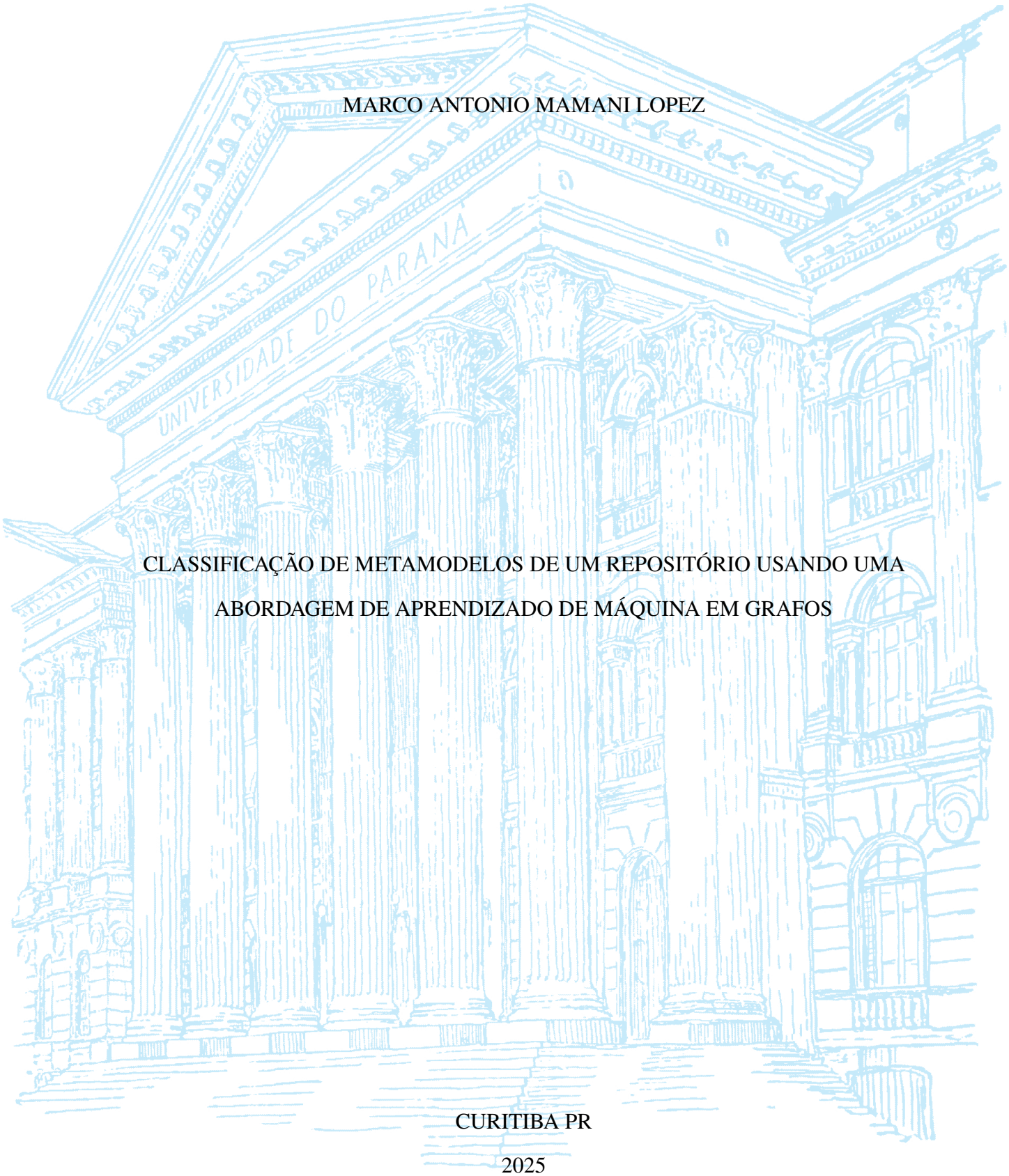
UNIVERSIDADE FEDERAL DO PARANÁ

MARCO ANTONIO MAMANI LOPEZ



CLASSIFICAÇÃO DE METAMODELOS DE UM REPOSITÓRIO USANDO UMA ABORDAGEM DE APRENDIZADO DE MÁQUINA EM GRAFOS

CURITIBA PR

2025

MARCO ANTONIO MAMANI LOPEZ

CLASSIFICAÇÃO DE METAMODELOS DE UM REPOSITÓRIO USANDO UMA ABORDAGEM DE APRENDIZADO DE MÁQUINA EM GRAFOS

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

Área de concentração: *Ciência da Computação*.

