adquirir conhecimento a partir de artefatos de modelagem já desenvolvidos. Nos últimos anos a adoção de repositórios vem se tornando cada vez mais comum em diferentes domínios de aplicação e em diferentes áreas, tais como: engenharias (ex: redes elétricas); biologia (ex: BioModels Database[1] e CellML[2]); engenharia de software (ex: GitHub[3] e SourceForge[4]);

[1] http://www.ebi.ac.uk/biomodels-main/
[2] http://models.cellml.org/cellml
[3] https://github.com
[4] https://sourceforge.net/

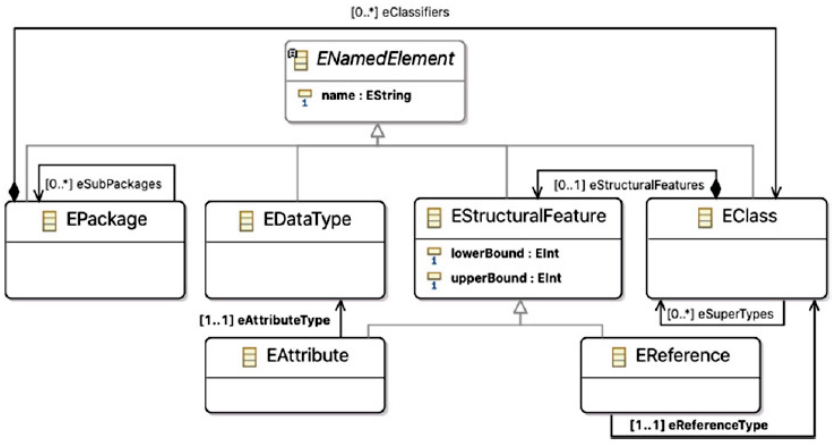



Figura 5.1: Trecho do metamodelo *Ecore* (Benelallam et al., 2014)

ou MDE *development* (ReMoDD, ATL Zoos[5], MDEForge), só para citar algumas. Uma vez que a capacidade de busca e descoberta de metamodelos impactam fortemente a prática do reuso em repositórios, cada plataforma precisa fornecer aos especialistas de domínio facilidades e algoritmos de busca avançados que são adequados para gerenciar grande volume de dados (*datasets*) e permitir a identificação rápida de itens armazenados (Nielsen et al., 2019). Só para termos uma ideia sobre as aplicações possíveis de um conjunto de dados de modelos disponíveis publicamente, podemos fazer referência ao GenMyModel[6] uma plataforma de modelagem online que oferece um conjunto predefinido de formalismos de modelagem, tais como, UML, BPMN e Flowchart. Atualmente essa plataforma oferece mais de um milhão de modelos, e possui mais de 900.000 usuários cadastrados (Ponsard et al., 2019).

Em relação aos repositórios de modelagem duplicados, um recente estudo realizado por Thongtanunam et al. (2019) revela que existe uma quantidade substancial de repositórios clonados. Através de um estudo empírico foi possível identificar arquivos de código duplicados em mais de 4,5 milhões de repositórios GitHub, incluindo os repositórios de metamodelos, os pesquisadores mostraram que mais de 40% dos arquivos estão duplicados nos repositórios analisados. Na contramão deste cenário, algumas plataformas específicas apresentam mecanismos para filtrar as duplicações. Por exemplo, usando uma função hash, a plataforma MDEForge verifica se há clones antes de adicionar novos metamodelos, e com isso ela consegue reduzir consideravelmente o número de duplicações na plataforma (Tekinerdogan et al., 2019).

Falando agora sobre o gerenciamento de repositórios de metamodelos, agrupar metamodelos armazenados em categorias facilita a pesquisa, bem como a manutenção desses repositórios. Em particular, o repositório ReMoDD permite que seus colaboradores marquem um artefato (ou seja, coloquem uma *tag*) com um conjunto predefinido de rótulos, por exemplo, tipo de artefato, palavras-chave ou domínios de sistemas/softwares (Torres et al., 2021). Já no ATL Zoos, os gestores do repositório podem atribuir uma classificação de domínio fraca para metamodelos.

No entanto, a classificação manual desses repositórios não é apenas demorada mas também suscetível a erros. Uma especificação incorreta de metadados, tais como descrição de artefatos, de rótulos ou de domínios podem comprometer substancialmente a acessibilidade, e com isso prejudicar o potencial reuso de um metamodelo. Veja o exemplo a seguir: a Figura 5.2 mostra metadados extraídos do repositório ATL Ecore Zoo para quatro metamodelos diferentes, ou seja, `xasm`, `AsmL`, `UMLDI-StateMachines` e `SyncCharts`. Através de

[5]https://bit.ly/39Re0HG

[6]https://www.genmymodel.com/

uma inspeção cuidadosa, percebemos que todos os metamodelos compartilham muitos elementos em comum, e estão relacionados com o domínio do *State Machine*. Em particular, como mostrado nas Figuras 5.3 e 5.4, existem semelhanças aos pares entre `xasm` e `AsmL`, ou entre `UMLDI-StateMachines` e `SyncCharts`. Observando as figuras, fica evidente que os metamodelos deveriam ser classificados na mesma categoria, por causa da grande quantidade de semelhanças entre eles. No entanto, o fato é que três deles foram classificados pelos gestores do ATL Ecore Zoo em três domínios independentes, enquanto que `xasm` ficou sem classificação. Só por esses exemplos percebemos que categorizar de maneira errada prejudica a acessibilidade, bem como a possibilidade dos metamodelos serem reusados (Ponsard et al., 2019).

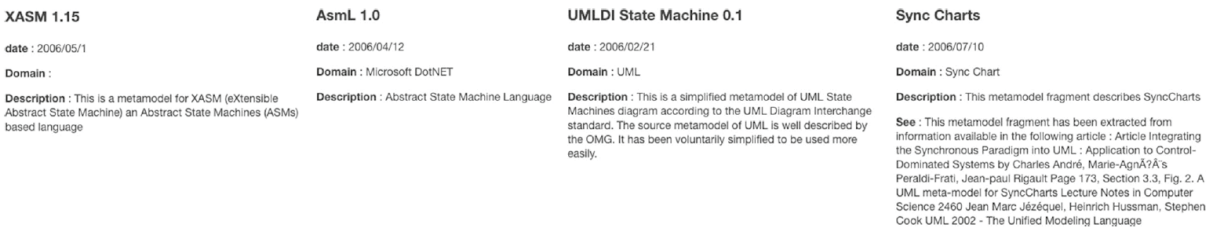



Figura 5.2: Metadados obtidos a partir do ATL Zoo

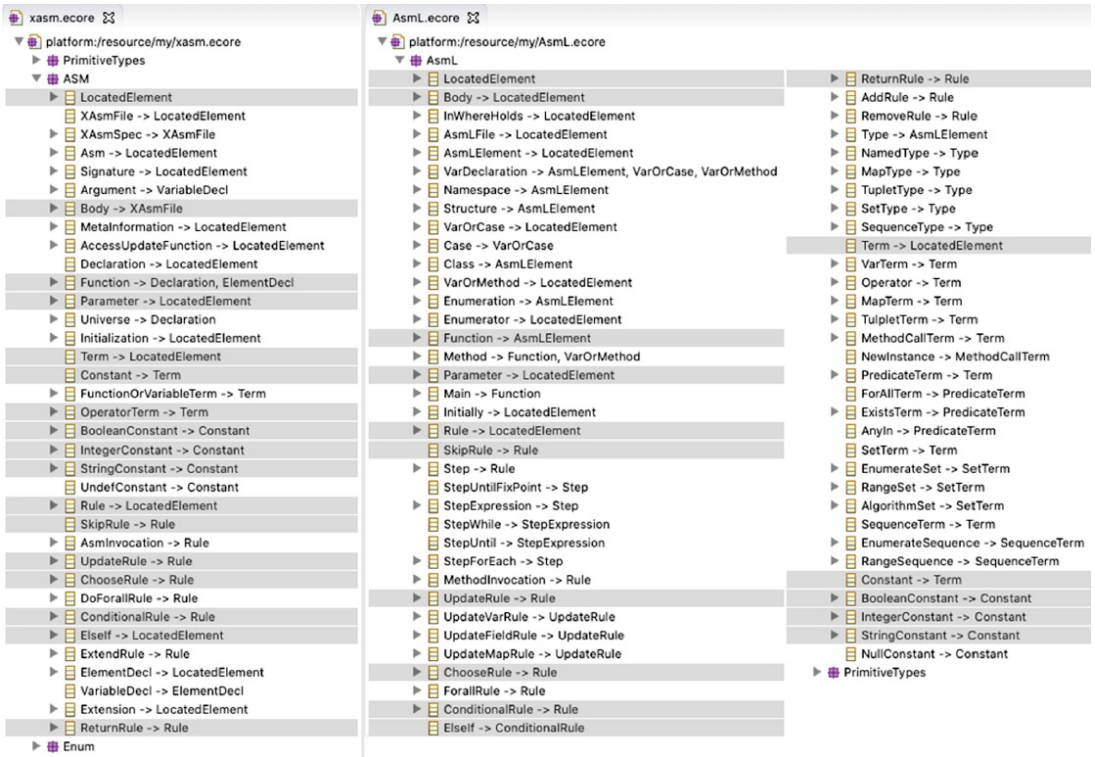



Figura 5.3: Semelhanças entre os metamodelos `xasm` e `AsmL` do ATL Zoo

### 5.1.3 A Importância das Tarefas de Clusterização e Classificação de Metamodelos

Os exemplos de repositórios apresentados na seção anterior chamam a atenção para a necessidade de encontrarmos soluções adequadas para automatizar o processo de clusterização de repositórios de metamodelos. Em outras palavras, há uma necessidade por técnicas avançadas e ferramentas para organizar automaticamente os artefatos de modelagem, especialmente quando

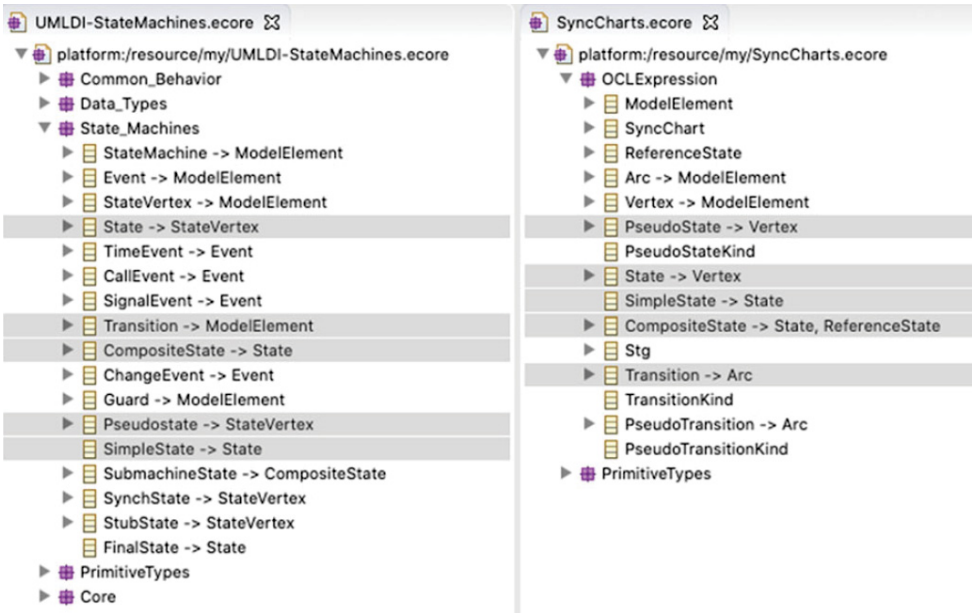



Figura 5.4: Semelhanças entre metamodelos `UMLDI-StateMachines` e `SyncCharts` do ATL Zoo

metamodelos são adicionados ou alterados com frequência (Torres et al., 2021). As tarefas de clusterização e classificação são algumas operações críticas usadas no detalhamento dos dados disponíveis para adquirir conhecimento e para identificar padrões repetitivos. Tanto a clusterização quanto a classificação têm sido amplamente utilizadas em diferentes campos, por exemplo, na medicina, na biologia, na física, na geologia, só para citar algumas. A tarefa de clusterização é uma tarefa de aprendizado não supervisionado, que visa distribuir um conjunto de objetos em diferentes grupos de acordo com alguma função de similaridade (Nguyen et al., 2017). Por ser uma tarefa não supervisionada, o número de classes não é conhecido a priori, como no caso das tarefas de aprendizagem supervisionadas. Na engenharia de software, essas técnicas têm sido aplicadas, por exemplo, na engenharia reversa e na manutenção de software para categorizar artefatos de software. Já no contexto de software de código aberto (*open-source*), categorizar projetos disponíveis através do entendimento de suas similaridades permite o reuso do código-fonte ou aprender como as similaridades presentes nos sistemas complexos foram desenvolvidas, melhorando assim a qualidade de software. Por outro lado, a tarefa de classificação é uma técnica de aprendizado supervisionado baseada na existência de classes predefinidas de objetos objetivando entender em qual classe um novo objeto pertence (Kotsiantis, 2007). Nesse sentido, dados de treinamento rotulados são pré-requisito para a tarefa de classificação, pois são necessários para orientar o processo de aprendizagem. No contexto da Arquitetura MCGML, para fins de demonstração conceitual, as metaclasses dos metamodelos apresentados na Figura 4.7 são convertidas em grafos através do Algoritmo 3 apresentado na Seção 4.2.

A possibilidade de aplicar técnicas de clusterização tem sido estudada na tentativa de organizar metamodelos reutilizáveis automaticamente. Por exemplo, a Figura 5.5 apresenta um dendrograma obtido como resultado de um processo de clusterização de um conjunto real de metamodelos, possibilitando que os modeladores possam receber uma visão geral estruturada dos metamodelos disponíveis, que normalmente são visualizados como meras listas de elementos armazenados. Dessa forma os metamodelos são automaticamente organizados em visões (*views*) como grafos conectados, onde cada grafo representa *clusters* identificados, onde os nós (nodos) correspondem a metamodelos e a similaridade entre os nós é representada como a espessura das arestas (Önder Babur, 2019).

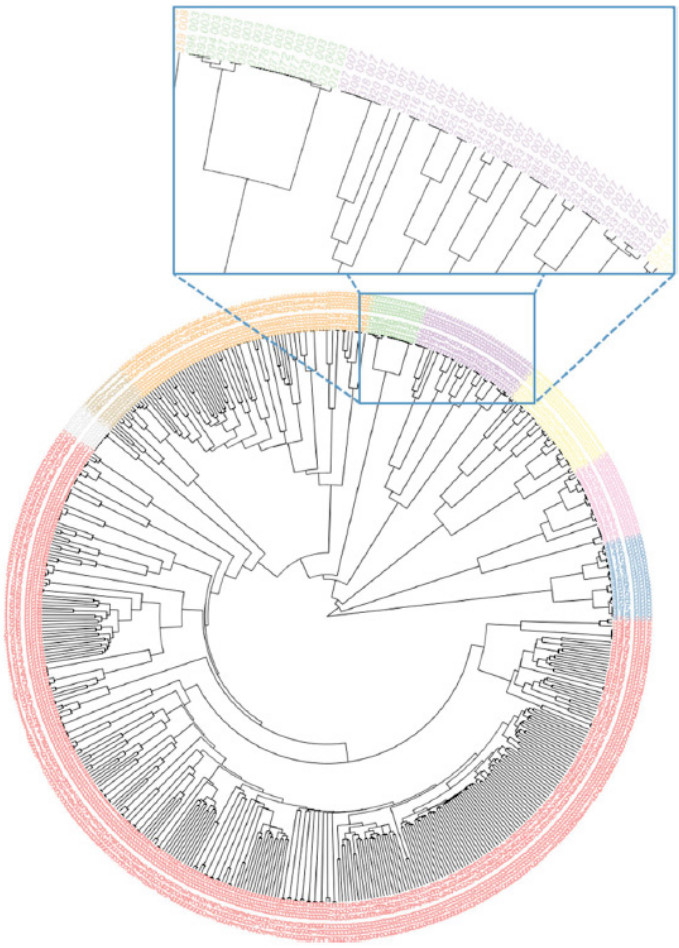


Figura 5.5: Dendograma obtido com a aplicação de um algoritmo de clusterização (Önder Babur, 2019)

### 5.1.4 Grafos e seus Fundamentos

Conforme vimos na Seção 3.6, um grafo é uma representação de conexões entre um conjunto de itens. Essa representação pode ser definida como $G = (V, E)$ , onde $V$ é um conjunto de nós (nodos - também chamados de vértices) ($V = V_1, V_2, ..., V_n$), e $E$ é um conjunto de conjuntos de dois elementos de arestas (também chamadas de links) $E = \{\{V_k, V_w\}, ..., \{V_i, V_j\}\}$ representando a conexão entre dois nós que pertencem à $V$ (Zhang et al., 2021).

Os grafos também podem ter uma série de diferentes estruturas e atributos:

- **Não direcionado (*Undirected*):** não tem direção, são úteis para os casos em que os relacionamentos são simétricos;
- **Dirigido (*Directed*):** possuem direção, são úteis para relacionamentos assimétricos;
- **Cíclico (*Cyclic*):** são aqueles onde os caminhos começam e terminam no mesmo nó, ciclos intermináveis podem impedir o término sem avisar;
- **Acíclico (*Acyclic*):** são aqueles onde os caminhos começam e terminam em nós diferentes, servem de base para muitos algoritmos;
- **Balanceado (*Weighted*):** são aqueles onde nem todos os relacionamentos são iguais, onde alguns podem ter mais peso em comparação com outros;

- **Não Balanceado (*Unweighted*):** são aqueles onde todos os relacionamentos possuem pesos iguais (de acordo com a grafo) apresentando balanceamento igual/inexistente;
- **Esparso (*Sparse*):** são aqueles onde cada nó em um subconjunto pode não ter um caminho para todos os outros nós;
- **Denso (*Dense*):** são aqueles onde cada nó em um subconjunto possui um caminho para todos os outros nós;
- **Monopartido, bipartido e k-partido (*Monopartite, bipartite, and k-partite*):** nesses tipos de grafos, ou os nós se conectam a apenas um outro tipo de nó (por exemplo, pessoas gostam de filmes), ou se conectam a muitos outros tipos de nós (por exemplo, pessoas gostam de pessoas que gostam de filmes).

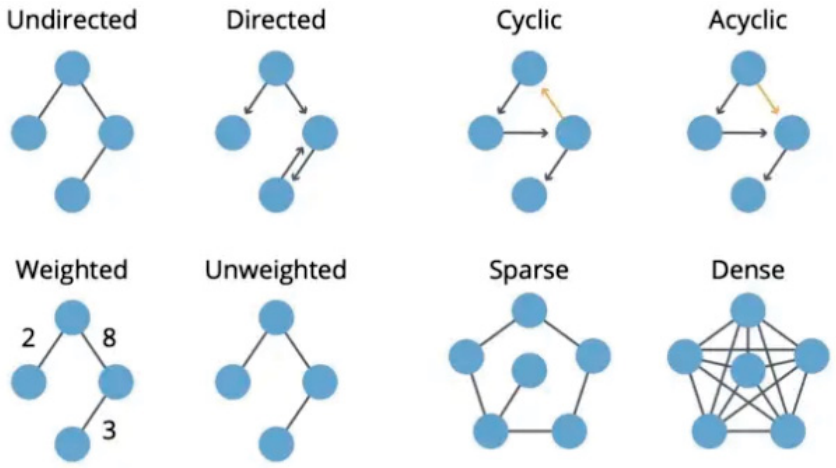



Figura 5.6: Tipos de grafos (Bayitaa et al., 2020)

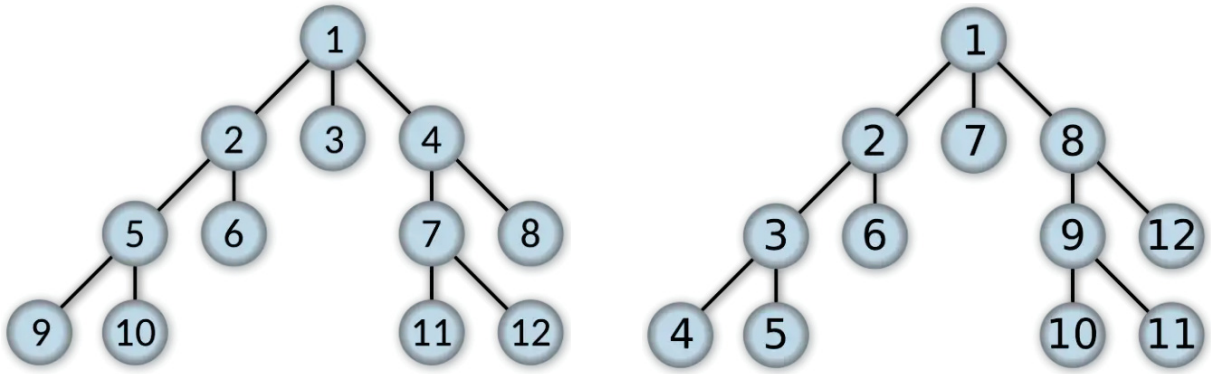
Os grafos também possuem algumas propriedades básicas sobre as quais técnicas e métodos avançados são desenvolvidos. A ordem de um grafo é o número de seus vértices $|V|$. O tamanho de um grafo é o número de suas arestas $|E|$. O grau de um vértice é o número de arestas adjacentes a ele $k_v$. Os vizinhos de um vértice $v$ em um grafo $G$ é um subconjunto do vértice $V_i$ induzido por todos os vértices adjacentes a $v$. Já o grafo de vizinhança de um vértice $v$ em um grafo $G$ é um subgrafo de $G$ composto pelos vértices adjacentes à $v$ e todas as arestas que conectam os vértices adjacentes à $v$ (Bayitaa et al., 2020).