Orientador: Dr. Andrey Ricardo Pimentel.

CURITIBA PR

2025

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

Lopez, Marco Antonio Mamani
Classificação de metamodelos de um repositório usando uma abordagem de aprendizado de máquina em grafos / Marco Antonio Mamani Lopez. – Curitiba, 2025.
1 recurso on-line : PDF.

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

Orientador: Andrey Ricardo Pimentel

1. Engenharia de software. 2. Aprendizado do computador. 3. Grafos. 4. Metamodelos. 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 Dissertação de Mestrado de **MARCO ANTONIO MAMANI LOPEZ** intitulada: **Classificação de Metamodelos de um Repositório usando uma abordagem de 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 mestre 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, 15 de Janeiro de 2025.

Assinatura Eletrônica
30/01/2025 11:44:20.0
ANDREY RICARDO PIMENTEL
Presidente da Banca Examinadora

Assinatura Eletrônica
28/01/2025 15:28:05.0
EDUARDO TODT
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
29/01/2025 10:17:08.0
WALMIR OLIVEIRA COUTO
Avaliador Externo (UNIVERSIDADE FEDERAL RURAL DA AMAZÔNIA)

A Deus, por ser minha inspiração.
À minha família, pelo seu amor e apoio incondicional.

AGRADECIMENTOS

À Universidade Federal do Paraná, pela oportunidade de continuar minha formação acadêmica e profissional.

Ao meu orientador e membro da banca, Andrey Ricardo Pimentel, pela disponibilidade e apoio durante todo meu processo de formação.

Aos outros membros da banca, Eduardo Todt e Walmir Oliveira Couto, pelas propostas para aprimorar a presente dissertação.

A todos os professores e funcionários do Departamento de Informática da UFPR, pelos ensinamentos e pelo suporte nesta jornada acadêmica.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Programa de Excelência Acadêmica (PROEX), pelo apoio à formação de novos profissionais.

RESUMO

A implementação da engenharia de software dirigida por modelos (na língua inglesa *model-driven software engineering* ou MDSE), além de incrementar a eficiência e eficácia do processo da engenharia de software por meio do uso sistemático e confiável de modelos, viabiliza o uso dos repositórios de modelos e metamodelos próprios e de terceiros. Desta maneira, os desenvolvedores podem localizar os modelos mais certos para seu reúso direito ou modelos próximos para adequá-los em um contexto específico. Não obstante, não se pode acreditar que todos os modelos ou metamodelos disponibilizados estejam categorizados corretamente e classificá-los manualmente é um processo complexo que precisa de grandes recursos e apresenta resultados bastante criticáveis. Por conseguinte, a classificação automática de modelos e metamodelos é uma linha de pesquisa aberta e desafiante. Neste sentido, a presente pesquisa classifica os metamodelos de um repositório usando uma abordagem de aprendizado de máquina em grafos, isto é, predizer se o metamodelo é, por exemplo, do tipo: Máquina de Estados, Rede de Petri, Biblioteca, Modelagem, Diagrama de Classes, etc. Para atingir o objetivo, foi usado o repositório ModelSet, que contém 5466 metamodelos do tipo Ecore, incluindo duplicados, rotulados com suas classes, representados em formato de grafos e armazenados em estruturas JSON (*JavaScript Object Notation*), proposto por López et al. (2021, 2022). Como parte da experimentação, os metamodelos foram representados em espaços vetoriais de baixa dimensionalidade por meio de um método de *embedding* de grafos superficiais chamado Graph2Vec desenvolvido por Narayanan et al. (2017). Assim, as representações serviram como fonte de entrada para os três algoritmos de aprendizado de máquina supervisionados: árvores de decisão impulsionadas por gradiente, máquina de vetores de suporte e o perceptron multicamada. Consequentemente, os desempenhos dos algoritmos classificadores foram avaliados por meio da validação cruzada do tipo *tenfolds*. Finalmente, evidenciou-se que a solução proposta neste estudo é eficaz, atingindo acurácias balanceadas máximas de 88,7378% e 79,6906% na classificação do ModelSet com e sem metamodelos duplicados, respectivamente.