**Conectividade de Grafos** Um grafo conectado é um grafo onde cada par de nós possui um caminho entre eles. Em um grafo pode haver vários componentes conectados, e estes componentes conectados são subconjuntos de nós, tais que: (1) todo nó no subconjunto tem um caminho para todos os outros nós; (2) nenhum outro nó tem um caminho para qualquer nó no subconjunto. **Componentes fortemente conectados** são considerados subconjuntos de nós onde: (1) todo nó no subconjunto tem um caminho para todos os outros nós; (2) nenhum outro nó tem um caminho *para* e *de* todos os nós no subconjunto. **Componentes fracamente conectados** são subconjuntos de nós de modo que a substituição de todas as suas arestas direcionadas por arestas não direcionadas produz um grafo conectado (não direcionado), ou todos os componentes são conectados por algum caminho, ignorando a direção (Hamilton et al., 2017a).

**Distância em Grafos** A distância entre dois nós é o comprimento do caminho mais curto entre eles. O comprimento do caminho é identificado pelo número de "passos"contados do começo ao fim para alcançar um nó $y$ a partir de um nó $x$. Encontrar essa distância, especialmente

em grafos de grande escala, pode ser muito custoso computacionalmente falando, por isso, para amenizar este problema existem dois algoritmos incorporados pelo *node2vec* que estão no centro da teoria dos grafos (Grover and Leskovec, 2016):

- ***Breadth-First Search* (BFS) - Pesquisa em largura:** nesse tipo de pesquisa, "descobrimos" os nós em camadas com base na conectividade. Esse tipo de pesquisa começa pelo nó raiz e encontra todos os nós na camada mais imediata de conectividade antes de continuar varrendo o grafo;
- ***Depth-First Search* (DFS) - Pesquisa em profundidade:** nesse tipo de pesquisa, a partir do nó raiz, "visitamos" os nós percorrendo todos os caminhos no grafo até o primeiro nó folha antes de seguir uma rota diferente no grafo.




(a) Pesquisa BFS (b) Pesquisa DFS

Figura 5.7: Tipos de pesquisa em grafos

Quando queremos agregar esses tipos de pesquisa (BFS e DFS) a um nível de grafo, há duas maneiras de fazer isso: (1) Calculando a **distância média** (*Average Distance*) que é a distância média entre cada par de nós. Essa distância média geralmente é restrita ao maior componente quando o grafo é do tipo não conectado; (2) Calculando o **diâmetro** (*Diameter*) que é a distância máxima entre um par qualquer de nós. Tanto a distância média quanto o diâmetro devem ser usados em pares sobre um domínio de dados conhecidos sendo modelado como um grafo (Hamilton et al., 2017a). Utilizamos essas métricas para realizarmos análises de similaridades entre metamodelos em grafos a partir da arquitetura MCGML proposta.

**Clusterização de Grafos** Conforme vimos na Seção 5.1.3, a clusterização é uma importante tarefa que podemos realizar com grafos para decompor e entender sua complexidade. O fechamento triádico em um grafo é a tendência daqueles nós que compartilham arestas e se tornam conectados. Isso é comumente feito em grafos de redes sociais quando uma pessoa $A$ é amiga da pessoa $B$, e a pessoa $B$ é amiga da pessoa $C$, portanto, uma recomendação para a pessoa $A$ pode sugerir uma amizade com a pessoa $C$. Isso é fundamentado em evidências presentes na maioria das redes no mundo real, principalmente nas redes sociais, os nós tendem a criar grupos coesos representados por uma densidade relativamente alta de laços (Grover and Leskovec, 2016). Uma maneira de medir a tendência de clusterização em um grafo é através do **coeficiente de clusterização**. Existem duas formas de medir o coeficiente de clusterização: uma local e outra global:

- **Coeficiente de Clusterização Local (*Local Clustering Coefficient*):** seria a fração de pares de amigos de um nó que são amigos entre si. Esse coeficiente é obtido pela fórmula:

$$Coeficiente\ de\ Clusterizacao\ Local = \frac{Qtde\ pares\ amigos\ de\ A\ que\ sao\ amigos\ entre\ si}{Qtde\ pares\ de\ amigos\ de\ A}$$

Podemos medir o coeficiente de clusterização, medindo o quão conectados os vizinhos do nó $v$ estão:

$$e_v = \frac{Qtde\ de\ arestas\ entre\ nos\ vizinhos}{\binom{k_v}{2}} \in [0, 1]$$

No exemplo a seguir temos a quantidade de arestas entre os nós vizinhos ao nó $v$ igual a 6, e o grau do nó $v$ igual a 4 ($k_v = 4$), logo o binômio $\binom{k_v}{2} = \binom{4}{2} = 6$, chegando assim nos seguintes coeficientes de clusterização:

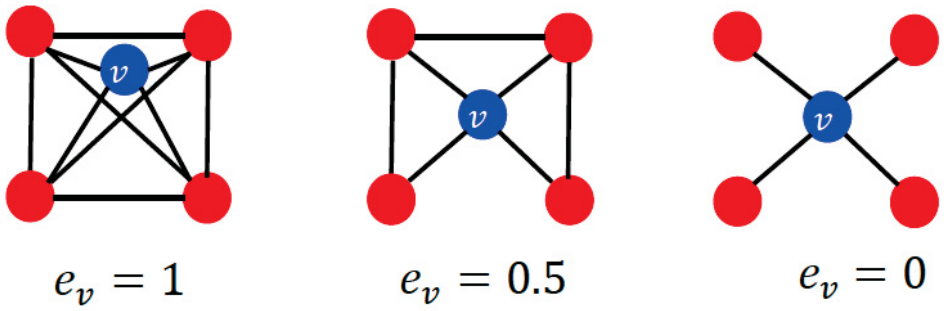



Figura 5.8: Exemplo de Coeficiente de Clusterização (You et al., 2022)

- **Coeficiente de Clusterização Global (*Global Clustering Coefficient*):** para calcular este coeficiente temos duas abordagem. (1) O **Coeficiente de Clusterização Local Médio** sobre todos os nós no grafo; (2) E a **Transitividade** que é um percentual de tríades abertas que formam triângulos no grafo. São os nós balanceados com graus mais altos.

A Tabela 5.1 a seguir mostra o sumário do resultado apresentado após uma pequena análise de similaridade feita em um metamodelo UML em grafo a partir da instanciação do algoritmo *node2vec*:

**Grau de Distribuição do Grafo** O grau de um nó em um grafo não-direcionado é o número de vizinhos que ele possui. O grau de distribuição de um grafo é a distribuição de probabilidade dos graus em todo o grafo. Em grafos não-direcionados, o grau de distribuição do grafo é simplesmente referido como grau, mas para grafos direcionados, podemos obter distribuições de grau de entrada (*in-degree*) e de grau de saída (*out-degree*). As distribuições de graus de entrada representam a distribuição de links de entrada (*in-links*) que cada nó no grafo possui. Argumenta-se que, em grafos do mundo real (principalmente nas redes sociais), quando plotamos as distribuições de graus de entrada, grau a grau, em uma escala log-log, isso representa uma distribuição da lei de potência. Essas propriedades em grafos sem escala são na verdade muito raras quando usamos técnicas estatisticamente rigorosas. De qualquer forma, avaliar o grau de distribuição é importante para entender um grafo, mas essa propriedade não fornece informações sobre como o grafo pode evoluir ao longo do tempo (Leskovec and Sosič, 2016).

**Centralidade do Grafo** Embora possamos manipular um grafo muito pequeno e compreensível, sabemos que os grafos podem facilmente escalar para uma complexidade dificilmente interpretável. Quando os grafos ficam muito grandes, é importante utilizar medidas de centralidade para nos auxiliar na compreensão dos dados. A centralidade do grafo é uma

Tabela 5.1: Resultado da análise de similaridade feita no metamodelo UML em grafo via *node2vec*

| **Graph Summary** | |
|---|---|
| Number of nodes : | 115 |
| Number of edges : | 613 |
| Maximum degree : | 12 |
| Minimum degree : | 7 |
| Average degree : | 10.660869565217391 |
| Median degree : | 11.0 |
| **Graph Connectivity** | |
| Connected Components : | 1 |
| **Graph Distance** | |
| Average Distance : | 2.5081617086193746 |
| Diameter : | 4 |
| **Graph Clustering** | |
| Transitivity : | 0.4072398190045249 |
| Average Clustering Coefficient : | 0.40321601104209814 |

forma de pensar sobre a importância dos nós e arestas em um grafo. Dependendo do nosso domínio de dados, podemos usar diferentes suposições e isso permitirá avaliar diferentes medidas de centralidade. As formas que podemos quantificar a "importância"em um grafo são através: da quantidade de graus de conectividade, da proximidade média a outros nós, da fração de caminhos mais curtos que passam pelo nó, etc. A seguir listamos algumas aplicações para as quais as medidas de centralidade de grafos podem ser usadas (Hamilton et al., 2017a):

- Encontrar nós influentes em uma rede social;
- Identificar nós que disseminam informações para vários nós;
- Identificar *hubs* em uma rede de transporte;
- Identificar páginas importantes na web;
- E no caso da MCGML, identificar principais elementos de um metamodelo.

Existem diversas medidas de centralidade em grafos que podemos utilizar, porém iremos listar algumas que utilizamos como métricas em nosso domínio de informações (metamodelos e modelos desestruturados em grafos):

- **Grau de Centralidade** (nós importantes possuem várias conexões): Esta é a medida mais básica de centralidade: número de vizinhos. Essa métrica usa o grau para grafos não direcionadas e grau de entrada ou grau de saída para grafos direcionadas. Os valores do grau de centralidade são normalizados dividindo-se pelo grau máximo possível em um grafo simples $n - 1$ onde $n$ é o número de nós em $G$;
- **Centralidade de Proximidade** (nós importantes estão próximos a outros nós): Esta métrica indica a reciprocidade da distância média do caminho mais curto para um

nó sobre todos os outros $n - 1$ nós alcançáveis. Se os nós estiverem desconectados, podemos ou considerar sua centralidade de proximidade com base apenas nos nós que podem alcançá-lo ou podemos considerar apenas os nós que podem alcançá-lo e normalizar esse valor pela fração de nós que ele pode alcançar;

$$Centralidade\ de\ Proximidade\ C(x) = \frac{N}{\sum_y d(x,y)}$$

A Centralidade de Proximidade do grafo inteiro nada mais é do que o somatório das Centralidades de Proximidade de cada nó individualmente, onde um nó será importante se tiver o menor caminho mais curto para todos os outros nós, onde:

$$c_v = \frac{1}{\sum_{u \neq v} tamanho\ do\ caminho\ mais\ curto\ entre\ u\ e\ v}$$

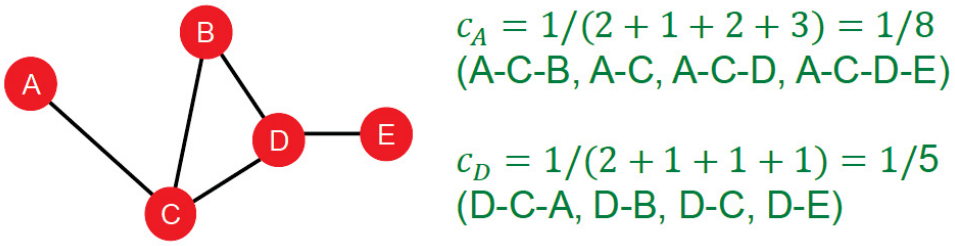



Figura 5.9: Exemplo de Centralidade de Proximidade (You et al., 2022)

- **Centralidade de Intermediação** (nós importantes estão próximos a outros nós): A centralidade de Intermediação de um nó $v$ é a soma da fração dos caminhos mais curtos de todos os pares de nós que passam pelo nó $v$. A fórmula abaixo analisa o número de caminhos mais curtos entre os nós $s$ e $t$ que passam pelo nó $v$ e o divide pelo número total de caminhos mais curtos entre $s$ e $t$ (e as somas sobre todos os caminhos que não começam ou não terminam em $v$). Os valores da centralidade de intermediação serão maiores em grafos com muitos nós. Para controlar isso, dividimos os valores desta centralidade pelo número de pares de nós no grafo (excluindo o nó $v$). A complexidade desta métrica surge quando há vários caminhos mais curtos no grafo. Como calcular esta métrica pode ser muito custoso computacionalmente, pode ser melhor calcular essa métrica para uma amostra de pares de nós. Esta métrica também pode ser usada para encontrar arestas importantes;

$$Centralidade\ de\ Intermediacao\ g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

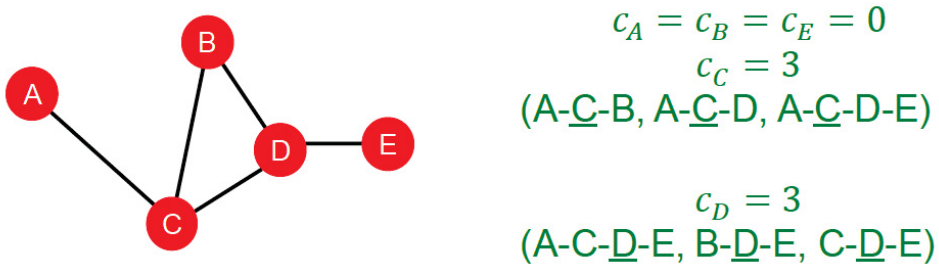



Figura 5.10: Exemplo de Centralidade de Intermediação (You et al., 2022)

- **Centralidade de Autovetor** (nós importantes estão conectados a nós centrais): A centralidade do autovetor calcula a centralidade de um nó com base na centralidade de seus vizinhos. Essa métrica mede a influência de um nó em um grafo. Pontuações relativas são atribuídas a todos os nós do grafo com base no conceito de que conexões entre nós de alta pontuação contribuem mais para a pontuação do nó em questão do que conexões iguais entre nós de baixa pontuação. Uma alta pontuação nesta métrica significa que um nó está conectado a muitos nós que possuem pontuações altas. Dessa forma um nó $v$ é importante se ele estiver rodeado por nós vizinhos importantes $u$, onde $u \in N(v)$. Podemos modelar a centralidade do nó $v$ através do somatório das centralidades dos nós vizinhos;

  $$Centralidade\ de\ Autovetor\ c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u$$

  Onde $\lambda$ é a constante de normalização (que será o maior autovalor da matriz de adjacências $A$), onde $A_{uv} = 1$ se $u \in N(v)$. E $c$ é o vetor de centralidade. Podemos observar que a equação acima modela a centralidade de maneira recursiva. E como podemos resolver isso? Reescrevendo a equação recursiva na forma de matriz.

  $$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u \Leftrightarrow \lambda c = Ac$$

  Podemos ver que a centralidade $c$ é o autovetor da matriz de adjacências $A$. O maior autovalor $\lambda_{max}$ sempre será positivo e único (comprovado pelo teorema de Perron Frobenius). O autovetor $c_{max}$ correspondente ao $\lambda_{max}$ é usado no cálculo da centralidade.

- **PageRank** (nós importantes são aqueles com muitos links de entrada de outros nós importantes) O algoritmo de *PageRank* realiza os seguintes passos: (1) Ele começa a partir de um nó aleatório; (2) Atribui a todos os nós um *PageRank* de $1/n$; (3) Escolhe uma aresta de saída aleatoriamente e segue-a até o próximo nó; (4) Executa a regra básica de atualização do *PageRank*: cada nó fornece uma parte igual de seu *PageRank* atual para todos os nós com os quais ele está vinculado; (5) O novo *PageRank* de cada nó é a soma de todos os *PageRank* recebidos de outros nós; (6) E por último repete este processo $k$ vezes;

- **Autoridades e *Hubs* - *Hubs Authorities* (HITS)** (nós importantes que possuem arestas de entrada de bons *hubs* são boas autoridades, e nós que possuem arestas de saída para boas autoridades são bons *hubs*): O algoritmo de HITS realiza os seguintes passos: (1) Ele atribui para cada nó uma pontuação de autoridade e de *hub* igual a `1`; (2) Aplica a Regra de Atualização da Autoridade: a pontuação de autoridade de cada nó é o

somatório das pontuações de *hub* de cada nó que aponta para ele; (3) Aplica a Regra de Atualização do *Hub*: a pontuação de *hub* de cada nó é o somatório das pontuações de autoridade de cada nó que aponta para ele; (4) Normaliza as pontuações de Autoridade e *Hub* de cada nó pela pontuação total de cada um; (5) E por último repete este processo $k$ vezes.

O algoritmo HITS começa pelo nó raiz ou por um conjunto de nós altamente relevantes (autoridades potenciais). Em seguida todos os nós que se ligam ao nó raiz são hubs potenciais. A base é definida como nós raízes e qualquer nó que se conecte a um nó na raiz. Todas as arestas que conectam os nós no conjunto base são consideradas, e isso concentra-se em um subconjunto específico do grafo que é relevante para uma consulta específica. Apresentamos os resultados das aplicações deste algoritmo nas análises de similaridades feitas em metamodelos e modelos desestruturados na seção de avaliações experimentais da arquitetura MCGML.

### 5.1.5 Baselines: os algoritmos *Gradient Boosted Decision Tree* (GBDT) e *Support Vector Machines* (SVM)

Algumas abordagens de *machine learning*, nos últimos anos, foram propostas para lidar com os problemas de classificação (Kotsiantis, 2007). Nesta seção, revisamos duas abordagens que utilizamos como *engines* de classificação de metamodelos nesta tese, o *Gradient Boosted Decision Tree* (GBDT) e o *Support Vector Machines* (SVM). Essas abordagens, tanto a GBDT, quanto o SVM, vêm sendo aplicadas em diversos domínios, e os seus desempenhos vêm sendo estudados pela comunidade da MDE na tentativa de mitigar os problemas de classificação de metamodelos (Wang et al., 2022).

Em *machine learning*, os classificadores fracos são aqueles que são modificados para melhorar sua capacidade de predição. Isso pode ser feito restringindo os classificadores de usar todas as informações disponíveis, ou através da simplificação de suas estruturas internas. Essas modificações permitem que os classificadores aprendam a partir de diferentes partes de um conjunto de treinamento, resultando dessa forma em uma redução na correlação de erro. O GBDT é um modelo de árvore de decisão para predizer um rótulo de destino (ou seja, uma classificação de destino). Durante a fase de aprendizado, o GBDT preenche uma série de árvores de decisão fracas, onde cada uma delas tenta corrigir os erros da árvore anterior. Em outras palavras, o GBDT aprende confiando no seguinte pressuposto: as observações que um classificador de árvore de decisão fraca pode manipular são ignoradas, enquanto que as observações difíceis remanescentes estão sujeitas a desenvolver novas árvores de decisão fracas (Friedman, 2000).

O algoritmo *Support Vector Machines* (SVM) pode ser usado também para classificar metamodelos. Primeiro ele representa metamodelos como pontos rotulados em um espaço N-dimensional; e então, para aprender a partir dos dados, o SVM extrai hiperplanos que melhor dividem os pontos entre as categorias. Um hiperplano pode ser entendido como uma fronteira entre duas classes. A dimensão do hiperplano varia de acordo com o número de características de entrada, ou seja, $N$ características. Desta forma, a fase de aprendizagem é um processo de otimização que tenta encontrar hiperplanos maximizando a distância entre pontos em classes independentes. Sempre que um metamodelo não rotulado é acrescentado no SVM, ele é transformado em um ponto de dados; e dessa forma, os hiperplanos gerados são usados para atribuir um rótulo ao metamodelo (Steinwart and Christmann, 2008).

Em nossa avaliação experimental, comparamos o desempenho de ambos classificadores com a abordagem MCGML proposta.

## 5.2 AVALIAÇÃO EXPERIMENTAL

Nesta seção, analisamos o desempenho da arquitetura MCGML proposta através de uma avaliação baseada em métricas de similaridades, e em métodos os quais realizam *embeddings* no nível de nó no grafo, que é o caso do *node2vec*. Para isso coletamos os metamodelos a partir de repositórios públicos como o GitHub. Sabemos que metamodelos podem ser coletados de diferentes fontes, porém no escopo desta tese fazemos uso do GitHub como o principal provedor de dados, uma vez que esta plataforma oferece um grande número de repositórios de metamodelos. A avaliação experimental realizada nesta pesquisa está pautada em duas hipóteses principais ($\mathbf{H}_1$ e $\mathbf{H}_2$), que deram origem a quatro perguntas de pesquisa:

- $\mathbf{H}_1$**:** A abordagem MCGML apresenta índices melhores de recomendação de metamodelos em relação aos algoritmos de aprendizado supervisionado, o *Gradient Boosted Decision Tree* (GBDT) e o *Support Vector Machines* (SVM).

- $\mathbf{H}_2$**:** Uma recomendação de artefatos de modelo usando um treinamento genérico baseado em grafos da MCGML (onde os conceitos de domínios são extraídos do relacionamento entre modelo e metamodelo) é equiparável à outra recomendação de modelo realizada para uma abordagem tradicional de ML treinada para um domínio específico.

Esta seção começa apresentado as perguntas de pesquisa, em seguida descrevemos o conjunto de dados usado para avaliarmos nossa proposta, a partir da Seção 5.2.2 explicamos a metodologia aplicada, e por último apresentamos algumas análises de similaridades realizadas utilizando as métricas descritas na Seção 5.1.4.

### 5.2.1 Perguntas de Pesquisa (PPs)

Avaliamos o desempenho da arquitetura MCGML proposta através das seguintes perguntas de pesquisa:

- $\mathbf{PP}_1$**:** Como podemos comparar os benefícios das tarefas de classificação e clusterização em grafos presentes na arquitetura MCGML com os algoritmos GBDT e SVM? Sabemos que as redes neurais são técnicas de *machine learning* bem difundidas. Em nossa pesquisa, comparamos as atividades de classificação e clusterização realizadas através da arquitetura MCGML com dois algoritmos de aprendizado supervisionado, o *Gradient Boosted Decision Tree* (GBDT) e o *Support Vector Machines* (SVM).

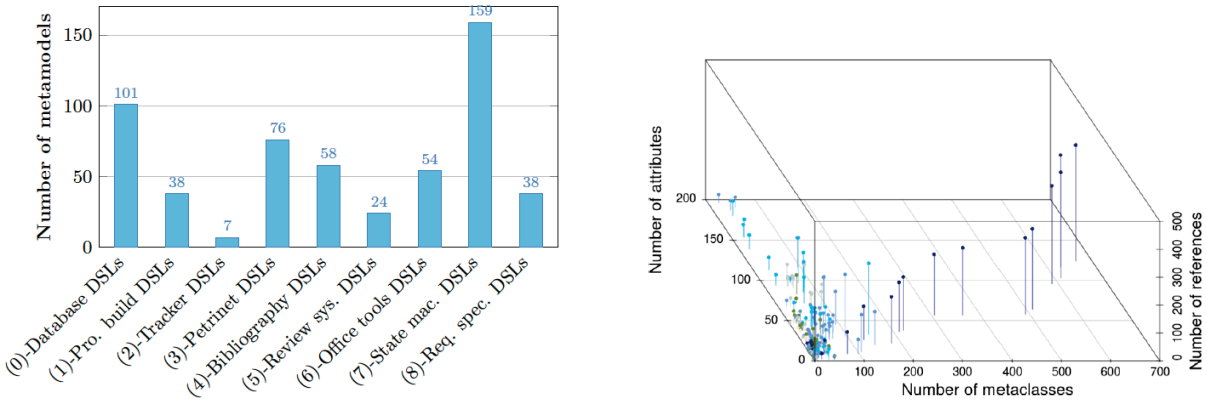
- $\mathbf{PP}_2$**:** É possível predizer uma propriedade de um nó (por exemplo: categorizar artefatos de metamodelos) utilizando os métodos de classificação de nós (nodos)? Realizamos tarefas de ML em nível de nó para realizar predições baseadas na sequência de artefatos que compõem os metamodelos.

- $\mathbf{PP}_3$**:** É possível predizer novas conexões a partir daquelas existentes, onde pares de nós, os quais não possuem links, são classificados, e os $K$ principais pares de nós são preditos? Realizamos tarefas de ML em nível de aresta para realizar predições de links baseadas nas interações entre os artefatos que compõem os metamodelos, recomendando possíveis ligações entre eles.

- **PP$_4$:** É possível predizer propriedades relacionadas a grafos inteiros ou subgrafos, permitindo realizar classificações binárias de documentos de acordo com metamodelos através da classificação de grafos? Realizamos tarefas de ML em nível de grafo para realizar predições de propriedades do grafo como um todo, possibilitando realizar classificações binárias.

### 5.2.2 Dataset: base de metamodelos

Nesta avaliação experimental fazemos uso de um dataset indexado[7] que consiste em um repositório de metamodelos relacionados a linguagens de domínio específico (*domain-specific languages* - DSLs), este dataset foi inspecionado e classificado manualmente por especialistas da área de modelagem com base em suas experiências, conforme trabalho realizado por Önder Babur (2019). Este dataset possui 555 metamodelos distribuídos em 9 categorias, a distribuição da quantidade de metamodelos por categoria está resumida na Figura 5.11(a). Já a Figura 5.11(b) mostra a distribuição dos artefatos dos metamodelos de acordo com a quantidade de metaclasses, a quantidade de atributos, e a quantidade de características (Nguyen et al., 2021). Observando a Figura 5.11(a), a categoria que possui a maior quantidade de metamodelos (de acordo com a classificação realizada por Önder Babur (2019)) é a categoria *State machine DSLs* com 159 metamodelos, isso corresponde à 29% dos metamodelos do repositório. Enquanto que a categoria com a menor quantidade de metamodelos é a *Issue Tracker DSLs* com apenas 7 metamodelos. Essa categoria com uma quantidade pequena de metamodelos pode dificultar as tarefas de classificação e clusterização nos *embeddings* em nível de grafo, uma vez que quanto maior a quantidade de grafos como elementos de entrada do conjunto de grafos $\mathcal{G} = \{G_1, G_2, ..., G_n\}$ maior será a quantidade de *embeddings* em nível de grafo, e consequentemente isso aumenta também a precisão das análises de similaridades baseadas nas matrizes de adjacências dos respectivos grafos (onde $\mathbf{A} \in \mathbb{R}^{|V \times V|}$ é a matriz de adjacências do grafo), conforme explicado na Seção 3.6.1.




(a) Quantidade de metamodelos por categoria

(b) Pesquisa DFS

Figura 5.11: Especificações do dataset (Nguyen et al., 2021)

Os metamodelos foram classificados manualmente, seguindo percepção humana baseada na experiência dos profissionais envolvidos no trabalho de Önder Babur (2019). Essa classificação pode variar na prática dependendo dos especialistas envolvidos, quando se trata do domínio da aplicação. Para simplificar, consideramos o domínio da aplicação como um projeto baseado em vários metamodelos, mesmo com alguns deles podendo ser classificados em categorias

[7]https://doi.org/10.5281/zenodo.2585456

variadas, dependendo do nível de granularidade presente. Para exemplificar, as categorias de metamodelos *Office-tools DSLs*, e *State Machine DSLs* podem ser consideradas como categorias de metamodelos DSLs de propósito geral, que implementam domínios específicos por definição. Além disso observamos que pode haver uma possível sobreposição entre categorias; por exemplo, a categoria *Review systems DSLs* parece ter muito em comum com a categoria *Bibliography DSLs*. Esses metamodelos vão orientar as tarefas de *machine learning* presentes na arquitetura MCGML, e uma vez convertidos em grafos através do Algoritmo 3 podemos realizar os *embeddings* em nível de grafo ou nó. Desta forma, assumimos que, dependendo dos metamodelos de entrada, podemos obter diferentes níveis de granularidade nos resultados das tarefas de classificação e clusterização.

### 5.2.3 Configurações dos Experimentos

A técnica de validação cruzada *tenfold* foi utilizada para avaliar o desempenho da arquitetura MCGML, uma vez que esta técnica está entre os melhores métodos para seleção de modelos em ML (Kohavi, 1995). Esta técnica de validação consiste em dividir o dataset original em dez partes iguais, chamados de *folds*, e numerá-los de 1 a 10. Para cada configuração de teste, a validação é conduzida em dez rounds. Para cada round $i$, o $fold_i$ é usado como conjunto de teste e os outros nove *folds* remanescentes $fold_j$ $(j \neq i, 1 \leqslant i, j \leqslant 10)$ são combinados para formar um conjunto de treinamento para os *embeddings* em nível de grafo através do conjunto de grafos $\mathcal{G} = \{G_1, G_2, ..., G_n\}$. Os metamodelos de treinamento são usados para encontrar as melhores configurações dos *embeddings* utilizando a instanciação que fizemos do algoritmo *node2vec* na Seção 4.2.1, ou seja, identificar os melhores parâmetros de configuração das caminhadas aleatórias que produzem as melhores análises de similaridades desejadas, a partir dos metamodelos em grafos de entrada, enquanto os metamodelos de teste são usados para avaliar a arquitetura proposta.

É importante destacar que todo metamodelo presente no conjunto de teste possui uma categoria rotulada (*label*) associada a ele, em uma espécie de aprendizado semi-supervisionado. No entanto, na fase de validação, esses rótulos são removidos e salvos como dados de referência. Os *embeddings* em nível de grafo são construídos usando os esquemas de codificação das matrizes de adjacências mencionados na Seção 3.6.1. Onde cada *embedding* produzido no processo de *encoder* dos metamodelos em grafos gera um rótulo, e este rótulo é comparado com àquele salvo como dados de referência para ver se ambos rótulos são correspondentes. Espera-se que os rótulos de todos os metamodelos de teste correspondam aos rótulos salvos usados como dados de referência.

O processo de validação simula uma implementação real da seguinte maneira: os dados de treinamento correspondem aos repositórios que estão em domínio público, disponíveis para serem utilizados; eles foram coletados de diferentes fontes e classificados manualmente por especialistas, ou seja, quando os metamodelos são criados, os próprios criadores o classificam. Enquanto que o dataset de teste representa os repositórios que precisam ser classificados. Assim, a validação experimental tenta analisar se a arquitetura MCGML pode ser aplicada a uma implementação real, como por exemplo, sendo adaptada para classificar metamodelos em repositórios como o GitHub. Para facilitar a reprodutibilidade, disponibilizamos no GitHub[8] todos os artefatos de software e dados utilizados na arquitetura MCGML juntamente com todos os metadados analisados e produzidos a partir do dataset original.

[8]https://github.com/walmircouto/GraphMetamodel

### 5.2.4 Métricas de Avaliação

Considerando $\mathcal{M}$ como um conjunto de metamodelos, e $\mathcal{G}$ um conjunto de grafos produzidos a partir da conversão de cada metamodelo presente em $\mathcal{M} = \{M_1, M_2, ..., M_n\}$ utilizando o Algoritmo 3 (Seção 4.2) resultando no conjunto de grafos $\mathcal{G} = \{G_1, G_2, ..., G_n\}$, onde cada $G_i$ presente em $\mathcal{G}$ precisa ser classificado em uma determinada categoria $c$ presente no conjunto de categorias $C = \{c_1, c_2, ..., c_n\}$. Para uma determinada categoria $c \in C$, definimos $\hat{Y}_c$ como o conjunto de metamodelos que precisam ser classificados na categoria $c$ por qualquer classificador, ou seja, o resultado da classificação tem que ser o mesmo via arquitetura MCGML, ou via os algoritmos GBDT e SVM. E o conjunto $Y_c$ definimos como sendo o conjunto atual dos metamodelos que realmente estão categorizados em $c$ de acordo com os dados de referência, ou seja, aqueles que foram rotulados manualmente por especialistas.

Antes de explicarmos as métricas, precisamos apresentar as notações utilizadas no contexto das validações dos experimentos, fazendo referência ao diagrama de Venn da Figura 5.12 (Rokach and Maimon, 2006).

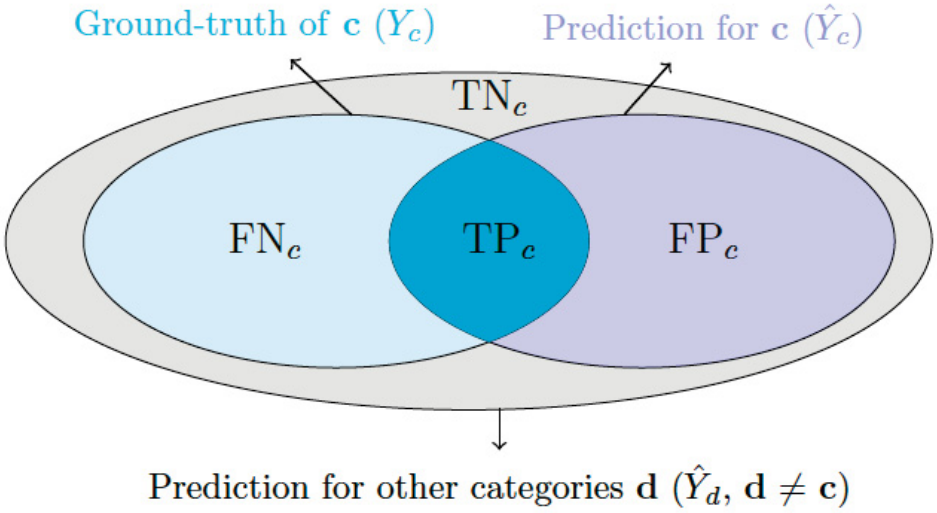



Figura 5.12: Relações entre $TP_c$, $TN_c$, $FP_c$ e $FN_c$ (Rokach and Maimon, 2006)

A partir de uma categoria $c \in C$, temos as seguintes definições:

- Verdadeiro Positivo (*True Positive - TP*): corresponde à quantidade de metamodelos que são classificados na categoria $c$, e realmente eles pertencem à $c$ de acordo com os dados de referência (*ground-truth data*), ou seja, $TP_c = \left|\hat{Y}_c \bigcap Y_c\right|$.
- Verdadeiro Negativo (*True Negative - TN*): corresponde à quantidade de metamodelos que são classificados em outras categorias diferentes de $c$, e eles não pertencem à $c$ de acordo com os dados de referência (*ground-truth data*), ou seja, $TN_c = \sum_{d \in C, d \neq c} \left|\hat{Y}_d\right|$.
- Falso Positivo (*False Positive - FP*): corresponde à quantidade de metamodelos que são classificados na categoria $c$, e eles não pertencem à $c$ de acordo com os dados de referência (*ground-truth data*), ou seja, $FP_c = \left|\hat{Y}_c \setminus Y_c\right|$.
- Falso Negativo (*False Negative - FN*): corresponde à quantidade de metamodelos que não são classificados na categoria $c$, e eles pertencem à $c$ de acordo com os dados de referência (*ground-truth data*), ou seja, $FN_c = \left|Y_c \setminus \hat{Y}_c\right|$.

Nesta primeira avaliação da arquitetura MCGML, nosso interesse é entender o quão bem as categorias produzidas casam com aquelas que foram classificadas manualmente pelos especialista no trabalho de Önder Babur (2019). Mais precisamente, medimos a relevância dos resultados classificados obtidos a partir da execução de três ferramentas de classificação, ou seja, MCGML, e os algoritmos GBDT, e SVM com os dados de referência, utilizando as métricas de avaliação: acurácia (*accuracy*), precisão (*precision*), *recall*, *F1-score*, *ROC* e *AUC*. As definições formais dessas métricas são apresentadas nas subseções a seguir.

#### *5.2.4.1 Acurácia (accuracy)*

Acurácia é a razão entre a quantidade de itens classificados corretamente e a quantidade total de itens, independentemente de suas categorias. Logo temos:

$$Accuracy = \frac{\sum_{c \in C} TP_c}{|S|} \times 100\%$$

#### *5.2.4.2 Precisão (precision) e Recall*

A precisão ($P$) mede a fração da quantidade de metamodelos classificados corretamente para a categoria $c$ em relação a quantidade total de metamodelos classificados nesta categoria. Já o *Recall* é a razão entre a quantidade de itens classificados corretamente para a categoria $c$ em relação à quantidade total de itens presentes nos dados de referência (*ground-truth data*) de $c$. O *Recall* é também conhecido como taxa de verdadeiros positivos (*True Positive Rate - TPR*).

$$P_c = \frac{TP_c}{TP_c + FP_c}$$

$$R_c = TPR_c = \frac{TP_c}{TP_c + FN_c}$$

#### *5.2.4.3 F1-score (F1)*

O *F1-score* é calculado a partir da combinação de *Precisão* e *Recall*.

$$F1_c = \frac{2 \cdot P_c \cdot R_c}{P_c + R_c}$$

#### *5.2.4.4 Taxa de Falsos Positivos (False Positive Rate - FPR)*

Essa taxa mede a proporção da quantidade de itens que são erroneamente classificados na categoria $c$ em relação à quantidade total de itens (tanto aqueles que são corretamente não classificados, quanto aqueles que são erroneamente classificados dentro de uma determinada categoria).

$$FPR_c = \frac{FP_c}{TN_c + FP_c}$$

Após a obtenção dessas métricas individualmente por categoria, calculamos as métricas de avaliação da arquitetura proposta utilizando as médias das pontuações obtidas para todas as categorias.

$$P = \frac{\sum_{c \in C} P_c}{|C|}$$

$$R = \frac{\sum_{c \in C} R_c}{|C|}$$

$$F1 = \frac{\sum_{c \in C} F1_c}{|C|}$$

#### *5.2.4.5 ROC e AUC*

A métrica ROC (*Receiver Operator Characteristic*) representa o relacionamento entre as taxas $FPR$ e $TPR$ (Davis and Goadrich, 2006). Nesta avaliação da arquitetura MCGML trabalhamos com classificação de multi-classes, onde um metamodelo precisa ser classificado em apenas uma classe de categoria de nove categorias consideradas. Porém as métricas ROC são utilizadas em classificações binárias em virtude de sua própria natureza. Sendo assim, transformamos o problema de classificação de multi-classes em um problema de classificação binária utilizando a seguinte solução alternativa: para cada categoria específica (entre as nove categorias consideradas), ou seja, para cada categoria $c_i$, nós a consideramos uma classe positiva, enquanto todas as outras oito categorias restantes são consideradas classes negativas. Isso significa que, se um metamodelo $m$, onde $m \in \mathcal{M}$, for corretamente classificado em sua classe positiva, então contabilizamos como verdadeiro (***true***). Da mesma forma, quaisquer metamodelos do conjunto $\mathcal{M}$ que não pertençam à categoria $c_i$ e não sejam classificados na categoria $c_i$ também serão contabilizados como verdadeiros (***true***).

Por outro lado, se o metamodelo $m$ for classificado em qualquer uma das classes negativas, será contabilizado como falso (***false***), ou se quaisquer metamodelos que não pertencem à categoria $c_i$, mas que foram classificados em $c_i$, também serão contabilizados como falsos (***false***). Dessa forma, ao invés de realizarmos uma classificação multi-classes, realizamos uma classificação binária, onde existem duas classes distintas: as Verdadeiras (***true***) (ou seja, a verdadeira positiva e a verdadeira negativa), e as Falsas (***false***) (ou seja, a falsa positiva e a falsa negativa). Um bom classificador deve atribuir uma probabilidade alta para o primeiro enquanto determina uma probabilidade baixa para o último. Em outras palavras, espera-se que o classificador tenha uma baixa taxa de $FPR$ e uma alta taxa de $TPR$. Podemos definir um limite para predizer as classificações com elevada confiabilidade, determinando uma certa probabilidade de ser diferente, assim a métrica ROC-AUC é calculada utilizando estes limites, permitindo compreender facilmente o desempenho em diferentes níveis de decisão para os quais os resultados obtidos pelo classificador são aceitos. O percentual obtido com os valores ROC-AUC é um indicador de quão preciso é um classificador. Um classificador simples, ou seja, aquele que atribui aleatoriamente um rótulo aos metamodelos, tem um percentual ROC-AUC de 50%, enquanto um classificador perfeito tem um percentual de ROC-AUC de 100%.

Na seção a seguir apresentamos em detalhes os resultados experimentais fazendo referência às perguntas de pesquisa da Seção 5.2.1.

## 5.3 RECURSOS COMPUTACIONAIS

Nesta seção descreveremos brevemente os recursos computacionais utilizados para desenvolver e avaliar nossa abordagem. Essas informações podem auxiliar na reprodutibilidade de nossos experimentos em futuras pesquisas com base na tese proposta.

O PyTorch é um *framework* de *deep learning* e *Pythonic machine learning* de código aberto. Ele tem como objetivo oferecer suporte ao desenvolvimento de trabalhos voltados à pesquisa. Ele possui diferenciação automática (ou seja, a responsabilidade das derivações computacionais é transferida do programador para o *framework*). Além disso, seu núcleo de tipos de dados (seus *datatype core*), ou seja, seus *Tensors* são muito semelhante aos *arrays* NumPy, proporcionando um aprendizado rápido do *framework* para aqueles que já estão familiarizados com ele (Paszke et al., 2019). Utilizamos também ferramentas de análise de grafos em Python,

como o SNAP.PY, e o NetworkX. Além é claro das implementações dos algortimos GBDT e SVM na biblioteca *scikit-learn*[9].

Nossos experimentos foram realizados em um computador com 12GB DDR3 RAM, processador de 2,53 GHz Intel Core i5, sistema operacional MacOS High Sierra 10.13.6, e o *framework* PyTorch 1.7.1. Todos os códigos para reproduzir nossos experimentos estão disponíveis no GitHub[10] e servem de ponto de partida para pesquisas futuras sobre os datasets utilizados.