Palavras-chave: 1. Classificação Supervisionada de Metamodelos 2. Engenharia de Software dirigida por Modelos 3. Aprendizado de Máquina em Grafos 4. *Embedding* de Grafos

ABSTRACT

The implementation of model-driven software engineering (MDSE), in addition to increasing the efficiency and effectiveness of the software engineering process through the systematic and reliable use of models, enables the use of proprietary and third-party model and metamodel repositories. In this way, developers can locate the most appropriate models for their right reuse or nearby models to fit them in a specific context. However, it cannot be believed that all available models or metamodels are correctly categorized and classifying them manually is a complex process that requires large resources and presents quite criticizable results. Therefore, the automatic classification of model and metamodel is an open and challenging line of research. In this sense, the present research classifies the metamodels of a repository using a graph machine learning approach, that is, to predict whether the metamodel is, for example, of the type: State Machine, Petrinet, Library, Modelling, Class-Diagram, etc. To achieve the objective, the ModelSet metamodel repository was used, which contains 5466 Ecore metamodels, including duplicates, labelled with their classes, represented in graph format and stored in JSON (JavaScript Object Notation) structures, proposed by López et al. (2021, 2022). As part of the experimentation, the metamodels were represented in low-dimensional vector spaces using a shallow graph embedding method called Graph2Vec developed by Narayanan et al. (2017). Thus, the representations served as an input data for the three supervised machine learning algorithms: gradient boosted decision trees, support vector machine and multilayer percentron. Consequently, the performances of the classifier algorithms were evaluated through of tenfolds cross-validation. Finally, the solution proposed in this study is effective, achieving maximum balanced accuracies of 88.7378% and 79.6906% in the classification of the ModelSet with and without duplicate metamodels, respectively.

Keywords: 1. Supervised Classification of Metamodels 2. Model-driven Software Engineering 3. Graph Machine Learning 4. Graph Embedding

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

CIN	Computation-Independent Model
DINF	Departamento de Informática
DSL	Domain-Specific Languages
PIM	Platform-Independent Model
PSM	Platform-Specific Model
PPGINF	Programa de Pós-Graduação em Informática
UFPR	Universidade Federal do Paraná
GBDT	Gradient Boosted Decision Trees
JSON	JavaScript Object Notation
MDA	Model-Driven Architecture
MDSE	Model-Driven Software Engineering
ML	Machine Learning
MLP	Multilayer Perceptron
MLPClassifier	Multilayer Perceptron Classifier
MOF	Meta-Object Facility
OMG	Object Management Group
SVC	Support Vector Classifier
SVM	Support Vector Machine
XGBoost	eXtreme Gradient Boosting
XGBClassifier	eXtreme Gradient Boosting Classifier
XMI	XML Metadata Interchange
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Neste capítulo introdutório são apresentados a motivação desta pesquisa, incluindo o problema, os objetivos e a justificativa do estudo.

1.1 MOTIVAÇÃO

A modelagem é um processo omnipresente nas ciências e tecnologias (Brambilla et al., 2017) e, por conseguinte, nas engenharias, incluindo a engenharia de software (Nguyen et al., 2021, 2019; Brambilla et al., 2017; Pons et al., 2010). Neste processo, os modelos abstraem as características ou propriedades mais importantes e reduzem a complexidade dos sistemas em um contexto ou domínio específico (Brambilla et al., 2017; Pressman e Maxim, 2020; Sommerville, 2016). Desta forma, o uso dos modelos é considerado como um fator chave no sucesso da engenharia de software moderna (Bucchiarone et al., 2020; Cabot et al., 2017; Whittle et al., 2014).

Nesta perspectiva, a engenharia de software dirigida por modelos (na língua inglesa *model-driven software engineering* ou MDSE) emerge como uma metodologia que permite incrementar a eficiência e eficácia do processo da engenharia de software por meio do uso sistemático e confiável de modelos e suas formas recursivas, como os metamodelos (Moin et al., 2022; Bucchiarone et al., 2020; Brambilla et al., 2017; Molina et al., 2012). Essencialmente, os modelos elevam os níveis de abstração e automatização do domínio da complexidade (Brambilla et al., 2017). Por sua vez, os metamodelos, foco principal da pesquisa, representam um conjunto de modelos em um nível ainda mais abstrato (Domingo, 2022; Brambilla et al., 2017), sendo considerados como modelos de modelos. Consequentemente, melhoram a qualidade do produto software (Domingo, 2022; Molina et al., 2012) e até a satisfação do desenvolvedor (Domingo, 2022).

Além disso, a MDSE possibilita a maximização da reusabilidade -ou capacidade de reúso- dos modelos e metamodelos (Domingo, 2022; Khalilipour et al., 2022; Bucchiarone et al., 2020; Yousaf et al., 2019; Brambilla et al., 2017), o que permite diminuir o custo, tempo, esforço e risco no processo de engenharia de software (Domingo, 2022; Khalilipour et al., 2022; Barangi et al., 2021). Especificamente, a reusabilidade possibilita que os metamodelos possam se adaptar a diferentes contextos ou a novas plataformas, acrescentando seu valor -em termos de capacidade, maturidade, incluindo sua evolução-, facilitando sua reparação e refatoração, sua otimização e extensibilidade com novas funcionalidades e, também, sua interoperabilidade com outros artefatos softwares (Khalilipour et al., 2022; Bucchiarone et al., 2020; Brambilla et al., 2017; Da Silva, 2015).