## 5.4 RESULTADOS EXPERIMENTAIS

A partir da execução dos experimentos, calculamos Acurácia (*Accuracy*), Precisão ($P$), *Recall* ($R$), e o *F1-score* ($F1$) para cada configuração de teste nas tarefas de classificação multi-classes, e utilizamos o *ROC-AUC* nas tarefas de classificação binária. Começaremos respondendo a $\mathbf{PP}_1$, onde comparamos a Acurácia (*Accuracy*), Precisão ($P$), *Recall* ($R$), e o *F1-score* ($F1$) obtidos com os resultados das classificações multi-classes (em categorias) via MCGML, GBDT e SVM. Para responder as demais perguntas de pesquisa exploraremos os *embeddings* em nível de nó, nível de arestas (*links*), e nível de grafo através das propriedades e métricas descritas na Seção 5.1.4.

### 5.4.1 $\mathbf{PP}_1$: Como podemos comparar os benefícios das tarefas de classificação e clusterização em grafos presentes na arquitetura MCGML com os algoritmos GBDT e SVM?

Assim como as redes neurais, os algoritmos GBDT e SVM estão entre os algoritmos de aprendizado de máquina mais bem fundamentados. Esses algoritmos têm sido amplamente utilizados em vários domínios, tais como classificação de padrões, reconhecimento facial, segmentação de imagens, ou em sistemas de recomendação (Diem et al., 2013). Para esta pergunta de pesquisa, realizamos experimentos para analisar como as tarefas de classificação e clusterização feitas através da arquitetura MCGML funcionam em comparação com os algoritmos GBDT e SVM em termos de precisão. Além disso, objetivando uma comparação justa, executamos validação cruzada (*cross-validation*) de *tenfold* utilizando exatamente os mesmos dados de entrada para todos os três classificadores, ou seja, MCGML, GBDT, e SVM[11]. Os resultados dos experimentos são apresentados na Tabela 5.2.

Em relação à Acurácia (*Accuracy*), na grande maioria das comparações a MCGML obteve uma melhor precisão em comparação com GBDT e SVM. Por exemplo, na tarefa de classificação binária, com $c = 2$, a MCGML obteve 93.20% de Acurácia, enquanto que os percentuais de Acurácia obtidos pelos algoritmos GBDT e SVM foram de 91.05% e 72.15% respectivamente. Porém realizando tarefas de classificação multi-classes, onde $c = 6$, a MCGML obteve uma leve piora em sua Acurácia, obtendo 92.23%, sendo superada pelo GBDT que obteve 93.40%, mas superando o SVM que ficou com 79.56% de Acurácia.

No *Recall* ($R$), o GBDT é o classificador que obtém o melhor desempenho em quase todas as configurações, isto é, em $c = \{2, 6, 8\}$. Isso significa que o GBDT é bom para recuperar os metamodelos que pertencem aos dados de referência. Porém quando consideramos o *F1-score* ($F1$), fica claro que a MCGML tem o melhor desempenho de classificação, pois sempre atinge o maior valor de $F1$ para todas as configurações das tarefas de classificação, ou seja, em todo

[9] https://scikit-learn.org

[10] https://github.com/walmircouto/GraphMetamodel

[11] Fizemos uso das implementações dos algortimos GBDT e SVM na biblioteca ***scikit-learn*** (https://scikit-learn.org)

Tabela 5.2: Comparação de precisão da tarefa de classificação entre MCGML, GBDT e SVM

| | **c = 2** | | | **c = 4** | | | **c = 6** | | | **c = 8** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | MCGML | GBDT | SVM | MCGML | GBDT | SVM | MCGML | GBDT | SVM | MCGML | GBDT | SVM |
| Acurácia (%) | **93.20** | 91.05 | 72.15 | **92.83** | 91.67 | 78.92 | 92.23 | **93.40** | 79.56 | **91.78** | 90.84 | 85.37 |
| Precisão (P) | **0.934** | 0.789 | 0.598 | **0.929** | 0.864 | 0.622 | 0.915 | **0.921** | 0.642 | **0.933** | 0.907 | 0.823 |
| *Recall* (R) | **0.905** | 0.894 | 0.762 | **0.912** | 0.872 | 0.786 | 0.873 | **0.914** | 0.732 | **0.905** | 0.897 | 0.849 |
| *F1-score* | **0.921** | 0.841 | 0.672 | **0.916** | 0.863 | 0.691 | 0.881 | **0.897** | 0.689 | **0.912** | 0.901 | 0.827 |

$c = \{2, 4, 6, 8\}$. No todo, levando em consideração todas as métricas de avaliação, chegamos a conclusão que a MCGML obteve os melhores resultados nas tarefas de classificação multi-classes.

**Resposta da $\mathbf{PP}_1$:** a arquitetura MCGML obteve as melhores métricas de qualidade nas tarefas de classificação frente ao GBDT e SVM.

### 5.4.2 $\mathbf{PP}_2$: É possível predizer uma propriedade de um nó (por exemplo: categorizar artefatos de metamodelos) utilizando os métodos de classificação de nós (nodos)?

Antes de começarmos a responder a $\mathbf{PP}_2$, iremos explicar como organizamos nosso *benchmark* de grafos baseado nos datasets indexados de metamodelos presentes nos repositórios relacionados a linguagens de domínio específico (*domain-specific languages* - DSLs), estes datasets possuem 555 metamodelos distribuídos em 9 categorias (Önder Babur, 2019), conforme apresentado na Seção 5.2.2. Primeiramente convertemos os 555 metamodelos em grafos utilizando o Algoritmo 3 (Seção 4.2) resultando no conjunto de grafos $\mathcal{G} = \{G_1, G_2, ..., G_{555}\}$, sendo o conjunto $\mathcal{G}$ o *benchmark* de grafos da arquitetura MCGML. E é justamente este *benchmark* que nos dá o suporte para realizarmos nossas avaliações com base nas principais tarefas de ML em grafos (*graph ML tasks*), isto é, predições de propriedades dos grafos, predições de arestas *links*, e predições de nós, onde cada uma dessas tarefas requer modelos para realizar predições em diferentes níveis nos grafos, ou seja, nível de nó, nível de aresta, e nível do grafo inteiro.

Como cada grafo no *benchmark* $\mathcal{G}$ pertence a um repositório de acordo com sua categoria (conforme a distribuição apresentada na Figura 5.11(a)), para cada categoria de repositório, criamos datasets de grafos $\mathcal{D} = \{D_1, D_2, ..., D_9\}$, onde detalhamos cada dataset $D_n \subset \mathcal{G}$ focando nas propriedades dos grafos e nas tarefas de predição. Os detalhes desta distribuição de datasets estão apresentados na Tabela 5.3. Também comparamos esses datasets de grafos analisando as estatísticas dos grafos que os compõem, por exemplo: os graus dos nós (*node degree*), o coeficiente de clusterização (*clustering coefficient*), e o seu diâmetro. Os detalhes dessas comparações são apresentados na Tabela 5.4. Analisando a Tabela 5.4 podemos observar que os datasets exibem diferentes características de grafos. Essas diferenças nas características do gráfico resultam na diferença inerente em como as informações se propagam nos grafos, podendo afetar significativamente o comportamento de alguns modelos de ML, entre eles os *embeddings* em nível de nós baseados nas caminhadas aleatórias (*random-walk-based*) via algoritmo *node2vec*. No geral, essa diversidade nas características dos grafos tem origem nos diversos domínios de aplicação e é de suma importância para avaliar a versatilidade dos modelos de ML em grafos.

Após apresentarmos como definimos e organizamos nosso *benchmark* de grafos baseado nos datasets indexados de metamodelos presentes nos repositórios relacionados a linguagens de domínio específico (*domain-specific languages* - DSLs), voltamos à análise da $\mathbf{PP}_2$ que trata da possibilidade de realizarmos predições de propriedades de um nó (*node-level prediction*) através da MCGML. Sabemos que para realizarmos as principais tarefas de ML em grafos: predição nível de nó (node-level prediction), predição nível de aresta (link-level prediction), e predição nível de grafo (graph-level prediction), precisamos extrair as características dos nós, das arestas, e dos grafos para um espaço vetorial de baixa dimensionalidade: os *embeddings* (Figura 5.13).

Tabela 5.3: Distribuição dos datasets que compõem o *benchmak* $\mathcal{G}$. As categorias de tarefas de ML são: predição de propriedades em nós (*node*), em arestas (*link*), e em grafos (*graph*)

| Categorias de Tarefas de ML | Repositórios de Metamodelos em Grafos | Quantidade de Grafos | Tipo de Tarefa | Métrica Utilizada |
|---|---|---|---|---|
| **Nó (node)** | `Database DSLs` | 101 | Classificação Multi-class | Acurácia (Accuracy) |
| | `Pro.build DSLs` | 38 | Classificação Multi-class | Acurácia (Accuracy) |
| | `Tracker DSLs` | 7 | Classificação Multi-class | Acurácia (Accuracy) |
| **Aresta (link)** | `Petrinet DSLs` | 76 | Predição de Aresta (Link prediction) | Hubs Authorities (HITS) |
| | `Bibliography DSLs` | 58 | Predição de Aresta (Link prediction) | Hubs Authorities (HITS) |
| | `Review sys.DSLs` | 24 | Predição de Aresta (Link prediction) | Hubs Authorities (HITS) |
| **Grafo (graph)** | `Office tools DSLs` | 54 | Classificação Binária (Binary class.) | ROC |
| | `State mac.DSLs` | 159 | Classificação Binária (Binary class.) | ROC |
| | `Req.spec.DSLs` | 38 | Classificação Binária (Binary class.) | ROC |

Tabela 5.4: Características estatísticas dos grafos que compõem o *benchmak* $\mathcal{G}$. Utilizamos a biblioteca SNAP.PY para calcular as estatísticas dos grafos, onde o diâmetro médio dos grafos é calculado realizando uma caminhada BFS a partir de 1000 nós escolhidos aleatoriamente

| Categorias de Tarefas de ML | Repositórios de Metamodelos em Grafos | Quantidade de Grafos | Quantidade de nós | Quantidade de arestas | Média dos graus dos nós | Média dos Coeficientes de Clusterização | Diâmetro Médio |
|---|---|---|---|---|---|---|---|
| **Nó (node)** | `Database DSLs` | 101 | 6824 | 27296 | 32.4 | 0.145 | 11 |
| | `Pro.build DSLs` | 38 | 2522 | 10088 | 212.7 | 0.247 | 14 |
| | `Tracker DSLs` | 7 | 483 | 1932 | 75.2 | 0.483 | 21 |
| **Aresta (link)** | `Petrinet DSLs` | 76 | 4769 | 19076 | 47.7 | 0.014 | 14 |
| | `Bibliography DSLs` | 58 | 8413 | 33652 | 15.0 | 0.741 | 29 |
| | `Review sys.DSLs` | 24 | 3984 | 15936 | 89.4 | 0.409 | 16 |
| **Grafo (graph)** | `Office tools DSLs` | 54 | 8421 | 25263 | 18.3 | 0.289 | 13 |
| | `State mac.DSLs` | 159 | 13479 | 53916 | 46.7 | 0.148 | 19.4 |
| | `Req.spec.DSLs` | 38 | 5946 | 17838 | 38.9 | 0.842 | 20.7 |

Utilizar características efetivas em grafos é o ponto chave para obtermos bom desempenho do modelo de aprendizado.

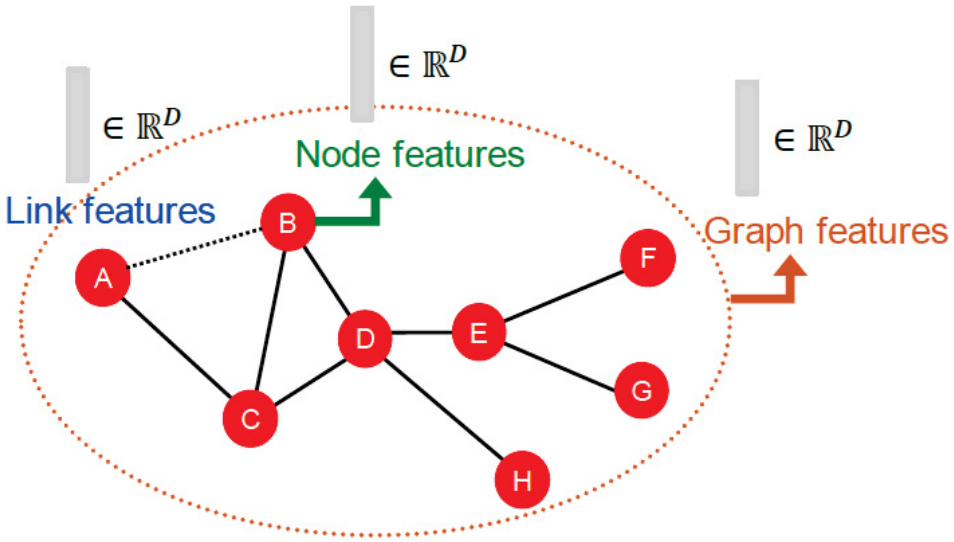



Figura 5.13: Extração de características para espaços vetoriais (You et al., 2022)

As principais tarefas de ML em grafos têm como objetivo realizar predições em um conjunto de objetos, mas para isso acontecer precisamos fazer boas escolhas de projeto, e isso inclui:

- Recursos (*Features*): seriam os vetores de baixa dimensionalidade (*d-dimensional vectors*).
- Objetos: nós, arestas, conjunto de nós, e grafos inteiros.
- Função objetiva: qual tarefa queremos solucionar?

Dessa forma, para solucionarmos a tarefa de predição de propriedades de um nó (respondendo a $\mathbf{PP}_2$) dado um determinado grafo $G = (V, E)$, precisamos aprender uma função $f : V \rightarrow \mathbb{R}$. E essa função de aprendizado $f$ é obtida pelos *embeddings* dos grafos. Em nossa avaliação experimental relacionadas à tarefa de predição de propriedades de um nó temos 3 datasets extraídos a partir de 3 categorias de repositórios de metamodelos (`Database DSLs`, `Pro.build DSLs`, e `Tracker DSLs` - Tabela 5.4). Onde o `Database DSLs` representa o dataset dos metamodelos em grafos relacionados às DSLs de banco de dados, o `Pro.build DSLs` são grafos relacionados a metamodelos de criação de uma DSL, e o dataset `Tracker DSLs` são grafos relacionados ao rastreamento de artefatos de metamodelos. Todos esses datasets de grafos possuem diferentes escalas, tarefas, e incluem grafos homogêneos e heterogêneos.

Os 3 datasets apresentam estatísticas dos grafos diferentes, conforme demonstrado na Tabela 5.4. Podemos observar que os grafos do `Pro.build DSLs` são muito mais densos em comparação aos grafos do `Database DSLs` e do `Tracker DSLs` pois os grafos do `Pro.build DSLs` possuem uma média dos graus dos nós muito elevada e um diâmetro médio mais baixo em comparação com os demais. Por outro lado, os grafos do `Tracker DSLs` têm uma estrutura de seus grafos mais clusterizada em comparação aos outros datasets, isso pode ser observado pela alta média dos coeficientes de clusterização. E finalmente os grafos que compõem o `Database DSLs` possuem características interessantes, apresentando simultaneamente pequena média dos graus dos nós, menor média nos coeficientes de clusterização, e o menor diâmetro médio. O **baseline** dos experimentos feitos para responder a $\mathbf{PP}_2$ está pautado no *node2vec* um *framework* algorítmico que utiliza como entrada a concatenação de nós crus e os *embeddings* (Grover and Leskovec, 2016).

O **`Database DSLs`** é formado por grafos que representam metamodelos relacionados à área de banco de dados, descrevendo metadados, e todos os artefatos presentes em projetos de banco de dados. Os nós podem representar `schemas`, `tables`, `attributes`, `columns`, `references`, `keys`, etc. As arestas entre dois nós indicam como todos esses artefatos se relacionam.

**Tarefa de Predição:** a tarefa que utilizamos nesta avaliação foi predizer em qual categoria de artefatos um artefato (nó) relacionado ao domínio `Database DSLs` está classificado em uma configuração de classificação onde listamos 15 categorias de alto-nível que são usadas para serem nossos rótulos alvo (*target labels*).

**Divisão do Dataset (*Dataset splitting*):** assumimos a divisão clássica, onde separamos randomicamente 90% dos nós presentes em `Database DSLs` para fase de treinamento, e os 10% restantes para os testes.

Tabela 5.5: Resultados da tarefa de predição de nós no dataset `Database DSLs` - tarefa de classificação

| **Método** | **Acurácia (%)** | | |
|---|---|---|---|
| | Treinamento | Validação | Testes |
| **MCGML (*node2vec*)** | 94.23±0.15 | 92.46±0.05 | 82.47±0.08 |

**Discussão:** os resultados da avaliação da tarefa de predição de nós realizada no dataset `Database DSLs`, que compreendem tarefas de classificação *Multi-class* são apresentados

na Tabela 5.5, mostram que a arquitetura MCGML proposta nesta tese é capaz de alcançar desempenho satisfatório no que diz respeito à acurácia das predições em nó, obtendo por exemplo 82.47±0.08 de acurácia no escopo de testes. Isso já responderia a $\mathbf{PP}_2$ onde podemos afirmar que através da arquitetura MCGML foi possível predizer propriedade de nós (categorizando artefatos de metamodelos do domínio de banco de dados). Porém pretendemos expandir essas análises, incorporando à MCGML outros métodos como o GCN (uma rede convolucional em grafo *full-batch*) (Kipf and Welling, 2017), e para GraphSAGE (GraphSAGE *full-batch*) onde utilizamos uma variante média de *pooling*, e conexões de saltos simples para preservar as características dos nós centrais (Hamilton et al., 2017a).

O dataset **`Pro.build DSLs`** é formado por grafos relacionados a metamodelos de criação de uma DSL, descrevendo todos os artefatos presentes em projetos de DSLs envolvendo comandos, sintaxe e semântica. Os nós podem representar `ResourceDescription`, `EDataType`, `EReference`, `EClass`, etc. As arestas entre dois nós indicam como todos esses artefatos se relacionam.

**Tarefa de Predição:** a tarefa que utilizamos nesta avaliação foi predizer quais os 7 tipos de estruturas de DSLs presentes no repositório, onde esses tipos de estruturas foram rotuladas manualmente através da análise dos metamodelos que fazem parte do repositório. Formalmente a tarefa consiste em predizer a categoria primária dos grafos presentes no `Pro.build DSLs`, passando a ser considerado um problema de classificação em 7 classes possíveis.

**Divisão do Dataset (*Dataset splitting*):** assumimos a divisão clássica, onde separamos randomicamente 90% dos nós presentes em `Pro.build DSLs` para fase de treinamento, e os 10% restantes para os testes.

Tabela 5.6: Resultados da tarefa de classificação no dataset `Pro.build DSLs`

| **Método** | **Acurácia (%)** | | |
|---|---|---|---|
| | Treinamento | Validação | Testes |
| **MCGML (*node2vec*)** | 82.57±0.75 | 76.83±0.15 | 74.36±0.17 |

**Discussão:** os resultados da avaliação da tarefa de classificação realizada no dataset `Pro.build DSLs` são apresentados na Tabela 5.6 e mostram que a MCGML utilizando o *node2vec* obteve resultados de acurácia razoáveis pois neste tipo de predição não fazemos uso de informações da estrutura dos grafos, sendo assim é muito provável que se utilizássemos métodos que fazem uso de informações de grafos poderíamos obter melhores resultados nesta tarefa de classificação. Pretendemos expandir os estudos comparativos, incorporando à arquitetura métodos de classificação que utilizam informações de grafos, tais como o GNC e o GraphSAGE. Mesmo com um desempenho razoável, a MCGML obteve 74.36±0.17 de acurácia no escopo de testes. Esse resultado também corroboraria para responder a $\mathbf{PP}_2$ onde observamos que através da arquitetura MCGML foi possível predizer razoavelmente os tipos de estruturas de DSLs presentes no repositório sendo considerado um problema de classificação em 7 classes possíveis.

E para finalizar esta análise das tarefas de ML sobre *embeddings* no nível de nós (*node embeddings*) temos o dataset `Tracker DSLs` onde estão grafos relacionados ao rastreamento de artefatos de metamodelos, detalhando como esses artefatos que compõem um determinado projeto de uma DSL se relacionam. Os nós podem representar qualquer artefato presente em uma DSL, já as arestas indicam qualquer tipo de relacionamento existente entre eles. Este dataset pode ser comparado, por exemplo, a um grafo direcionado representando uma rede de citações bibliográficas entre *papers* indexados pela MAG (*Microsoft Academic Graph*) (Wang et al., 2020).

**Tarefa de Predição:** a tarefa que utilizamos nesta avaliação foi predizer quais os 10 artefatos (os nós) mais referenciados no dataset `Tracker DSLs`, pois essa tarefa torna-se importante para classificar metamodelos por categorias ou tópicos de acordo com as características dos artefatos mais referenciados. Formalmente este problema pode ser formulado como uma tarefa de classificação em 10 classes possíveis.

**Divisão do Dataset (*Dataset splitting*):** assumimos a divisão clássica, onde separamos randomicamente 90% dos nós presentes em `Tracker DSLs` para fase de treinamento, e os 10% restantes para os testes.

Tabela 5.7: Resultados da tarefa de classificação dos nós mais referenciados no dataset `Tracker DSLs`

| **Método** | **Acurácia (%)** | | |
|---|---|---|---|
| | Treinamento | Validação | Testes |
| **MCGML (*node2vec*)** | 78.41±0.83 | 73.36±0.16 | 71.28±0.14 |

**Discussão:** os resultados da avaliação da tarefa de classificação realizada no dataset `Tracker DSLs` são apresentados na Tabela 5.7 e mostram que a MCGML utilizando o *node2vec* obteve resultados de acurácia satisfatórios, novamente porque neste tipo de predição não fazemos uso de informações da estrutura dos grafos, sendo assim poderíamos melhorar esta precisão se utilizássemos métodos que fazem uso de informações de grafos, tais como o GNC e o GraphS. Pretendemos expandir os estudos comparativos, incorporando à arquitetura métodos de classificação que utilizam informações de grafos, tais como o GNC e o GraphSAGE. Com um desempenho um pouco inferior à avaliação realizada sob o dataset `Pro.build DSLs` (Tabela 5.6), a MCGML obteve 71.28±0.14 de acurácia no escopo de testes, reforçando mais uma vez para responder à viabilidade da **PP**$_2$ onde com a MCGML foi possível predizer satisfatoriamente os nós mais referenciados no repositório.

### 5.4.3 PP$_3$: É possível predizer novas conexões a partir daquelas existentes, onde pares de nós, os quais não possuem links, são classificados, e os $K$ principais pares de nós são preditos?

Antes de começarmos a responder à **PP**$_3$ precisamos entender para que servem e como funcionam as tarefas de ML em grafos que trabalham com predições no nível de arestas (*Link-level prediction task*). O objetivo maior das tarefas de predição de links (conexões) é justamente predizer novas conexões a partir daquelas existentes, onde durante os testes, aqueles pares de nós, os quais não possuem links, são classificados, e os $K$ principais pares de nós são preditos. O ponto central é projetar características para um par de nós.

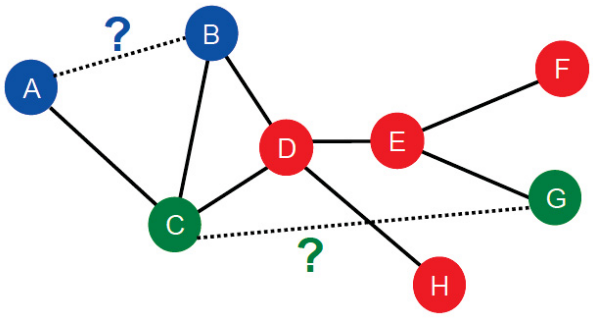



Figura 5.14: Tarefa de predição de links entre nós (Zhang and Chen, 2018)

Basicamente existem duas formulações para a tarefa de predição de links. A primeira baseada na **remoção de links aleatoriamente**, onde removemos um conjunto aleatório de links e, em seguida, tentamos predizê-los. A segunda é baseada em **links ao longo do tempo** onde dado um grafo $G[t_0, t'_0]$ definido por suas arestas até o tempo $t'_0$, a saída será uma lista $L$ contendo a classificação das arestas não contidas em $G[t_0, t'_0]$ ($L \not\subset G[t_0, t'_0]$) as quais são previstas a aparecer no instante $G[t_1, t'_1]$, onde $n = |E_{new}|$ corresponde a quantidade de novas arestas que aparecem durante o período de testes $[t_1, t'_1]$, pegando as $n$ principais arestas contidas em $L$ e contando quantas delas estão corretas. A metodologia da tarefa de predição de links pode ser resumida nas seguintes etapas (You et al., 2022):

- Para cada par de nós $(x, y)$ calculamos uma pontuação (*score*) $c(x, y)$. Onde $c(x, y)$ poderia ser a quantidade de vizinhos em comum entre $x$ e $y$.
- Em seguida ordenamos os pares $(x, y)$ de maneira decrescente em relação às pontuações $c(x, y)$.
- Predizemos os $n$ pares principais como novos links.
- E por último comparamos quais desses links realmente aparecem em $G[t_1, t'_1]$.


(a) Predição de links ao longo do tempo

(b) Verificação de links corretos

Figura 5.15: Metodologia da tarefa de predição de links (Zhang and Chen, 2018)

Apresentamos como definimos e organizamos nosso *benchmark* de grafos na Seção 5.4.2, e continuaremos retomando à análise da $\mathbf{PP}_3$ que trata da possibilidade de realizarmos predições de propriedades no nível de arestas (*link-level prediction*) através da MCGML. Na avaliação experimental relacionada à tarefa de predição de links (arestas) também separamos 3 datasets extraídos a partir de 3 categorias de repositórios de metamodelos (`Petrinet DSLs`, `Bibliography DSLs`, e `Review sys.DSLs` - Tabela 5.4). Onde `Petrinet DSLs` representa o dataset dos metamodelos em grafos relacionados às DSLs relacionadas a especificações de redes de Petri, no `Bibliography DSLs` estão grafos relacionados a metamodelos de criação de DSLs de referências bibliográficas, e no dataset `Review sys.DSLs` estão grafos relacionados a artefatos de metamodelos de testes de software (Önder Babur, 2019). Lembrando que todos esses datasets possuem diferentes escalas, tarefas, e incluem grafos homogêneos e heterogêneos.

As estatísticas dos grafos presentes nesses datasets podem ser visualizadas na Tabela 5.4. Podemos observar que em matéria de densidade, os grafos do `Review sys.DSLs` são mais densos em comparação aos grafos do `Petrinet DSLs` e do `Bibliography DSLs` pois os grafos do `Review sys.DSLs` possuem a maior média dos graus dos nós, a menor quantidade de nós em comparação com os demais, e um diâmetro médio relativamente baixo. Já os grafos do `Bibliography DSLs` têm uma estrutura de seus grafos mais clusterizada em comparação aos demais datasets, isso pode ser observado pela alta média dos coeficientes de clusterização. O **baseline** dos experimentos realizados para responder a $\mathbf{PP}_3$ está também pautado na instanciação que fizemos do *node2vec* utilizando como entrada tipos de arestas simples e os *node embeddings* (Grover and Leskovec, 2016). Durante os treinamentos, separamos amostras de arestas aleatórias e as utilizamos como exemplos negativos; usamos o mesmo número de arestas negativas e positivas.

O **`Petrinet DSLs`** dataset é formado por grafos direcionados que representam metamodelos relacionados à área de modelagem de sistemas distribuídos, possuindo nós de posição, nós de transição, e arcos direcionados conectando posições com transições. Onde os nós no grafo indicam os nós de posição, e as arestas representam os arcos direcionados que conectam posições com transições.

**Tarefa de Predição:** a tarefa que utilizamos nesta avaliação consiste em predizer novas associações de arestas a partir das arestas de treinamento. Esta avaliação é baseada em quão bem o modelo de ML em grafos presente na MCGML classifica arestas de teste positivas sobre arestas de teste negativas. Para isso, classificamos cada aresta positiva presente no conjunto de validação/testes em relação a uma amostra aleatória de 50.000 arestas negativas, e depois contabilizamos a proporção de arestas positivas que estão classificadas entre a $K$-ésima posição ou acima (HITS-$K$). Consideramos para esta avaliação um $K = 50$ como limite para avaliar o desempenho do modelo presente na MCGML (*node2vec*). Essa métrica é mais desafiadora em comparação ao ROC-AUC porque o modelo precisa classificar consistentemente as arestas positivas em posições mais altas do que quase todas as arestas negativas.

**Divisão do Dataset (*Dataset splitting*):** dividimos as arestas dentro de três conjuntos (Treinamento/Validação/Testes). As arestas de treinamento são formadas pelos arcos com os maiores pesos (sendo consideradas as arestas mais importantes do grafo). Em contrapartida, as arestas de validação/testes contêm associações de posições com transições com pesos menores. O objetivo é predizer um tipo particular de associação de posições e transições a partir de outros tipos associações que podem ser mais facilmente medidas e são conhecidas por se correlacionarem com aquelas associações que nos interessam.

Tabela 5.8: Resultados da tarefa de predição de arestas no dataset `Petrinet DSLs`

| **Método** | **HITS-50** | | |
|---|---|---|---|
| | Treinamento | Validação | Testes |
| **MCGML (*node2vec*)** | 35.62±0.73 | 30.53±0.81 | 30.27±0.78 |

**Discussão:** os resultados da avaliação da tarefa de predição de arestas realizada no dataset `Petrinet DSLs` são apresentados na Tabela 5.8, onde a MCGML apresentou falha para *overfit* no conjunto de arestas de treinamento e alcançou desempenhos similares entre os conjuntos de Treinamento/Validação/Testes, isso pode ter ocorrido em virtude da ausência de ricas características de nós para este dataset. Além disso, o mal desempenho nos treinamentos sugere que a informação posicional a qual não conseguimos capturar apenas com o *node2vec* pode ser crucial para ajustar as arestas de treinamento e obter *node embeddings* significativos. Esse fraco desempenho na generalização nos motiva a buscar o desenvolvimento de novas ideias de pesquisa

para superar este *gap*, por exemplo, buscando uma maneira de inserir informações posicionais na instanciação feita do *node2vec*, ou utilizando outros modelos como as GNNs (*Graph Neural Networks*), ou ainda desenvolvendo técnicas de amostragem negativas mais elaboradas. Esse resultado não corrobora para responder a $\mathbf{PP}_3$ onde, para este dataset, observamos uma dificuldade da arquitetura MCGML em predizer se existem links desconexos entre dois nós (por exemplo: analisar se existem artefatos de metamodelos vagos sem associação).

O dataset **`Bibliography DSLs`** possui grafos relacionados a metamodelos de criação de DSLs de referências bibliográficas, como o clássico BibTex, descrevendo todos os artefatos presentes em projetos de DSLs envolvendo o domínio de referências bibliográficas, onde os nós referentes a qualquer tipo de publicação possui conexões (arestas) a diversos outros nós (atributos) que podem representar `Title`, `Author`, `Journal`, `Volume`, `Year`, etc., e todas as arestas neste dataset estão associadas a dois tipos de meta-informação: o ano, e o peso da aresta representando a quantidade, por exemplo, de co-autores que publicaram obras em um determinado ano. Os grafos presentes neste dataset podem ser vistos como grafos multi-dinâmicos uma vez que podem possuir múltiplas arestas entre dois nós se há colaboração entre eles por mais de um ano.

**Tarefa de Predição:** basicamente a tarefa que utilizamos nesta avaliação consiste em predizer os relacionamentos colaborativos entre autores em um determinado ano a partir de colaborações passadas. Como esta tarefa é um problema de série temporal, é natural que os modelos de ML em grafos incorporem as informações de arestas mais recentes para realizar a tarefa de predição, por exemplo, usando arestas de validação quando estiver realizando os teste de predição de arestas. Porém o modelo presente no *node2vec* não apresenta este recurso (informações de arestas mais recentes), o que poderia ser um fator negativo. A métrica de avaliação utilizada é similar à métrica aplicada ao dataset anterior `Petrinet DSLs`, onde gostaríamos que o modelo de ML da MCGML fosse capaz de classificar colaborações verdadeiras acima das colaborações falsas. Para isso, classificamos cada colaboração verdadeira entre um conjunto de 20.000 colaborações negativas sendo amostras aleatórias, e contabilizamos a proporção de arestas positivas que estão classificadas entre a $K$-ésima posição ou acima (HITS-$K$). Consideramos para esta avaliação um $K = 20$ como um bom limite em nossos experimentos preliminares para avaliar o desempenho do modelo presente na MCGML (*node2vec*).

**Divisão do Dataset (*Dataset splitting*):** dividimos os dados por tempo, na tentativa de simular uma aplicação realista de recomendação de colaboração. Neste caso, separamos as arestas que representam colaborações entre os anos de 2015 à 2017 como arestas de treinamento, e as correspondentes ao período entre os anos de 2018 e 2019 como arestas de validação e as de 2020 como arestas de teste.

Tabela 5.9: Resultados da tarefa de predição de arestas no dataset `Bibliography DSLs`

| **Método** | **HITS-20** | | |
|---|---|---|---|
| | Treinamento | Validação | Testes |
| **MCGML (*node2vec*)** | 96.54±0.31 | 53.48±0.56 | 44.83±0.48 |

**Discussão:** os resultados da avaliação da tarefa de predição de arestas realizada no dataset `Bibliography DSLs` são apresentados na Tabela 5.9, onde o modelo da MCGML considera a configuração convencional em que as arestas de validação são usadas apenas para a seleção. Para estas configurações a MCGML alcançou bons resultados. Isso pode ser explicado pelo fato de que as informações posicionais, ou seja, aquelas referentes a colaborações passadas, são um recurso muito mais valioso para predizer colaborações futuras em comparação a apenas confiar na média das representações de publicações dos autores, ou seja, no mesmo interesse de

pesquisa. Observando os índices obtidos nas etapas de Validação e Testes suspeitamos que o algoritmo da MCGML entrou em *overfitting* para o dataset `Bibliography DSLs`, ou seja, ele passou à não mais predizer, mas a chutar valores. Uma maneira de tentar melhorar o desempenho nas fases de validação e testes seria inserindo informações posicionais na nossa versão do *node2vec*, ou utilizando outros modelos de ML como as GNNs (*Graph Neural Networks*), mas precisaríamos elucidar a dúvida: será que os dados de treinamento estão livres de ruídos? Isso provavelmente levantaria um questionamento quanto à correta classificação feita pelos autores originais do dataset (Önder Babur, 2019). Este dataset fornece uma oportunidade de pesquisa para grafos multi-dinâmicos. Para explorar o potencial benefício da modelagem de séries temporais, podemos expandir os experimentos para os modelos GCN e GraphSAGE, incorporamos as arestas mais recentes (ou seja, as arestas de validação) como entrada para estes modelos. Os resultados obtidos com este dataset ajudam a responder a $\mathbf{PP}_3$ demonstrando que a arquitetura MCGML apresentou certa dificuldade em predizer novas conexões a partir daquelas existentes, onde pares de nós, os quais não possuem links, são classificados, e os $K$ principais pares de nós são preditos, alcançando uma regular proporção de arestas positivas que são classificadas entre a $K$-ésima posição ou acima (HITS-$K$), para um $K = 20$ (HITS-20), em virtude do possível *overfitting* ocasionando a geração de predições aleatórias.

E para finalizar as avaliações sobre predição de arestas, o dataset **`Review sys.DSLs`** é composto por grafos direcionados e homogêneos relacionados a artefatos de metamodelos de testes de software, onde cada nó representa um artefato de teste ou uma etapa do processo de teste de software. As arestas representam as interações entre todos esses artefatos, e podem ser interpretadas como o fenômeno do efeito de aplicarmos em conjunto um ou mais artefatos sobre o processo de teste de software, sendo consideravelmente diferente do efeito da aplicação isolada de um determinado artefato, independente uns dos outros.

**Tarefa de Predição:** para este dataset, a tarefa que utilizamos nesta avaliação consiste em predizer interações, artefato-artefato, artefato-processo, ou processo-processo, a partir de informações já conhecidas das interações entre esses elementos. A métrica de avaliação utilizada é similar à métrica aplicada ao dataset anterior `Bibliography DSLs`, onde gostaríamos que o modelo de ML da MCGML fosse capaz de classificar interações verdadeiras em uma proporção maior em comparação às interações falsas. Para isso, classificamos cada interação verdadeira entre um conjunto de 10.000 colaborações negativas sendo amostras aleatórias, e contabilizamos a proporção de arestas positivas que estão classificadas entre a $K$-ésima posição ou acima (HITS-$K$). Consideramos para esta avaliação um $K = 10$ como um bom limite em nossos experimentos preliminares para avaliar o desempenho do modelo presente na MCGML (*node2vec*).

**Divisão do Dataset (*Dataset splitting*):** para este experimento, optamos por uma divisão baseada na interação alvo, o que significa que dividimos as arestas dos artefatos de teste de software de acordo com os efeitos desses artefatos no processo de teste. Como resultado, nosso conjunto de teste foi formado por aqueles artefatos que interagem predominantemente a artefatos diferentes daqueles presentes nos conjuntos de treinamento e validação. Isso significa que os artefatos do conjunto de teste provocam efeitos diferentes no processo de teste de software em comparação aos artefatos presentes nos conjuntos de treinamento e validação. Com esta divisão baseada na interação alvo foi possível avaliar até que ponto o modelo de ML utilizado na MCGML é capaz de realizar predições úteis, ou seja, predições não triviais, aquelas que não são prejudicadas pela suposição de que existem artefatos conhecidos e muito semelhantes no conjunto de treinamento.

**Discussão:** os resultados da avaliação da tarefa de predição de arestas realizada no dataset `Review sys.DSLs` são apresentados na Tabela 5.10, e o modelo de ML da MCGML

Tabela 5.10: Resultados da tarefa de predição de interações alvo no dataset `Review sys.DSLs`

| **Método** | **HITS-10** | | |
|---|---|---|---|
| | Treinamento | Validação | Testes |
| **MCGML (*node2vec*)** | 38.72±1.13 | 33.87±1.05 | 24.37±1.85 |

apresentou resultados regulares, mostrando uma variação considerável entre a proporção alcançada com os conjuntos de Treinamento/Validação em relação à proporção com o conjunto de testes. Essa variação pode ser parcialmente atribuída à densidade dos grafos e o desafio da divisão desse dataset, onde a divisão baseada nas interações alvo é mais complexa e desafiadora em relação à divisão aleatória convencional de arestas. No geral, o maior desafio que avaliamos para este dataset `Review sys.DSLs` está em predizer arestas (*links*) fora da distribuição, em grafos com uma certa densidade. Esse desempenho regular na generalização nos motiva a buscar alternativas para superar este *gap*, por exemplo, utilizando outros modelos de ML como as GNNs (*Graph Neural Networks*) e o GCN. Mesmo esses resultados não corroborando para responder a **PP**$_3$, observamos possibilidades na arquitetura MCGML em melhorar as predições em nível de arestas através da incorporação de outros modelos de ML além da instanciação do *node2vec*, mas para isso precisaríamos explorar mais datasets para cada um dos grupos de `DSLs`.

### 5.4.4 PP$_4$: É possível predizer propriedades relacionadas a grafos inteiros ou subgrafos, permitindo realizar classificações binárias de documentos de acordo com metamodelos através da classificação de grafos?

Apresentamos nosso *benchmark* de grafos na Seção 5.4.2, e para realizarmos as avaliações e análises da **PP**$_4$ que trata da possibilidade de realizarmos predições de propriedades no nível de grafo (*graph-level prediction*) através da MCGML. Na avaliação experimental relacionada à tarefa de predição de propriedades do grafo inteiro ou de subgrafos também separamos 3 datasets extraídos a partir de 3 categorias de repositórios de metamodelos (`Office tools DSLs`, `State mac.DSLs`, e `Req.spec.DSLs` - Tabela 5.4). Onde `Office tools DSLs` representa o dataset dos metamodelos em grafos relacionados às DSLs de especificações de ferramentas *Office*, no dataset `State mac.DSLs` estão grafos relacionados a metamodelos de criação de modelos de máquinas de estado, e no dataset `Req.spec.DSLs` estão grafos relacionados a artefatos de metamodelos para as DSLs voltadas a especificações de requisitos (Önder Babur, 2019). Lembrando que todos esses datasets possuem diferentes escalas, tarefas, e incluem grafos homogêneos e heterogêneos.

As estatísticas dos grafos presentes nesses datasets podem ser visualizadas na Tabela 5.4. Podemos observar que em matéria de densidade, os grafos do `State mac.DSLs` são mais densos em comparação aos grafos do `Office tools DSLs` e do `Req.spec.DSLs` pois os grafos do `State mac.DSLs` possuem a maior média dos graus dos nós (`46.7`), a maior quantidade de nós em comparação com os demais, e um diâmetro médio mediano em relação aos outros dois. Já os grafos do `Req.spec.DSLs` têm uma estrutura de seus grafos mais clusterizada em comparação aos demais datasets, isso pode ser observado pela alta média dos coeficientes de clusterização (`0.842`), podemos afirmar também que os grafos presentes no `Req.spec.DSLs` são relativamente densos, com uma média dos graus dos nós de `38.9`. O **baseline** dos experimentos realizados para responder a **PP**$_4$ também está pautado na instanciação que fizemos do *node2vec* utilizando como entrada os *node embeddings*, os quais são agrupados para obtermos o *embedding* do grafo inteiro (*graph embedding*), e em seguida aplicamos um modelo linear sob o *graph embedding* para realizarmos as predições correspondentes (Grover and Leskovec, 2016).

O **`Office tools DSLs`** dataset é formado por grafos de metamodelos relacionados às DSLs de especificações de ferramentas *Office*, onde cada grafo representa um modelo de ferramenta *Office*, onde cada nó representa um artefato que compõe a ferramenta, e as arestas fazem referências às conexões (aos relacionamentos) entre esses artefatos.

**Tarefa de Predição:** a tarefa que utilizamos nesta avaliação consiste em predizer as propriedades da ferramenta *Office* alvo com a maior precisão possível, onde as propriedades da ferramenta são vistas como rótulos binários, por exemplo, se uma ferramenta realiza ou não a conversão de arquivos de um formato para outro (*.doc para *.pdf, por exemplo). A métrica utilizada nesta avaliação seguiu as sugestões do trabalho de pesquisa feito por Hao et al. (2021), onde aplicamos a métrica ROC-AUC para a avaliação. A métrica ROC (*Receiver Operator Characteristic*) representa o relacionamento entre as taxas $FPR$ (*False Positive Rate*) e $TPR$ (*True Positive Rate*), conforme detalhado na Seção 5.2.4.5.