Assim, os desenvolvedores das organizações que implementam o MDSE estão procurando constantemente os metamodelos mais certos para seu reúso direito ou metamodelos próximos para adequá-los a um contexto específico (López et al., 2022; Nguyen et al., 2021, 2019). Esta escolha pode ser feita em seus próprios repositórios ou coleções de metamodelos, com artefatos documentados ou ligados a certos esquemas que permitam sua classificação ou categorização efetiva, ou em outros repositórios livres ou proprietários, alguns deles sob as licenças do software livre ou de código aberto, ainda em alguns casos com metamodelos sem informação do seu domínio ou sem classificação conhecida.

Para reconhecer o domínio dos metamodelos e, desta forma, classificá-los é preciso determinar sua estrutura, sintaxe e semântica em relação outros metamodelos previamente

estabelecidos (Couto, 2023; López et al., 2022; Couto et al., 2020; Nguyen et al., 2021, 2019). A classificação manual dos metamodelos é um processo, sistemático ou não, que requer pessoas altamente treinadas, por tanto, é um processo baseado na subjetividade, altamente tedioso, que precisa de um major tempo e com resultados bastante criticáveis (Couto, 2023; Nguyen et al., 2021, 2019). Em comparação com a classificação automática, o qual reconhece o domínio do modelo eficientemente e pode classificá-lo com major eficácia (Couto, 2023; López et al., 2022; Nguyen et al., 2021, 2019). Uma classificação incorreta pode reduzir à acessibilidade, navegação e a busca dos metamodelos e limitar sua reutilização (Nguyen et al., 2021, 2019; Clarisó e Cabot, 2018).

A classificação automática dos modelos pode ser feita usando a abordagem *cognifying*, o que usa os enfoques ligados -ainda não está restringido- à inteligência artificial na melhoria do desempenho e impacto dos processos aplicáveis ao contexto MDSE (Cabot et al., 2018). Assim, o aprendizado de máquina na língua inglesa *machine learning*, o qual é parte da inteligência artificial, usa algoritmos que dão aos computadores a habilidade de aprender e melhorar automaticamente por meio do reconhecimento de padrões sobre os dados, sem ser explicitamente programado para tal fim (Géron, 2023; Jiang, 2021; Lee, 2019; Samuel, 1959).

Um dos tipos mais frequentes do aprendizado de máquina é a aprendizagem supervisionado (Alpaydın, 2021; Stamile et al., 2021; Lee, 2019), sendo suas duas tarefas a regressão e a classificação. A regressão, procura a estimação de valores contínuos, e a classificação, tenta prever as categorias ou classes -como resultados discretos- do conjunto de dados ou instâncias disponibilizadas (Géron, 2023; Alpaydın, 2021; Jiang, 2021; Lee, 2019). Neste sentido, uma classe é interpretada como um conjunto de instâncias as quais compartem características comuns. Adicionalmente, neste sentido, pode se acreditar que na engenharia a classificação é um trabalho mais empírico (Alpaydın, 2021).

Desta forma, no contexto da pesquisa, pode se reconhecer que a abordagem mais efetiva para representar um metamodelo -e preservar suas propriedades- é basada em grafos (Couto, 2023; López et al., 2022; Nguyen et al., 2021, 2019; Couto et al., 2020; Brambilla et al., 2017; Molina et al., 2012). Consequentemente, as classes, atributos e referencias dos metamodelos, por exemplo, do tipo Ecore ou UML(*Unified Modeling Language*), podem ser interpretados como uma coleção de nós (entidades, objetos ou metaclasses) e arestas (relações ou referencias entre nós). Por sua vez, sob uma perspectiva de aprendizado de máquina em grafos, os grafos podem ser representados em espaços vetoriais latentes usando os métodos *embeddings* de grafos ou na língua inglesa *graph embeddings* (Stamile et al., 2021; Zhou et al., 2021; Hamilton, 2020) e, assim, classificá-los coerentemente (Negro, 2021; Stamile et al., 2021; Géron, 2023).

Especificamente, neste estudo, como parte do processo de aprendizado de máquina em grafos, foram abstraídas as características mais relevantes dos metamodelos de um repositório, incluindo algumas propriedades topológicas, usando um método de *embedding* de grafos inteiros, que os mapeou em representações ou espaços vetoriais densos, de baixa dimensionalidade e de longitude fixa. Consequentemente, as representações vetoriais geradas -com suas classes ou categorias rotuladas- foram usados por três algoritmos de aprendizado de máquina supervisionados para classificá-los. E, na última parte deste processo, os desempenhos dos classificadores avaliaram-se usando a técnica de validação cruzada do tipo *tenfolds* ou na língua inglesa *tenfolds cross-validation*.

1.2 PROBLEMA DA PESQUISA

Como classificar os metamodelos de um repositório usando uma abordagem de aprendizado de máquina em grafos?

1.3 OBJETIVOS DA PESQUISA

1.3.1 Objetivo Geral da Pesquisa

Classificar os metamodelos de um repositório usando uma abordagem de aprendizado de máquina em grafos.

1.3.2 Objetivos Específicos da Pesquisa

- Implementar um método de *embedding* de grafos sob uma abordagem de grafos inteiros para mapear os metamodelos de um repositório em representações vetoriais densas, de baixa dimensionalidade e de longitude fixa.
- Implementar e comparar o desempenho de três algoritmos de aprendizado de máquina para classificar as representações vetoriais geradas.

1.4 JUSTIFICATIVA DA PESQUISA

A disponibilidade de repositórios de metamodelos -próprios e de terceiros- permitem que as organizações que adotam a metodologia de engenharia de software dirigida por modelos possam reusar os metamodelos em contextos específicos (López et al., 2022; Nguyen et al., 2021, 2019). Desta forma, a MDSE melhora significativamente a eficiência e eficácia -especificamente em termos de desempenho e de qualidade- do processo de engenharia de software (Bucchiarone et al., 2020; Brambilla et al., 2017; Molina et al., 2012).

Embora, neste contexto, não se pode acreditar que todos os metamodelos disponibilizados estejam categorizados corretamente, consequentemente, a classificação automática de metamodelos é uma linha de pesquisa aberta e desafiante (Couto, 2023; López et al., 2022; Bucchiarone et al., 2020; Couto et al., 2020; Nguyen et al., 2021, 2019). Além disso, a abordagem *cognifying* é uma área de interesse da comunidade da MDSE (Cabot et al., 2018), envolvendo o aprendizado de máquina em grafos -com seus métodos de *embedding*- e possibilitando a implementação de uma solução viável neste contexto (Couto, 2023; López et al., 2022; Nguyen et al., 2021, 2019).

Adicionalmente, pode se acreditar que o aprendizado de máquina em grafos oferece alguns dos algoritmos mais fascinantes e poderosos, os quais estão recebendo cada vez mais atenção, devido ao grande poder expressivo dos grafos (Wu et al., 2022; Negro, 2021; Stamile et al., 2021; Hamilton, 2020; Liu e Zhou, 2020), os quais podem ser o futuro do aprendizado de máquina (Negro, 2021; Stamile et al., 2021).

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os principais conceitos, métodos e técnicas relacionados à engenharia de software dirigida por modelos e ao aprendizado de máquina sob um abordagem tradicional e em grafos. Destacando a importância dos metamodelos, da classificação e reusabilidade de metamodelos, da aproximação ao *cognifying* e dos métodos de *embedding* de grafos.

2.1 ENGENHARIA DE SOFTWARE DIRIGIDA POR MODELOS

A engenharia de software dirigida por modelos, na língua inglesa *model-driven software engineering* (MDSE) ou *model-based software engineering* (MBSE), é uma metodologia (a maneira de um conjunto de instrumentos e diretrizes) que aproveita as vantagens da modelagem, ao elevar os níveis de abstração e de automatização do domínio da complexidade e, desta forma, aumentar a eficiência e eficácia do processo de engenharia de software (Bucchiarone et al., 2020; Brambilla et al., 2017; Molina et al., 2012). Além disso, a MDSE possibilita o reúso de modelos e suas formas recursivas, como os metamodelos (Domingo, 2022; Khalilipour et al., 2022; Bucchiarone et al., 2020; Yousaf et al., 2019; Brambilla et al., 2017). Igualmente, a MDSE reduz o custo, tempo, esforço e risco da produção do software (Domingo, 2022; Khalilipour et al., 2022; Barangi et al., 2021), garante a interoperabilidade entre todos os modelos e ferramentas, promove a engenharia inversa de sistemas legados, formaliza os processos de negócios (Brambilla et al., 2017), acrescenta a qualidade do produto (Domingo, 2022; Molina et al., 2012) e melhora a satisfação do desenvolvedor (Domingo, 2022).