**Divisão do Dataset (*Dataset splitting*):** assumimos a divisão clássica, onde separamos randomicamente 90% dos modelos de ferramentas presentes no `Office tools DSLs` para fase de treinamento, e os 10% restantes para os testes. Onde listamos 5 propriedades das ferramentas como rótulos binários (se possuem ou não possuem), e analisamos o índice ROC-AUC para verificar o nível de precisão nas predições de propriedades de grafos.

Tabela 5.11: Resultados da tarefa de predição de propriedades no dataset `Office tools DSLs`

| **Método** | **ROC-AUC(%)** | | |
|---|---|---|---|
| | Treinamento | Validação | Testes |
| **MCGML (*node2vec*)** | 85.43±0.96 | 79.43±0.71 | 70.68±1.13 |

**Discussão:** os resultados da avaliação da tarefa de predição de propriedades de grafo realizada no dataset `Office tools DSLs`, que compreendem tarefas de classificação binária *Binary-class* são apresentados na Tabela 5.11, mostram que a arquitetura MCGML é capaz de alcançar desempenho relativamente satisfatório no que diz respeito a este tipo de classificação, obtendo por exemplo `70.68±1.13` de índice ROC-AUC no escopo de testes (apresentando um leve *overfitting*). Isso já responderia a $\mathbf{PP}_4$ onde podemos afirmar que através da arquitetura MCGML é possível predizer propriedades de grafos (se uma ferramenta possui ou não uma determinada propriedade). Pretendemos expandir essas análises para outros métodos, como por exemplo para a GCN (rede convolucional em grafo *full-batch*) (Kipf and Welling, 2017).

O dataset **`State mac.DSLs`** é formado por grafos de metamodelos voltados à criação de modelos de máquinas de estado, que mapeiam o comportamento de artefatos (objetos) de um sistema entre várias ocorrências de transições comportamentais possíveis ao longo de um processo. Onde cada nó representa um artefato (objeto) presente em um determinado processo, e as arestas fazem referências às transições comportamentais entre os estados possíveis.

**Tarefa de Predição:** a tarefa que utilizamos nesta avaliação consiste em predizer as propriedades de uma máquina de estados alvo (*State machine target*) com a maior precisão possível, onde essas propriedades novamente são vistas como rótulos binários, por exemplo, se um diagrama de máquina de estado realiza ou não uma determinada transição (aberto para fechado, por exemplo). Aplicamos mais uma vez, para esta avaliação, a métrica ROC-AUC (Hao et al., 2021).

**Divisão do Dataset (*Dataset splitting*):** nesta avaliação seguimos novamente a divisão clássica, onde separamos randomicamente 90% dos modelos de *State machine* presentes no `State mac.DSLs` para a fase de treinamento, e os 10% restantes para os testes. Dessa vez listamos 10 propriedades das máquinas de estado como rótulos binários (se possuem ou não

possuem), e analisamos o índice ROC-AUC para verificar o nível de precisão nas predições de propriedades de grafos.

Tabela 5.12: Resultados da tarefa de predição de propriedades no dataset `State mac.DSLs`

| **Método** | **ROC-AUC(%)** | | |
|---|---|---|---|
| | Treinamento | Validação | Testes |
| **MCGML (*node2vec*)** | 80.82±0.76 | 75.16±0.68 | 68.56±0.92 |

**Discussão:** os resultados da avaliação da tarefa de predição de propriedades de grafo realizada no dataset `State mac.DSLs`, que compreendem tarefas de classificação binária *Binary-class* são apresentados na Tabela 5.12, mostram novamente que a arquitetura MCGML é capaz de alcançar um desempenho razoável no que diz respeito a este tipo de classificação, obtendo `68.56±0.92` de índice ROC-AUC no escopo de testes, um pouco inferior em relação à avaliação realizada sobre o dataset `Office tools DSLs`, mas acreditamos que esta queda de performance esteja relacionada ao aumento da quantidades de propriedades que listamos para esta avaliação (10 contra 5 do `Office tools DSLs`). Os resultados obtidos corroboram para responder a $\mathbf{PP}_4$ tendo a MCGML a possibilidade de predizer propriedades de grafos (se um modelo de máquina de estado possui ou não uma determinada propriedade). Pretendemos expandir também esta avaliação para o método GCN.

E finalizando nossa série de avaliações, analisando as possibilidades de realizarmos predições de propriedades no nível de grafo (*graph-level prediction*) através da MCGML, temos o dataset **`Req.spec.DSLs`** onde estão grafos relacionados a artefatos de metamodelos para as DSLs voltadas a especificações de requisitos, relacionados ao rastreamento de requisitos em um determinado domínio de negócio, detalhando como os requisitos se relacionam. Os nós podem representar qualquer artefato que represente qualquer tipo de requisito, e as arestas indicam qualquer tipo de relacionamento (conexão) existente entre requisitos.

**Tarefa de Predição:** a tarefa que utilizamos nesta avaliação consiste em predizer as propriedades (se um determinado requisito está ou não presente em um determinado domínio) com a maior precisão possível, onde essas propriedades novamente são vistas como rótulos binários, por exemplo, se um domínio apresenta ou não um requisito qualquer. Utilizamos mais uma vez a métrica ROC-AUC para esta avaliação.

**Divisão do Dataset (*Dataset splitting*):** nesta avaliação seguimos novamente a divisão clássica, onde separamos randomicamente 90% de requisitos presentes no `Req.spec.DSLs` para a fase de treinamento, e os 10% restantes para os testes. Dessa vez listamos 20 requisitos como rótulos binários (se possuem ou não possuem), e analisamos o índice ROC-AUC para verificar o nível de precisão nas predições de propriedades de grafos.

Tabela 5.13: Resultados da tarefa de predição de propriedades no dataset `Req.spec.DSLs`

| **Método** | **ROC-AUC(%)** | | |
|---|---|---|---|
| | Treinamento | Validação | Testes |
| **MCGML (*node2vec*)** | 82.38±1.07 | 78.74±0.76 | 71.63±0.84 |

**Discussão:** os resultados da avaliação da tarefa de predição de propriedades de grafo realizada no dataset `Req.spec.DSLs`, que compreendem tarefas de classificação binária *Binary-class* são apresentados na Tabela 5.13, mostram mais uma vez que a arquitetura MCGML é capaz de alcançar um bom desempenho neste tipo de classificação, obtendo `71.63±0.84` de índice ROC-AUC no escopo de testes, um pouco superior em relação à avaliação realizada sobre o dataset `State mac.DSLs`, essa surpresa de superioridade acreditamos que esteja relacionada

à simplicidade na determinação da presença de requisitos. Esses resultados reforçam a resposta para a $\mathbf{PP}_4$ onde a MCGML demonstra sua capacidade de predizer propriedades de grafos (se um determinado requisito está ou não presente em um determinado domínio). Novamente há a possibilidade de expansão dessas avaliações utilizando o método GCN.

## 5.5 CONCLUSÃO

Neste capítulo apresentamos em detalhes todo o processo de avaliação experimental da arquitetura MCGML proposta nesta tese, onde, a partir de 4 perguntas de pesquisa, conseguimos demonstrar a viabilidade da arquitetura proposta, com grandes perspectivas de melhorias e expansões visando a incorporação de outros modelos de ML. No capítulo seguinte apresentaremos as principais discussões relacionadas às lições aprendidas com os experimentos realizados, bem como as perspectivas futuras de expansão da arquitetura MCGML proposta.

# 6 DISCUSSÕES GERAIS E CONTRIBUIÇÕES DE PESQUISA

Neste capítulo apresentaremos uma visão geral das contribuições obtidas ao longo do processo de pesquisa, e algumas discussões relacionadas aos resultados obtidos nos experimentos de avaliação de viabilidade da arquitetura MCGML, discutiremos também alguns trabalhos relacionados que tratam do problema de classificação de metamodelos apresentados no Capítulo 3, as publicações originadas desta tese, as lições aprendidas, bem como apresentaremos uma visão sobre as possibilidades de desenvolvimento de trabalhos futuros.

## 6.1 O INÍCIO: CLASSIFICAÇÃO DE MODELOS ATRAVÉS DE REDES NEURAIS MLP

O primeiro passo importante que obtivemos durante esta pesquisa foi contribuir para a classificação de modelos desestruturados, apresentando uma metodologia para analisar e classificar documentos JSON, tendo como base metamodelos existentes. Fizemos isso extraindo metamodelos JSON, utilizando uma solução *One-hot encoding*, para uma rede neural *Multi-Layer Perceptron* (MLP), onde traduzimos os elementos do metamodelo em neurônios de entrada da rede. Treinamos nossa rede neural MLP e então a utilizamos para classificar documentos JSON, os quais foram traduzidos em dados de entrada para serem classificados (Seção 4.1). Ao final deste processo, conduzimos uma série de experimentos utilizando MLPs com quantidades diferentes de camadas intermediárias, demonstrando que esta primeira abordagem se mostrava efetiva na classificação de documentos (Couto. et al., 2020). O fluxo de execução para a classificação de modelos desestruturados (descrito na Figura 4.2) serviu de inspiração para o desenho da arquitetura MCGML que viria a seguir.

## 6.2 *ENCODING*: O PROCESSO DE CONVERSÃO DE METAMODELOS EM GRAFOS

Após obtermos bons resultados no processo de classificação de modelos desestruturados através de uma rede neural MLP, continuamos expandindo a pesquisa, e chegamos na idealização da arquitetura MCGML (Figura 4.5). O primeiro desafio seria converter metamodelos em grafos, e pensando nisso criamos o Algoritmo 3, utilizando as bibliotecas Python SNAP, *gnuplot*, e *graphviz*. O processo de *encoding* dos metamodelos em grafos começava com a leitura do metamodelo $\mathcal{M}$, sendo $\mathcal{M}$ composto de classes, atributos e referências, que representam uma coleção de pares chave/valor (*key/value*) no formato JSON, e criamos o grafo do metamodelo através da leitura de cada par chave/valor do metamodelo JSON, e realizando a adição de vértices e arestas ao grafo *Snap TNGraph*.

Com a conclusão dessa etapa, tínhamos a possibilidade de realizar a manipulação do grafo, calcular suas propriedades estruturais com o algoritmo *node2vec*, e realizar análises de similaridades através das caminhadas aleatórias em largura e profundidade. Dávamos assim mais um importante passo na construção da arquitetura MCGML, porém era preciso explorar e adaptar o algoritmo *node2vec* as nossas necessidades, entrávamos em uma etapa crucial para a viabilidade da MCGML.

## 6.3 INSTANCIANDO O ALGORITMO *NODE2VEC*

Após a conclusão do processo de *encoding*: convertendo metamodelos JSON em grafos, avançamos os estudos na área de *Machine Learning* (ML) em grafos, tendo como principal desafio encontrar uma forma de representar ou codificar informações sob a forma de estruturas em grafos que possam ser facilmente exploradas pelos modelos de *machine learning*. Na seção 3.4.1 apresentamos o *node2vec*: um *framework* algorítmico para aprender de forma contínua a representar características dos vértices (nodos) de um grafo. Iniciando assim mais uma etapa na construção da MCGML, com o desafio de instanciar o algoritmo *node2vec* para as nossas necessidades. Com o processo de extração do metamodelo JSON para grafo concluído, passamos a estudar os recursos do algoritmo *node2vec* na tentativa de explorar sua capacidade de realizar predições e análises de similaridades entre diferentes vértices em um grafo, visando obter um espaço vetorial de baixa dimensionalidade (*embedding*) que servisse de recurso de entrada para os algoritmos de ML em grafos. Assim demos mais uma contribuição para a pesquisa, criando o Algoritmo 4, instanciando o *node2vec* para explorar os seus recursos no grafo do metamodelo, tendo como entrada o grafo do metamodelo JSON extraído no processo de *encoding* descrito na seção anterior. Um dos principais desafios dessa etapa foi entender e ajustar os parâmetros do *node2vec* para o nosso problema de pesquisa, em um processo que exigiu algumas tentativas de entendimento e adaptações de parâmetros, conseguindo assim a produção dos dois principais arquivos desta etapa: um que guarda as análises de inferências de acordo com as similaridades encontradas ($arquivoInferenciasSimilaridades$) consistindo no processo de *embedding* e na geração do vetor numérico de baixa dimensionalidade, e o segundo ($arquivoModeloInferencias$) que guarda o resultado do modelo de inferência obtido após realizar todas as caminhadas aleatórias tendenciosas, com os respectivos pesos atribuídos para cada passo na caminhada aleatória (o processo foi descrito na Seção 4.2.1). Chegávamos ao ponto chave da arquitetura MCGML: a exploração dos níveis de *embeddings*.

## 6.4 OS *EMBEDDINGS*

Com a conclusão da etapa de instanciação do algoritmo *node2vec* dávamos início a etapa de exploração dos recursos da área da ciência de dados em grafos (*Graph Data Science*) e a sua capacidade de fazer predições em diversos cenários. Através dos *embeddings* de grafos aprendemos a estrutura do grafo, ao invés de depender de fórmulas predeterminadas para calcular características específicas, possibilitando o aprendizado de máquina em grafos para explorar toda a capacidade da análise preditiva. Por exemplo: dado um grafo qualquer, a partir da instanciação que fizemos do algoritmo *node2vec* tivemos a capacidade de aprender a representar características de forma contínua (um vetor de números) para cada nó do grafo, e esse vetor de características utilizamos para várias tarefas de aprendizado de máquina, como as classificações e predições de arestas. O processo de *embedding* em grafo é o ponto chave da arquitetura MCGML, pois através dos *embeddings* conseguimos transformar a topologia e as características de um determinado grafo em um vetor de tamanho fixo representando unicamente cada nó do grafo. Dessa forma, podemos preservar as características chaves do grafo enquanto reduzimos a dimensionalidade de uma forma que podemos decodificá-la para usá-las nas tarefas de predições de ML que realizamos em nossas avaliações experimentais.

Os *embeddings* obtidos utilizam representações no nível de grafos ou no nível de nós (nodos) para definir funções de similaridades ou predizer scores (pontuações) de similaridades. Por exemplo: dada uma coleção de grafos, o processo de *embedding* converte cada grafo $G$ para um espaço $d-$dimensional ($d \ll \|V\|$), onde o grafo é representado ou como um conjunto de

vetores $d-$dimensional onde cada vetor representa o *embedding* de um nó (ou seja, um *embedding* no nível de nó (nodo) (*node-level representations*)), ou como um vetor $d-$dimensional para o grafo inteiro como um *embedding* no nível de grafo (*graph-level representations*). Os *embeddings* no nível de nó comparam os grafos usando representações no nível de nó aprendidas a partir dos grafos. Os scores de similaridades obtidos por esses métodos capturam principalmente as similaridades entre nós correspondentes em dois grafos distintos, dando prioridade nas informações no nível de nó durante o processo de aprendizado. Já os *embeddings* no nível de grafo aprendem uma representação vetorial para cada grafo, e então aprendem a calcular os scores de similaridades entre grafos com base nessa representação vetorial. Nesta tese utilizamos ambos tipos de *embedding* em nossas avaliações experimentais, demonstrando a viabilidade da arquitetura proposta.

## 6.5 ANÁLISE DAS AVALIAÇÕES EXPERIMENTAIS

Na Seção 5.1.2 apresentamos os repositórios de modelos e metamodelos que serviram como *baseline* para as avaliações experimentais da MCGML (Önder Babur, 2019). Na Seção 5.1.3 discutimos sobre a importância das tarefas de clusterização e classificação de metamodelos, chamando a atenção para a necessidade de encontrarmos soluções adequadas para automatizar o processo de clusterização de repositórios de metamodelos, por meios de técnicas avançadas e ferramentas para organizar automaticamente os artefatos de modelagem, especialmente quando metamodelos são adicionados ou alterados com frequência (Torres et al., 2021). Vimos que a possibilidade de aplicar técnicas de clusterização tem sido estudada na tentativa de organizar metamodelos reutilizáveis automaticamente, e isso nos motivou a contribuir para mitigar esses problemas, servindo de inspirações para o desenho da MCGML.

Após a apresentação de alguns conceitos e métricas importantes relacionadas às análises em grafos na Seção 5.1.4, apresentamos os algoritmos GBDT e SVM na Seção 5.1.5 que utilizamos em nosso experimento para respondermos nossa primeira pergunta de pesquisa ($\mathbf{PP}_1$) relacionada a uma avaliação comparativa dos benefícios das tarefas de classificação e clusterização em grafos presentes na arquitetura MCGML com os algoritmos GBDT e SVM. Em relação à $\mathbf{PP}_1$ (Seção 5.4.1), consideramos este o principal experimento para credibilizar a viabilidade da MCGML para realizar tarefas de classificação. Nos resultados apresentados na Tabela 5.2 calculamos a Acurácia (*Accuracy*), a Precisão ($P$), o *Recall* ($R$), e o *F1-score* ($F1$) para cada configuração de teste nas tarefas de classificação multi-classes. Em quase todos os cenários a MCGML obteve desempenho superior em comparação com os algoritmos GBDT e SVM, com destaque para a Acurácia, onde, na grande maioria das comparações, a MCGML obteve melhor precisão em comparação com GBDT e SVM. Por exemplo, na tarefa de classificação binária, com $c = 2$, a MCGML obteve 93.20% de Acurácia, enquanto que os percentuais de Acurácia obtidos pelos algoritmos GBDT e SVM foram de 91.05% e 72.15% respectivamente. Mas realizando tarefas de classificação multi-classes, onde $c = 6$, a MCGML obteve uma leve piora em sua Acurácia, obtendo 92.23%, sendo superada pelo GBDT que obteve 93.40%, porém superando o SVM que ficou com 79.56% de Acurácia. Acreditamos que essa leve queda de precisão na tarefa de classificação multi-classes (com $c = 6$ - seis categorias de metamodelos - Figura 5.11(a)) se deve a proximidade em semelhança entre os artefatos contidos entre as categorias de metamodelos. No contexto geral e com base nos demais indicadores, conseguimos responder à $\mathbf{PP}_1$ de maneira satisfatória, reforçando a viabilidade da arquitetura proposta.

Nas perguntas de pesquisa $PP_2$, $PP_3$ e $PP_4$ (apresentadas na Seção 5.2.1) exploramos as tarefas de predição e análise de similaridades em grafos, sendo os destaques específicos de nosso estudo. Na $\mathbf{PP}_2$ (Seção 5.4.2) exploramos a tarefa de predição de propriedades de um nó (por

exemplo: categorizar artefatos de metamodelos) utilizando os métodos de classificação de nós (nodos), para isso utilizamos os recursos de *embedding* no nível de nó para realizar predições baseadas na sequência de artefatos que compõem os metamodelos. Antes de entrarmos nas avaliações que corroboraram para responder à $\mathbf{PP}_2$ (Seção 5.4.2), explicamos como organizamos nosso *benchmark* de grafos baseado nos datasets indexados de metamodelos presentes nos repositórios relacionados a linguagens de domínio específico (*domain-specific languages* - DSLs), nestes datasets dispomos de 555 metamodelos distribuídos em 9 categorias, seguindo o trabalho realizado por Önder Babur (2019), apresentado na Seção 5.2.2. As Tabelas 5.3 e 5.4 detalham a distribuição dos datasets que compõem o *benchmak*, e as características estatísticas dos grafos que compõem o *benchmak* respectivamente. Na $\mathbf{PP}_2$ tratamos da possibilidade de realizarmos predições de propriedades de um nó (*node-level prediction*) através da MCGML. E para solucionarmos a tarefa de predição de propriedades de um nó (e responder a $\mathbf{PP}_2$) dado um determinado grafo $G = (V, E)$, foi preciso aprender uma função $f : V \rightarrow \mathbb{R}$, onde essa função de aprendizado $f$ foi obtida pelos *embeddings* dos grafos. Nesta avaliação experimental relacionadas à tarefa de predição de propriedades de um nó utilizamos 3 datasets extraídos a partir de 3 categorias de repositórios de metamodelos (`Database DSLs`, `Pro.build DSLs`, e `Tracker DSLs` - Tabela 5.4). E para cada repositório fizemos análise de Acurácia para analisarmos o nível de precisão da MCGML (que utiliza uma instanciação do algoritmo *node2vec*) nas tarefas de predição de propriedades de nós. Com base nas Tabelas 5.5, 5.6 e 5.7 discutimos os resultados de cada uma delas com base nas tarefas de predição definidas em cada experimento. Dessa forma, observamos que o nível de Acurácia variou em virtude das particularidades de cada repositório de metamodelos. Porém em todos eles, a MCGML obteve uma Acurácia acima de 70% nos cenários de testes, o que podemos considerar como satisfatórios (contribuindo para responder à $\mathbf{PP}_2$). Sabemos que podemos enriquecer a arquitetura com outros métodos de predição, como por exemplo aqueles baseados em redes neurais em grafos - as GNNs, e pretendemos expandir os estudos comparativos com outros métodos de predição em trabalhos futuros.