Neste abordagem, a modelagem é referida como o processo de abstração ou representação de objetos ou entidades, desenvolvido por meio da generalização de características específicas dos objetos, da classificação dos objetos em grupos coerentes e da agregação de objetos em entidades mais complexas (Brambilla et al., 2017). Desta forma, na pesquisa, o modelo pode se conceitualizar como uma representação abstrata simplificada total ou parcial de uma entidade ou coleção de entidades, feita para realizar uma tarefa ou chegar a um acordo sobre um tema sob condições ou situações de interesse em um contexto específico (Brambilla et al., 2017; Molina et al., 2012). Por sua vez, o metamodelo representa um conjunto de modelos em um nível ainda mais abstrato (Domingo, 2022; Brambilla et al., 2017), sendo considerado como modelo de modelos.

Especificamente, o modelo tem que apresentar os seguintes três critérios para se distinguir de qualquer outro tipo de artefato: o primeiro é o critério de mapeamento, referido à possibilidade de identificar o objeto ou sistema original que está sendo representado; o segundo é o critério de redução, associado à representação das propriedades ou aspectos mais relevantes do modelo original; e, finalmente, o terceiro é o critério pragmático, relacionado à utilidade do modelo em um contexto específico, o que pode ser a prescrição no sentido de uma especificação, descrição para explicações, simulação ou avaliação formal com propósitos de análise, etc. (Cabot et al., 2018; Brambilla et al., 2017; Da Silva, 2015).

Na MDSE, os modelos permitem analisar e simular as propriedades do sistema ao abordar visões diferenciadas -sob diferentes níveis de abstração e graus de detalhes- em vários estágios do processo da engenharia de software, desde a conceitualização e design inicial até a verificação, implementação, teste, manutenção e evolução do produto software (Moin et al., 2022). Ademais, os modelos podem ser transformados, possibilitando a geração automática

-completa ou parcial- de novos artefatos de software, incluindo o código fonte (Barangi et al., 2021; Brambilla et al., 2017; Molina et al., 2012). Desse modo, o uso dos modelos é considerado como um fator chave no sucesso da engenharia de software moderna (Bucchiarone et al., 2020; Cabot et al., 2017; Whittle et al., 2014).

Coerentemente, segundo Molina et al. (2012), o MDSE apresenta quatro princípios básicos: o primeiro, estabelece que os modelos são usados para abstrair as características -estruturais, sintácticas e semânticas- de um sistema de software; o segundo principio, indica que os modelos devem ser expressados usando uma linguagem de domínio específico ou na língua inglesa *domain-specific language* (DSL); o terceiro principio, precisa que os DSL sejam definidos separando sua notação de sua sintaxe abstrata (representada por um metamodelo); e, o quarto principio, aponta a que as transformações dos modelos permitam à automação do processo de desenvolvimento de software.

No contexto da pesquisa, o MDSE é conceitualizado como uma metodologia que incrementa a eficiência e eficácia do processo da engenharia de software por meio do uso sistemático e confiável dos modelos e suas transformações (Moin et al., 2022; Bucchiarone et al., 2020; Brambilla et al., 2017; Molina et al., 2012).

2.2 NÍVEIS DE MODELAGEM NA ENGENHARIA DE SOFTWARE DIRIGIDA POR MODELOS

Baseada na proposta da *Model-Driven Architecture* (MDA), suportada pelo *Object Management Group* (OMG), a MDSE reconhece três níveis de modelagem sob o nível de abstração: o modelo independente de computação, o modelo independente de plataforma, e o modelo específico de plataforma. Neste contexto, pode se entender à plataforma como um conjunto de subsistemas e tecnologias que fornecem funcionalidades coerentes por meio de interfaces e padrões de uso em contextos específicos (Brambilla et al., 2017; Molina et al., 2012).

O modelo independente de computação ou na língua inglesa *Computation-Independent Model* (CIM) é o nível de modelagem mais abstrato, que representa ao sistema sob a perspectiva do negócio, especialmente, em termos de organização do sistema, papéis, funções, processos, documentações, restrições, etc. Neste sentido, o CIM apresenta exatamente o contexto, os requerimentos e o propósito que a solução deve fazer, mas sem especificações relacionadas à computação ou às TI. Desta maneira, o CIM pode ser referido como um modelo de negócios ou modelo de domínio devido a que preenche a lacuna conceitual entre especialistas de domínio e especialistas em sistemas que conhecem seu projeto e sua construção. Ainda, em princípio, as partes do CIM podem nem mesmo mapear para uma implementação baseada em software (Brambilla et al., 2017; Molina et al., 2012).

O modelo independente de plataforma ou na língua inglesa *Platform-Independent Model* (PIM) é o nível de modelagem intermediário de abstração, que descreve o comportamento e a estrutura do sistema independentemente da tecnologia ou linguagem para sua implementação. Essencialmente, o PIM representa e elicita os requisitos apenas da parte do CIM que será resolvida usando uma solução baseada em software. O PIM apresenta um grau de independência suficiente para permitir seu mapeamento para uma ou mais plataformas de implementação concretas (Brambilla et al., 2017; Molina et al., 2012).

O modelo específico de plataforma ou na língua inglesa *Platform-Specific Model* (PSM) é o nível de modelagem baixo de abstração, que contém todas as informações necessárias sobre o comportamento e a estrutura do software especificadas no PIM combinadas com os detalhes de uma plataforma específica. Assim, o PSM usa os conceitos, abordagens ou arcabouços próprios da plataforma, como: linguagens de programação, mecanismos de exceção, modelos de

componentes, etc., com os quais o PSM poderá ser transformado a código fonte (Brambilla et al., 2017; Molina et al., 2012).

Neste abordagem, como pode se ver na Figura 2.1, os mapeamentos entre cada nível são sequenciais e são definidos por meio das transformações do modelo. Assim, cada CIM pode ser mapeado para diferentes PIMs, e, por sua vez, cada PIM pode se mapear para diferentes PSMs (Brambilla et al., 2017; Molina et al., 2012).

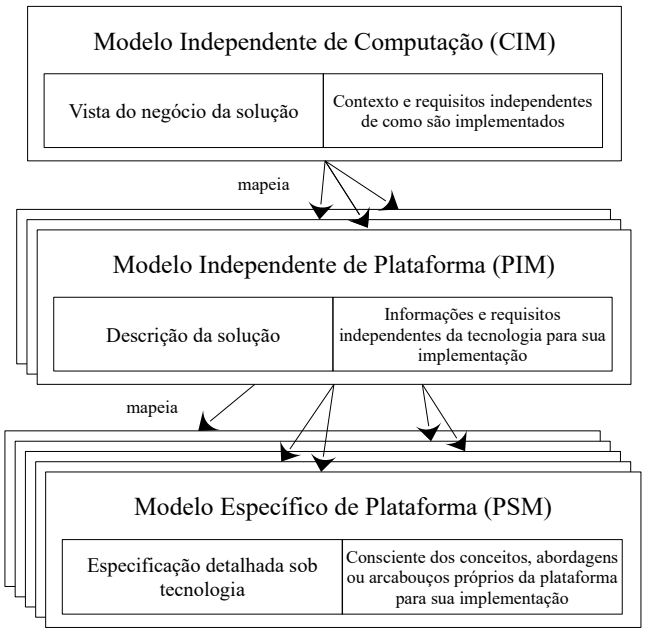


Figura 2.1: Níveis de Abstração de Modelagem segundo a MDA

Fonte: Traduzido de Brambilla et al. (2017).

2.3 METAMODELOS NA ENGENHARIA DE SOFTWARE DIRIGIDA POR MODELOS

A MDSE, segunda a MDA, apresenta uma arquitetura de modelagem de quatro camadas orientadas a padronizar as relações entre o sistema e seus modelos, do nível mais concreto ao mais abstrato (Brambilla et al., 2017; Da Silva, 2015). Além do conceito do modelo, nesta estrutura, pode se conceitualizar ao metamodelo como uma representação mais abstrata de um modelo ou de um conjunto de modelos, sendo considerado como modelo de modelos (como pode se ver na Figura 2.2). Desta forma, o metamodelo é definido intencionalmente sob uma linguagem de modelagem (Brambilla et al., 2017; Molina et al., 2012), especificamente mediante uma linguagem domínio específico (DSL) com suas estruturas e relações próprias (sintaxe abstrata, gramática e semântica estática).

Assim, nesta arquitetura, como pode se ver na Figura 2.3, a primeira camada chamada M0 representa o sistema em estudo ou na língua inglesa *system under study* (SUS), o qual é o nível mais concreto (Brambilla et al., 2017), com elementos do domínio ou instâncias já existentes no mundo real ou que podem existir no futuro (Da Silva, 2015), como por exemplo: um aplicativo, uma plataforma ou qualquer outro artefato de software. A segunda camada chamada M1 representa aos modelos que abstraem os aspectos mais importantes dos elementos do sistema da camada M0, os modelos são expressados principalmente usando o UML (*Unified Modeling Language*). A terça camada chamada M2 representa aos metamodelos dos modelos da camada

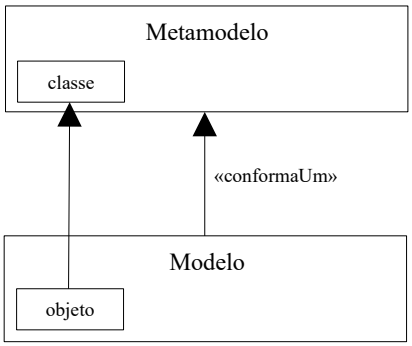


Figura 2.2: Relações entre Modelo e Metamodelo

Fonte: Traduzido de Brambilla et al. (2017).