O desafio da $\mathbf{PP}_3$ (Seção 5.4.3) estava em verificar a possibilidade da MCGML em realizar a predição de novas conexões a partir das conexões existentes entre os artefatos de metamodelos representados em grafos. Sabíamos que para isso, a MCGML teria que ser capaz de executar tarefas de ML em grafos que trabalham com predições no nível de arestas (*Link-level prediction task*), sendo que o objetivo maior dessas tarefas de predição de links (conexões) é justamente predizer novas conexões a partir daquelas existentes, onde durante os testes, aqueles pares de nós, os quais não possuem links, são classificados, e os $K$ principais pares de nós são preditos. O ponto central dessas tarefas é projetar características para um par de nós. Para esta avaliação experimental relacionada à tarefa de predição de links (arestas) utilizamos também 3 datasets extraídos a partir de 3 categorias diferentes de repositórios de metamodelos (`Petrinet DSLs`, `Bibliography DSLs`, e `Review sys.DSLs` - Tabela 5.4). O **baseline** desses experimentos da $\mathbf{PP}_3$ também foi pautado na instanciação que fizemos do *node2vec*, utilizando como entrada os tipos de arestas simples e os *node embeddings*, onde separamos amostras de arestas aleatórias e as utilizamos como exemplos negativos, adotando o mesmo número de arestas positivas e negativas. Outra questão desafiadora da $\mathbf{PP}_3$ foi definir corretamente a tarefa de predição a ser executada sobre estes datasets, e definir também a métrica de análise adequada para esta avaliação, pois esta avaliação foi baseada em quão bem o modelo de ML em grafos presente na MCGML classificou arestas de teste positivas sobre arestas de teste negativas. E para que esta avaliação fosse possível, classificamos cada aresta positiva presente no conjunto de validação/testes em relação a uma amostra aleatória de 50.000 arestas negativas, e depois contabilizamos a proporção de arestas positivas que estão classificadas entre a $K$-ésima posição

ou acima, ou seja, HITS-$K$. Consideramos nesta avaliação um $K = 50$ como limite para avaliar o desempenho do modelo presente na MCGML. O ajuste desta métrica foi mais desafiador em comparação ao ROC-AUC, porque o modelo de ML presente precisa classificar consistentemente as arestas positivas em posições mais altas do que quase todas as arestas negativas. Sendo uma tarefa extremamente complexa. E essa complexidade da definição foi traduzida em resultados baixos nos percentuais obtidos para o dataset `Petrinet DSLs`, onde a MCGML alcançou níveis na casa dos 30% para HITS-50. Os resultados da avaliação da tarefa de predição de arestas realizada no dataset `Petrinet DSLs` foram apresentados na Tabela 5.8, onde detectamos que a MCGML apresentou falha para *overfit* no conjunto de arestas de treinamento e em virtude disso alcançou desempenhos similares entre os conjuntos de Treinamento/Validação/Testes, a causa para a ocorrência de *overfit* pode estar na ausência de ricas características de nós para este dataset. Mesmo com este percentual baixo, consideramos este resultado animador para afirmar a viabilidade da MCGML.

Repetimos essa mesma dinâmica de avaliação para o dataset `Bibliography DSLs`, onde classificamos cada colaboração verdadeira entre um conjunto de 20.000 colaborações negativas sendo amostras aleatórias, e contabilizamos a proporção de arestas positivas que estão classificadas entre a $K$-ésima posição ou acima (HITS-$K$). Consideramos nesta avaliação um $K = 20$ como um bom limite para avaliar o desempenho do modelo da MCGML (*node2vec*). E para este dataset, obtivemos um resultado um pouco melhor, a Tabela 5.9 mostrou que conseguimos um HITS-20 na casa dos 45% no cenário de testes. Mas esta melhora no desempenho pode ser explicada pelo fato de que as informações posicionais, ou seja, aquelas referentes a colaborações passadas, são recursos mais valiosos para predizer colaborações futuras. Esse experimento reforçou a ideia de tentar melhorar o desempenho nas fases de validação e testes, inserindo informações posicionais na nossa versão da instanciação do *node2vec*, ou incorporando outros modelos de ML como as GNNs (*Graph Neural Networks*). Vale destacar que quando definimos um HITS-10 como métrica de avaliação para o dataset `Review sys.DSLs`, a MCGML voltou a apresentar uma piora nos resultados, com um percentual na casa dos 24% no cenário de testes. Essa variação pode ser parcialmente atribuída à densidade dos grafos e o desafio da divisão desse dataset, onde a divisão baseada nas interações alvo foi mais complexa e desafiadora em relação à divisão aleatória convencional de arestas. No geral, o maior desafio que avaliamos para este dataset `Review sys.DSLs` foi predizer arestas (*links*) fora da distribuição, em grafos com uma certa densidade. Novamente esses resultados não tão bons nas tarefas de predições de *links* nos motiva a buscar alternativas para superar este *gap*, por exemplo, utilizando outros modelos de ML como as GNNs (*Graph Neural Networks*) e o GCN, sendo essa expansão para outros modelos de ML nosso principal desafio de trabalhos futuros.

Na $\mathbf{PP}_4$ (Seção 5.4.4) exploramos outra vertente dos *embeddings*, aqueles realizados no nível de grafo (*graph-level prediction*), na tentativa de verificar a possibilidade da MCGML em predizer propriedades relacionadas a grafos inteiros ou a subgrafos, permitindo assim realizar classificações binárias de documentos de acordo com metamodelos através da classificação de grafos. Novamente utilizamos 3 datasets extraídos a partir de 3 categorias de repositórios de metamodelos (`Office tools DSLs`, `State mac.DSLs`, e `Req.spec.DSLs` - Tabela 5.4). O desafio em responder à $\mathbf{PP}_4$ passava pela definição da tarefa de predição adequada para esta avaliação envolvendo a *graph-level prediction*, onde para o dataset `Office tools DSLs` a tarefa consistia em predizer as propriedades da ferramenta *Office* alvo com a maior precisão possível, onde as propriedades da ferramenta eram vistas como rótulos binários, por exemplo, se uma ferramenta realiza ou não uma conversão de arquivos de um formato para outro (*.doc para *.pdf, por exemplo). A métrica que utilizamos nesta avaliação seguiu as sugestões do trabalho de pesquisa feito por Hao et al. (2021), onde aplicamos a métrica ROC-AUC para a avaliação. A

métrica ROC (*Receiver Operator Characteristic*) representa o relacionamento entre as taxas $FPR$ (*False Positive Rate*) e $TPR$ (*True Positive Rate*). Aplicamos esta métrica nas tarefas de predição sobre os 3 datasets, e em todos os cenários de testes a MCGML apresentou um percentual de ROC-AUC próximo dos 70%, respondendo assim à $\mathbf{PP}_4$ onde podemos afirmar que através da arquitetura MCGML é possível predizer propriedades de grafos (ex: se uma ferramenta possui ou não uma determinada propriedade). Novamente aqui cabe a possibilidade de expansão dessas análises para outros métodos, como por exemplo para a GCN (rede convolucional em grafo *full-batch*).

## 6.6 TRABALHOS RELACIONADOS

Nossas principais contribuições com esta tese estão relacionadas à aplicação de técnicas de ML em grafos em áreas da MDE, tais como a classificação de metamodelos e predição de similaridades entre artefatos de metamodelos. Sendo assim, nesta seção, revisamos alguns trabalhos relacionados à adoção de algoritmos de ML no contexto da MDE, e à categorização de modelos.

### 6.6.1 *Machine Learning* na *Model-Driven Engineering* (MDE)

A área da MDE avançou consideravelmente nos últimos anos em relação à adoção de algoritmos de aprendizado de máquina na tentativa de encontrar soluções para questões diferenciadas (Burgueño et al., 2019). Kusmenko et al. (2019) desenvolveram MontiAnna, um *framework* holístico de modelagem de aprendizado profundo, onde redes neurais podem ser projetadas, treinadas e integradas a sistemas *Component Connector*. Em seguida os autores estenderam o MontiAnna, usando técnicas de *Reinforcement Learning* (RL) (Gatto et al., 2019). No contexto da modelagem colaborativa, Barriga et al. (2018) propuseram a adoção de uma abordagem de *reinforcement learning* não supervisionada na tentativa de reparar modelos "quebrados", que foram corrompidos em virtude de mudanças conflitantes. A intenção primária era reparar modelos com qualidade de um reparo humano sem necessidade de supervisão.

Diversos estudos aplicam técnicas de ML na tentativa de inferir automaticamente regras de transformação de modelos a partir de conjuntos de modelos fontes e modelos alvos (Burgueño, 2019; Dolques et al., 2010). *Metalearning* é uma técnica cujo objetivo é aplicar a própria ML para aprender automaticamente os algoritmos e os parâmetros mais apropriados para um determinado problema de ML. Nesse contexto, Hartmann et al. (2019) intercalaram o aprendizado de máquina com a modelagem de domínio. Mais precisamente eles desmembraram a ML em pequenas unidades de aprendizados computáveis, encadeáveis, reutilizáveis, e independentes. Essas micro unidades de aprendizado são modeladas junto com, e/ou no mesmo nível que os conceitos de domínio. Cabot et al. (2017) apresentaram uma discussão importante sobre como a engenharia de software orientada a modelos (MDSE) poderia se beneficiar com a adoção das técnicas de ML, mais especificamente da cognição, que é a "aplicação do conhecimento para potencializar o desempenho e o impacto de um processo", onde *bots* de modelagem, modelos de inferências, e revisores de modelo em tempo real são apenas alguns dos exemplos das novas funcionalidades que podem ser introduzidas através da cognição.

Com base na experiência que adquirimos nesta tese, vislumbramos um potencial de aplicação do modelo de ML instanciado na MCGML para realizarmos análises de similaridades em artefatos de software, aplicando técnicas de recomendações no domínio da MDSE. Já desenvolvemos um classificador MLP de modelos desestruturados que pode ser aplicado a repositórios de artefatos. Como trabalhos futuros, planejamos melhorar ainda mais a abordagem

proposta com a MCGML, empregando redes neurais em grafos e fazendo um estudo comparativo com as redes neurais *long short-term memory* (LSTM).

### 6.6.2 Classificação e Categorização de Metamodelos

Técnicas de classificação e clusterização estão sendo aplicadas para categorizar metamodelos em diversos domínios, potencializando assim o reuso. Mais precisamente, técnicas de clusterização hierárquica aglomerativa vêm sendo aplicadas na tentativa de realizar a clusterização de metamodelos em repositórios automaticamente. Técnicas de clusterização hierárquica são o principal mecanismo para desempenhar a comparação, a análise, e a visualização de metamodelos (Babur and Cleophas, 2017). Nessas abordagens, os elementos dos metamodelos são convertidos em termos léxicos, e representados como um modelo no espaço vetorial. Este processo de *encoding* nos permite utilizar os algoritmos de clusterização hierárquica para agrupar objetos similares e servi-los como um dendrograma. Essas abordagens, de certa forma, compartilham com um dos objetivos desta tese, que é contribuir para mitigar os problemas que envolvem as tarefas de classificação e clusterização dentro do domínio da modelagem.

Mezghani et al. (2019) apresentaram uma abordagem orientada a domínio para reuso de requisitos de software. Onde primeiramente, os requisitos capturados de diagramas semiformais são inseridos em modelos, modelos esses que são então analisados para verificar quaisquer prováveis problemas de qualidade. E por último, um mecanismo para agrupar (tarefa de clusterização) os requisitos é aplicado para permitir que os padrões dos requisitos de domínio sejam identificados e reutilizados. Já Robinson and Woo (2004) desenvolveram uma ferramenta usada para recuperar automaticamente artefatos UML relacionados, e então a ferramenta sugeria esses artefatos ao desenvolvedor que estivesse modelando. Um algoritmo de busca era aplicado para encontrar artefatos semelhantes, classificando-os em uma rede conceitual de similaridades. Em seguida, a ferramenta desenvolvida comparava a consulta do artefato de entrada com esta rede conceitual de similaridades, e assim tentava retornar o conjunto de artefatos mais próximos em similaridades.

Há trabalhos que discutem as diferentes características dos mecanismos de extensão do metamodelo UML em um esquema que envolve quatro níveis de classificação (Jiang et al., 2004). Onde cada nível de extensão do metamodelo tem diferentes características em aspectos contrastantes como: legibilidade, capacidade de expressão, escopo de uso, e suporte de ferramentas. O objetivo do trabalho era fornecer aos desenvolvedores uma base teórica confiável para selecionar o nível correto para estender o metamodelo visando encontrar a compensação correta baseada nos aspectos citados. Um modelo de classificação para estereótipos UML foi desenvolvido por Berner et al. (1999), onde os artefatos eram analisados de acordo com seu potencial de alterar a sintaxe e a semântica da linguagem base, para poder controlar a aplicação desses artefatos na prática. Critérios baseados em recursos para classificar abordagens de acordo com o tipo de relação envolvida têm sido amplamente discutidas (Bottoni and Grau, 2004). Em contraste a esta tese, nossa pesquisa trata de uma abordagem empírica que explora o domínio da área de ML em grafos para contribuir com as tarefas de classificação, clusterização e análise de similaridades em artefatos relacionados em metamodelos representados na forma de grafos.

A maioria das abordagens que propõem realizar a tarefa de classificação são baseadas em informações léxicas e estruturais codificadas nos metamodelos. Podemos aproveitar este conhecimento, em uma análise mais consciente em relação à semântica, quando os artefatos são restritos a categorias específicas. Por exemplo, a abordagem proposta por Mezghani et al. (2019) concentra-se em diagramas de requisitos, onde os vários relacionamentos existentes no diagrama têm significados que ajudam alcançar um melhor desempenho nas tarefas de classificação. Recentemente, técnicas de clusterização genéricas têm sido constantemente investigadas. O

trabalho apresentado por Nielsen et al. (2019) apresenta uma ferramenta que dá suporte à tarefa de decomposição de um metamodelo em *clusters* de elementos de modelos, onde o objetivo principal é melhorar a manutenibilidade do modelo, facilitando a compreensão do modelo. Mais precisamente, a modularidade é obtida através da decomposição do modelo de entrada em um conjunto de submodelos. A principal característica que torna este estudo diferente de outros é que ele funciona dividindo um único modelo em *clusters*, onde uma fase pós-processamento é introduzida: aos usuários é permitido adicionar mais *clusters*, renomeá-los, removê-los, bem como reagrupar os já existentes. De fato, os *clusters* frutos do resultado desse processo ficam sem identificação dentro de um domínio, e isso difere da abordagem trazida nesta tese com a MCGML, onde as tarefas de classificação realizadas estão apenas agrupando artefatos com similaridades em um mesmo repositório.

### 6.6.3 Aplicações Envolvendo Redes Neurais

A capacidade de aprender a partir de dados rotulados sustenta a principal força das redes neurais, tornando-as uma técnica bem definida na área de aprendizado de máquina. Para citar apenas algumas, reconhecimento de padrões (Bishop, 1995), previsão (Zhang et al., 1998) e classificação (Saeed et al., 2022) são os principais domínios de aplicação das redes neurais. Nas tarefas de classificação, as redes neurais demonstraram sua adequação em várias aplicações. Mitschke et al. (2018) investigaram dois tipos diferentes de redes neurais para classificação: as redes neurais probabilísticas e as redes neurais por retro-propagação (*back-propagation*), e descobriram que apenas as probabilísticas são adequadas para as tarefas de detecção de novos padrões. As redes neurais convolucionais (CNNs) foram originalmente projetadas para trabalhar com imagens, e assim, elas têm sido amplamente utilizadas na resolução de vários problemas que envolvem as tarefas de reconhecimento. Para citar algumas aplicações, as CNNs já foram utilizadas para detectar azeitonas (Gatica et al., 2013), para classificar imagens (Krizhevsky et al., 2012). Esse sucesso das CNNs na área da visão computacional foi uma fonte de inspiração para aplicações em outros domínios. Por exemplo, as CNNs foram aplicadas para classificar sequências de DNA (Aoki and Sakakibara, 2018), e processar linguagem natural (Belinkov and Glass, 2019). As redes neurais foram fonte de inspiração para a MCGML, onde iniciamos nosso desafio de pesquisa explorando as tarefas de classificação de documentos desestruturados com base em metamodelos através de uma rede MLP (Couto. et al., 2020). Temos a previsão de expandir a arquitetura MCGML incorporando métodos de classificação e predição utilizando CNNs.

A ferramenta AURORA (Nguyen et al., 2019), uma aplicação de redes neurais para classificar metamodelos, serviu de inspiração para esta tese, e foi uma das primeiras aplicações de redes neurais para classificar metamodelos. Além disso, o trabalho de Nguyen et al. (2019) está entre as primeiras tentativas de aplicar redes neurais em um domínio completamente novo, ou seja, na MDE. Esta tese se distingue dos estudos existentes na MDE da seguinte forma: *(i)* Empregamos uma técnica de ML bem fundamentada para automatizar a classificação de metamodelos (a ML em grafos); *(ii)* a MCGML pode aprender automaticamente através de metamodelos rotulados para realizar predições em metamodelos não rotulados, podendo ser aplicada a cenários reais de classificação de metamodelos; e *(iii)* por último, mas não menos importante, esperamos que a abordagem proposta abra caminhos para adoções de outros algoritmos avançados de aprendizado de máquina, incluindo também a aplicação de aprendizado profundo (*deep learning*) na MDE (Goodfellow et al., 2016).

## 6.7 PUBLICAÇÕES

Ao longo do trabalho de pesquisa desta tese alcançamos uma importante publicação, e uma submissão para um *journal* referência na MDE:

- Couto., W. O., Morais., E. C., and Didonet Del Fabro, M. (2020). Classifying unstructured models into metamodels using multi layer perceptrons. In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,* pages 271–278. INSTICC, SciTePress.

- Couto., W. O. and Didonet Del Fabro, M. (2022). MCGML Architecture: Classifing Unstructured Models using Machine Learning in Graphs. In *International Journal on Software and Systems Modeling (SoSyM)*. Springer *(submission)*.

## 6.8 CONCLUSÃO

Neste capítulo fizemos uma discussão geral a respeito das contribuições de pesquisa alcançadas com esta tese. Destacando a classificação de modelos através de redes neurais MLP, o processo de *encoding* para converter metamodelos em grafos, a instanciação do algoritmo *node2vec*, e o processo de *embedding*. Discutimos e analisamos nossas avaliações experimentais, e os trabalhos relacionados, comparando algumas abordagens existentes com os métodos presentes da MCGML. Apresentamos também algumas direções para o progresso e expansão da arquitetura com sugestões de trabalhos futuros a serem desenvolvidos em relação às nossas abordagens.

# 7 CONCLUSÕES

Sabemos que a área de classificação de modelos desestruturados em metamodelos é de fundamental importância para a MDE e a MDSE, e esta tese apresentou todos os processos desenvolvidos pelo autor e seus orientadores que serviram de base para a proposta da arquitetura MCGML de manipulação de metamodelos em grafos, capaz de realizar análises de similaridades entre diferentes metamodelos, porém as questões de preparação do vetor de características para ser utilizado por redes neurais, e pelos algoritmos de ML ainda são desafiadoras para a comunidade científica.

Pretendemos com esta proposta contribuir com uma nova perspectiva de ganhos em precisão, desempenho e escalabilidade, aplicando os recursos da análise em grafos, da análise de similaridades, e inferência (predição) de conhecimento que a biblioteca *Stanford Network Analysis* - SNAP e o algoritmo *node2vec* possuem.

Superamos desafios para realizar as adaptações necessárias na arquitetura MCGML visando a extração dos elementos de modelos desestruturados e metamodelos a partir de fontes de dados estruturadas, como JSON ou XMI, e manipulá-los realizando análises de similaridades, predições, e com isso construímos um vetor de características (processo de *embedding*) que foi usado como recurso de entrada para explorar o poderio de classificação e predição dos algoritmos de ML em grafos.

Planejamos expandir a arquitetura MCGML, incorporando outros métodos de classificação, por exemplo utilizando redes neurais convolucionais (CNNs), realizando experimentos práticos, coletando resultados, e comparando esses resultados com outros obtidos por meio de diferentes abordagens no ramo das classificações de modelos.

# REFERÊNCIAS


Acerbis, R., Bongio, A., Brambilla, M., Tisi, M., Ceri, S., and Tosetti, E. (2007). Developing ebusiness solutions with a model driven approach: The case of acer EMEA. In *Web Engineering, 7th International Conference, ICWE 2007, Como, Italy, July 16-20, 2007, Proceedings*, pages 539–544.

Ahmed, A., Shervashidze, N., Narayanamurthy, S., Josifovski, V., and Smola, A. J. (2013). Distributed large-scale natural graph factorization. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, page 37–48, New York, NY, USA. Association for Computing Machinery.

Akbari, M. G., Khorashadizadeh, S., and Majidi, M.-H. (2022). Support vector machine classification using semi-parametric model. *Soft Comput.*, 26(19):10049–10062.

Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39.

Aoki, G. and Sakakibara, Y. (2018). Convolutional neural networks for classification of alignments of non-coding RNA sequences. *Bioinformatics*, 34(13):i237–i244.

Armbrust, M., Das, T., Davidson, A., Ghodsi, A., Or, A., Rosen, J., Stoica, I., Wendell, P., Xin, R., and Zaharia, M. (2015a). Scaling spark in the real world: Performance and usability. *Proc. VLDB Endow.*, 8(12):1840–1843.

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M. (2015b). Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA. ACM.

Atkinson, C. and Kühne, T. (2002). Rearchitecting the uml infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321.

Auth, G. and Maur, E. v. (2002). A software architecture for xml-based metadata interchange in data warehouse systems. In *Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, EDBT '02, pages 1–14, London, UK, UK. Springer-Verlag.

Babur, Ö. and Cleophas, L. (2017). Using n-grams for the automated clustering of structural models. In Steffen, B., Baier, C., van den Brand, M., Eder, J., Hinchey, M., and Margaria, T., editors, *SOFSEM 2017: Theory and Practice of Computer Science*. Springer. 43rd Conference on Current Trends in Theory and Practice of Computer Science : Theory and Practice of Computer Science, SOFSEM ; Conference date: 16-01-2017 Through 20-01-2017.

Barriga, A., Rutle, A., and Heldal, R. (2018). Automatic model repair using reinforcement learning. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*.

Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L., and Pierantonio, A. (2016). Automated clustering of metamodel repositories. In Nurcan, S., Soffer, P., Bajec, M., and Eder, J., editors, *Advanced Information Systems Engineering*, pages 342–358, Cham. Springer International Publishing.

Batra, D. and M.Marakas, G. (1995). Conceptual data modelling in theory and practice. *European Journal on Information Systems*, 4:185–193.

Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., and Mottu, J.-M. (2010). Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143.

Bayitaa, S. A., Nanor, E., Kpiebaareh, M. Y., Agyemang, B., and Wu, W.-P. (2020). Graph based analytics enhanced by deep learning. In *Proceedings of 2020 the 6th International Conference on Computing and Data Engineering*, ICCDE 2020, page 39–43, New York, NY, USA. Association for Computing Machinery.

Beck, F. and Diehl, S. (2013). On the impact of software evolution on software clustering. *Empirical Software Engineering*, 18(5):970–1004.

Belinkov, Y. and Glass, J. (2019). Analysis methods in neural language processing: A survey. *Transactions of the Association for Computational Linguistics*, 7:49–72.

Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., and Launay, D. (2014). Neo4emf, a scalable persistence layer for emf models. In Cabot, J. and Rubin, J., editors, *Modelling Foundations and Applications*, volume 8569 of *Lecture Notes in Computer Science*, pages 230–241. Springer International Publishing.

Bengio, Y. (2009). Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2(1):1–127.

Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1798–1828.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Berkhin, P. (2006). *A Survey of Clustering Data Mining Techniques*, pages 25–71. Springer Berlin Heidelberg, Berlin, Heidelberg.

Berner, S., Glinz, M., and Joos, S. (1999). A classification of stereotypes for object-oriented modeling languages. In *Proceedings of the 2nd International Conference on The Unified Modeling Language: Beyond the Standard*, UML'99, page 249–264, Berlin, Heidelberg. Springer-Verlag.

Bézivin, J. (2005). On the unification power of models. *Software and System Modeling*, 4(2):171–188.

Bishop, C. (1995). *Neural networks for pattern recognition*. Oxford University Press, USA.

Bislimovska, B., Bozzon, A., Brambilla, M., and Fraternali, P. (2014). Textual and content-based search in repositories of web application models. *ACM Trans. Web*, 8(2):11:1–11:47.

Borges, H. and Valente, M. T. (2018). What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112 – 129.

Bottoni, P. and Grau, A. (2004). A suite of metamodels as a basis for a classification of visual languages. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, page 83–90, USA. IEEE Computer Society.

Brambilla, M., Cabot, J., and Wimmer, M. (2012). *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 1st edition.

Burgueño, L. (2019). An lstm-based neural network architecture for model transformations. In *IEEE/ACM 22nd International Conference on Model Driven Engineering Languages and Systems*.

Burgueño, L., Burdusel, A., Gérard, S., and Wimmer, M. (2019). Preface to mde intelligence 2019: 1st workshop on artificial intelligence and model-driven engineering. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 168–169.

Cabot, J., Clarisó, R., Brambilla, M., and Gérard, S. (2017). Cognifying model-driven software engineering. In Seidl, M. and Zschaler, S., editors, *STAF Workshops*, volume 10748 of *Lecture Notes in Computer Science*, pages 154–160. Springer.

Capiluppi, A., Ruscio], D. D., Rocco], J. D., Nguyen, P. T., and Ajienka, N. (2020). Detecting java software similarities by using different clustering techniques. *Information and Software Technology*, 122:106279.

Ciresan, D., Meier, U., Masci, J., and Schmidhuber, J. (2011). A committee of neural networks for traffic sign classification. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1918–1921.

Couto., W. O., Morais., E. C., and Didonet Del Fabro, M. (2020). Classifying unstructured models into metamodels using multi layer perceptrons. In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, pages 271–278. INSTICC, SciTePress.

Davis, J. and Goadrich, M. (2006). The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 233–240, New York, NY, USA. Association for Computing Machinery.

Deng, L. and Yu, D. (2014). Deep learning: Methods and applications. *Found. Trends Signal Process.*, 7(3–4):197–387.

Diem, M., Fiel, S., Garz, A., Keglevic, M., Kleber, F., and Sablatnig, R. (2013). Icdar 2013 competition on handwritten digit recognition (hdrc 2013). In *2013 12th International Conference on Document Analysis and Recognition*, pages 1422–1427.

Dolques, X., Huchard, M., Nebut, C., and Reitz, P. (2010). Learning transformation rules from transformation examples: An approach based on relational concept analysis. In *2010 14th IEEE International Enterprise Distributed Object Computing Conference Workshops*, pages 27–32.

Du, S. S., Hou, K., Póczos, B., Salakhutdinov, R., Wang, R., and Xu, K. (2019). *Graph Neural Tangent Kernel: Fusing Graph Neural Networks with Graph Kernels*. Curran Associates Inc., Red Hook, NY, USA.

Elman, J. L. (1990). Finding structure in time. *Cognitive science*, 14(2):179–211.

Fondement, F. and Silaghi, R. (2004). Defining model driven engineering processes. In *in Proceedings of WISME*.

Friedman, J. H. (2000). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232.

Gasevic, D., Djuric, D., and Devedzic, V. (2009). *Model Driven Engineering and Ontology Development*. Springer Publishing Company, Incorporated, 2nd edition.

Gatica, G., Best, S., Ceroni, J., and Lefranc, G. (2013). Olive fruits recognition using neural networks. *Procedia Computer Science*, 17:412–419. First International Conference on Information Technology and Quantitative Management.

Gatto, N., Kusmenko, E., and Rumpe, B. (2019). Modeling deep reinforcement learning based architectures for cyber-physical systems. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 196–202.

Goodfellow, I. J., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press, Cambridge, MA, USA. `http://www.deeplearningbook.org`.

Grace, K., Salvatier, J., Dafoe, A., Zhang, B., and Evans, O. (2017). When will AI exceed human performance? evidence from AI experts. *CoRR*, abs/1705.08807.

Gronback, R. C. (2009). *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1 edition.

Grover, A. and Leskovec, J. (2016). Node2vec: Scalable feature learning for networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 855–864, New York, NY, USA. ACM.

Hailpern, B. and Tarr, P. (2006). Model-driven development: The good, the bad, and the ugly. *IBM Syst. J.*, 45(3):451–461.

Hamilton, W. L., Ying, R., and Leskovec, J. (2017a). Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, Red Hook, NY, USA. Curran Associates Inc.

Hamilton, W. L., Ying, R., and Leskovec, J. (2017b). Representation learning on graphs: Methods and applications. *CoRR*, abs/1709.05584.

Hao, B., Zhang, J., Yin, H., Li, C., and Chen, H. (2021). Pre-training graph neural networks for cold-start users and items representation. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*, WSDM '21, page 265–273, New York, NY, USA. Association for Computing Machinery.

Harel, D. and Rumpe, B. (2000). Modeling languages: Syntax, semantics and all that stuff part i: The basic stuff. Technical report, Jerusalem, Israel.

Hartmann, T., Moawad, A., Fouquet, F., and Le Traon, Y. (2019). The next evolution of mde: A seamless integration of machine learning into domain modeling. *Softw. Syst. Model.*, 18(2):1285–1304.

Helmy, A. A. (2019). A multilingual encoding method for text classification and dialect identification using convolutional neural network. *CoRR*, abs/1903.07588.

Hochreiter, S. and Schmidhuber, J. (1997a). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hochreiter, S. and Schmidhuber, J. (1997b). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Hodler, A. E. and Needham, M. (2022). Graph data science using neo4j. In *Massive Graph Analytics*, pages 433–457. Chapman and Hall/CRC.

Hoff, P. D., Raftery, A. E., and Handcock, M. S. (2002). Latent space approaches to social network analysis. *Journal of the American Statistical Association*, 97(460):1090–1098.

Hutchinson, J., Rouncefield, M., and Whittle, J. (2011). Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 633–642, New York, NY, USA. ACM.

J. Miller and J. Mukerji (2003). Mda guide version 1.0.1.

Jahrer, M., Töscher, A., and Legenstein, R. (2010). Combining predictions for accurate recommender systems. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, page 693–702, New York, NY, USA. Association for Computing Machinery.

Jain, A. K., Murty, M. N., and Flynn, P. J. (1999). Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323.

Jeh, G. and Widom, J. (2002). Simrank: A measure of structural-context similarity. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, page 538–543, New York, NY, USA. Association for Computing Machinery.

Jiang, Y., Shao, W., Zhang, L., Ma, Z., Meng, X., and Ma, H. (2004). On the classification of uml's meta model extension mechanism. In Baar, T., Strohmeier, A., Moreira, A., and Mellor, S. J., editors, *«UML» 2004 — The Unified Modeling Language. Modeling Languages and Applications*, pages 54–68, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kent, S. (2002). Model driven engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK. Springer-Verlag.

Kipf, T. N. and Welling, M. (2017). Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*, ICLR '17.

Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, page 1137–1143, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Kolovos, D. S., Rose, L. M., Matragkas, N., Paige, R. F., Guerra, E., Cuadrado, J. S., De Lara, J., Ráth, I., Varró, D., Tisi, M., and Cabot, J. (2013). A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 2:1–2:10, New York, NY, USA. ACM.

Koltcov, S., Koltsova, O., and Nikolenko, S. (2014). Latent dirichlet allocation: Stability and applications to studies of user-generated content. In *Proceedings of the 2014 ACM Conference on Web Science*, WebSci '14, page 161–165, New York, NY, USA. Association for Computing Machinery.

Kotsiantis, S. B. (2007). Supervised machine learning: A review of classification techniques. In *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, pages 3–24, Amsterdam, The Netherlands, The Netherlands. IOS Press.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

Kusmenko, E., Nickels, S., Pavlitskaya, S., Rumpe, B., and Timmermanns, T. (2019). Modeling and training of neural processing systems. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 283–293.

Lano, K. (2012). The uml-rsds manual.

Lecun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521:434 – 444.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

LeCun, Y., Kavukvuoglu, K., and Farabet, C. (2010). Convolutional networks and applications in vision. In *Proc. International Symposium on Circuits and Systems (ISCAS'10)*. IEEE.

Leopold, E. and Kindermann, J. (2002). Text categorization with support vector machines. how to represent texts in input space? *Mach. Learn.*, 46(1-3):423–444.

Lerique, S., Abitbol, J. L., and Karsai, M. (2020). Joint embedding of structure and features via graph convolutional networks. *Applied Network Science*, 5(1):5.

Leskovec, J. and Sosič, R. (2016). Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1.

Liben-Nowell, D. and Kleinberg, J. (2007). The link-prediction problem for social networks. *J. Am. Soc. Inf. Sci. Technol.*, 58(7):1019–1031.

Lipton, Z. C. (2015). A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019.

Liu, H. and Bourey, J. P. (2011). Ontology-based semantic annotations for business processes in BPMN2.0. In *IVM, FTMDD, RTSOABIS & MSVVEIS 2011 - Proceedings of the International Joint Workshop on Information Value Management, Future Trends of Model-Driven Development, Recent Trends in SOA Based Information Systems and Modelling and Simulation, Verification and Validation of Enterprise Information Systems, In conjunction with ICEIS 2011, Beijing, China, June, 2011*, pages 24–33.

Ma, G., Ahmed, N. K., Willke, T. L., and Yu, P. S. (2019). Deep graph similarity learning: A survey. *CoRR*, abs/1912.11615.

Mezghani, M., Kang Choi, J., Kang, E.-B., and Sèdes, F. (2019). Clustering for Traceability Managing in System Specifications. In *27th IEEE International Requirements Engineering conference (RE 2019)*, pages 257–264, Jeju Island, South Korea.

Mitschke, N., Heizmann, M., Noffz, K.-H., and Wittmann, R. (2018). Gradient Based Evolution to Optimize the Structure of Convolutional Neural Networks. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 3438–3442. IEEE.

Mokaddem, C. e., Sahraoui, H., and Syriani, E. (2018). Recommending model refactoring rules from refactoring examples. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, page 257–266, New York, NY, USA. Association for Computing Machinery.

Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y., and Jaiswal, S. (2017). graph2vec: Learning distributed representations of graphs. *CoRR*, abs/1707.05005.

Negro, A. (2021). *Graph-powered machine learning*. Simon and Schuster.

Nguyen, P., Di Rocco, J., Di Ruscio, D., Pierantonio, A., and Iovino, L. (2019). Automated classification of metamodel repositories: A machine learning approach. In *IEEE/ACM 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*.

Nguyen, P., Eckert, K., Ragone, A., and Di Noia, T. (2017). Modification to k-medoids and clara for effective document clustering. In *ISMIS 2017 - 23rd International Symposium on Methodologies for Intelligent Systems*, LNCS. Springer.

Nguyen, P., Rocco, J., Iovino, L., Di Ruscio, D., and Pierantonio, A. (2021). Evaluation of a machine learning classifier for metamodels. *Software and Systems Modeling*, 20.

Nguyen, P. T., Rocco, J. D., Rubei, R., and Ruscio, D. D. (2018). Crosssim: Exploiting mutual relationships to detect similar oss projects. In Bures, T. and Angelis, L., editors, *SEAA*, pages 388–395. IEEE Computer Society.

Nielsen, T. D., Rouillard, T., and Makarov, N. (2019). A semantic search capability for a grid model repository.

Olivé, A. (2007). *Conceptual Modeling of Information Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Overbeek, I. J. F. (2006). Meta object facility (mof) investigation of the state of the art.

Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*.

Perini, A., Susi, A., and Avesani, P. (2013). A machine learning approach to software requirements prioritization. *IEEE Trans. Softw. Eng.*, 39(4):445–461.

Perozzi, B., Al-Rfou, R., and Skiena, S. (2014). Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, page 701–710, New York, NY, USA. Association for Computing Machinery.

Phillips, P. J. (1998). Support vector machines applied to face recognition. In *Proceedings of the 11th International Conference on Neural Information Processing Systems*, NIPS'98, page 803–809, Cambridge, MA, USA. MIT Press.

Ponsard, C., Darimont, R., and Touzani, M. (2019). Robust design of a collaborative platform for model-based system engineering: Experience from an industrial deployment. In *Model and Data Engineering: 9th International Conference, MEDI 2019, Toulouse, France, October 28–31, 2019, Proceedings*, page 333–347, Berlin, Heidelberg. Springer-Verlag.

Portugal, I., Alencar, P. S. C., and Cowan, D. D. (2015). The use of machine learning algorithms in recommender systems: A systematic review. *Expert Syst. Appl.*, 97:205–227.

Riedmiller, M. (1994). Advanced supervised learning in multi-layer perceptrons from backpropagation to adaptive learning algorithms. *Computer Standards and Interfaces*, 16(3):265–278.

Robinson, W. N. and Woo, H.-G. (2004). Finding reusable uml sequence diagrams automatically. *IEEE Software*, 21:60–67.

Rokach, L. and Maimon, O. (2006). Data mining for improving the quality of manufacturing: A feature set decomposition approach. *Journal of Intelligent Manufacturing*, 17(3):285–299.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252.

Saeed, A., Shoukat, S., Shehzad, K., Ahmad, I., Eshmawi, A. A., Amin, A. H., and Tag-Eldin, E. (2022). A deep learning-based approach for the diagnosis of acute lymphoblastic leukemia. *Electronics*, 11(19).

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117.

Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31.

Shafiei, M., Wang, S., Zhang, R., Milios, E., Tang, B., Tougas, J., and Spiteri, R. (2007). Document representation and dimension reduction for text clustering. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*, pages 770–779.

Sherstinsky, A. (2018). Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, abs/1808.03314.

Shi, M., Tang, Y., Zhu, X., and Liu, J. (2020). Topic-aware web service representation learning. *ACM Trans. Web*, 14(2).

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition.

Steinwart, I. and Christmann, A. (2008). Support vector machines. In *Information Science and Statistics*.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Tekinerdogan, B., Babur, O., Cleophas, L., van den Brand, M., and Aksit, M. (2019). *Model Management and Analytics for Large Scale Systems*. Elsevier Science.

Thongtanunam, P., Shang, W., and Hassan, A. E. (2019). Will this clone be short-lived? towards a better understanding of the characteristics of short-lived clones. *Empirical Softw. Engg.*, 24(2):937–972.

Tisi, M., Martínez, S., Jouault, F., and Cabot, J. (2011). Lazy execution of model-to-model transformations. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 32–46, Berlin, Heidelberg. Springer-Verlag.

Tixier, A. J.-P., Nikolentzos, G., Meladianos, P., and Vazirgiannis, M. (2019). Graph classification with 2d convolutional neural networks. In *Artificial Neural Networks and Machine Learning – ICANN 2019: Workshop and Special Sessions: 28th International Conference on Artificial Neural Networks, Munich, Germany, September 17–19, 2019, Proceedings*, page 578–593, Berlin, Heidelberg. Springer-Verlag.

Torres, W., vannbsp;den Brand, M. G. J., and Serebrenik, A. (2021). A systematic literature review of cross-domain model consistency checking by model management tools. *Softw. Syst. Model.*, 20(3):897–916.

Ul Haq, I., Gondal, I., Vamplew, P., and Brown, S. (2019). Categorical features transformation with compact one-hot encoder for fraud detection in distributed environment. In Islam, R., Koh, Y. S., Zhao, Y., Warwick, G., Stirling, D., Li, C.-T., and Islam, Z., editors, *Data Mining*, pages 69–80, Singapore. Springer Singapore.

Venuti, K. (2021). Predicting mergers and acquisitions using graph-based deep learning. *arXiv preprint arXiv:2104.01757*.

Wang, K., Shen, Z., Huang, C., Wu, C.-H., Dong, Y., and Kanakia, A. (2020). Microsoft Academic Graph: When experts are not enough. *Quantitative Science Studies*, 1(1):396–413.

Wang, Z., Wang, J., Yang, K., Wang, L., Su, F., and Chen, X. (2022). Semantic segmentation of high-resolution remote sensing images based on a class feature attention mechanism fused with deeplabv3+. *Comput. Geosci.*, 158(C).

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144.

Wuest, T., Weimer, D., Irgens, C., and Thoben, K.-D. (2016). Machine learning in manufacturing: advantages, challenges, and applications. *Production & Manufacturing Research*, 4(1):23–45.

Xie, T. (2018). Intelligent software engineering: Synergy between ai and software engineering. In Feng, X., Müller-Olm, M., and Yang, Z., editors, *Dependable Software Engineering. Theories, Tools, and Applications*, pages 3–7, Cham. Springer International Publishing.

Yanardag, P. and Vishwanathan, S. (2015). Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 1365–1374, New York, NY, USA. Association for Computing Machinery.

You, J., Du, T., and Leskovec, J. (2022). Roland: graph learning framework for dynamic graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 2358–2366.

Zahran, M. A., Magooda, A., Mahgoub, A. Y., Raafat, H., Rashwan, M., and Atyia, A. (2015). Word representations in vector space and their applications for arabic. In Gelbukh, A., editor, *Computational Linguistics and Intelligent Text Processing*, pages 430–443, Cham. Springer International Publishing.

Zhang, C., Ren, M., and Urtasun, R. (2018). Graph HyperNetworks for Neural Architecture Search. *arXiv preprint arXiv:1810.05749*.

Zhang, G., Eddy Patuwo, B., and Y. Hu, M. (1998). Forecasting with artificial neural networks:: The state of the art. *International Journal of Forecasting*, 14(1):35–62.

Zhang, M. and Chen, Y. (2018). Link prediction based on graph neural networks. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 5165–5175. Curran Associates, Inc.

Zhang, Y., Lo, D., Kochhar, P. S., Xia, X., Li, Q., and Sun, J. (2017). Detecting similar repositories on github. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 13–23.

Zhang, Z., Bu, J., Ester, M., Li, Z., Yao, C., Yu, Z., and Wang, C. (2021). H2mn: Graph similarity learning with hierarchical hypergraph matching networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery amp; Data Mining*, KDD '21, page 2274–2284, New York, NY, USA. Association for Computing Machinery.

Zhou, Y., Zheng, H., Huang, X., Hao, S., Li, D., and Zhao, J. (2022). Graph neural networks: Taxonomy, advances, and trends. *ACM Trans. Intell. Syst. Technol.*, 13(1).

Önder Babur (2019). A labeled Ecore metamodel dataset for domain clustering.