M1, os metamodelos podem ser descritos por linguagens como o DSL, MOF (*Meta-Object Facility*) suportada pela OMG, UML, Ecore proposto por EMF (*Eclipse Modeling Framework*), etc. E, a quarta camada chamada M3 representa aos meta-metamodelos dos metamodelos da camada M2. Nesta estrutura, o meta-metamodelo é uma descrição mais abstrata, autocontida e autoreflexiva de metamodelos sob um domínio específico, e podem ser definidos por linguagens como MOF, KM3 (*Kernel MetaMetaModel*), entre outras (Brambilla et al., 2017; Da Silva, 2015; Molina et al., 2012).

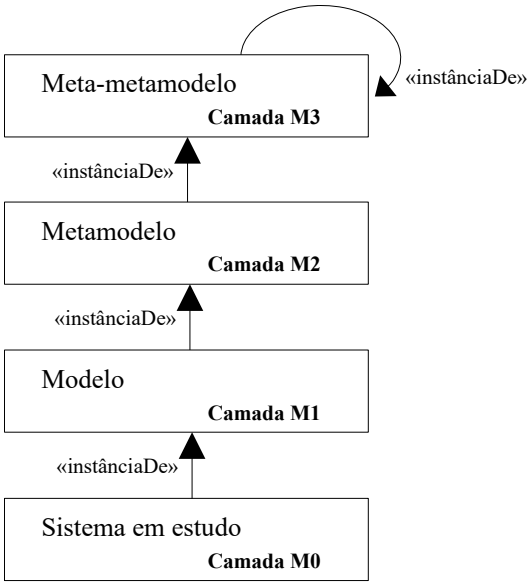


Figura 2.3: Arquitetura de Modelagem de Quatro Camadas

Fonte: Traduzido de Brambilla et al. (2017).

Então, neste contexto é possível acreditar que cada elemento da camada M0 é uma instância de um modelo da camada M1; do mesmo modo, cada modelo da camada M1 é uma instância de um metamodelo da camada M2, e recursivamente, cada metamodelo da camada M2 é uma instância de um meta-metamodelo da camada M3 (Brambilla et al., 2017; Da Silva, 2015; Molina et al., 2012). Em teoria pode se definir infinitos níveis de metamodelagem, mas foi demonstrado, na prática, que os meta-metamodelos podem ser definidos recursivamente e, portanto, geralmente não faz sentido um nível superior de abstração (Brambilla et al., 2017).

2.4 METAMODELOS ECORE NA ENGENHARIA DE SOFTWARE DIRIGIDA POR MODELOS

Os metamodelos, ao igual que os modelos, são expressados por meio de alguma notação que depende de sua finalidade e de seus destinatários. Na MDSE, os metamodelos precisam de ser representados sob uma linguagem de metamodelagem que forneça uma sintaxe abstrata, gramática e semântica estática permitindo a definição de sua estrutura lógica. Desta maneira, a OMG padronizou uma linguagem de metamodelagem chamada *Meta Object Facility* (MOF). Baseado no MOF, diversas linguagens e ferramentas para metamodelagem foram propostas, sendo a mais prominente o *Eclipse Modeling Framework* (EMF) que oferece a linguagem de metamodelagem chamado Ecore (Brambilla et al., 2017; Steinberg et al., 2009).

Como pode se ver na Figura 2.4, conforme descrito por (Brambilla et al., 2017) e Steinberg et al. (2009), a arquitetura do metamodelo Ecore apresenta alguns elementos principais como: EClass, EAttribute, EDataType e EReference. O EClass é usado para representar a classe modelada, o EAttribute denota um atributo do modelo, EDataType representa o tipo dado e o elemento EReference que indica uma associação entre classes. Considerando que um EClass pode ter vários elementos EClass, EReferences ou EAttributes para definir suas características estruturais. Por sua vez, um EAttribute somente pode possuir um EDataType.

Adicionalmente, todos os metamodelos Ecore têm o elemento EPackage, e qual pode agrupar múltiplos EClass relacionadas, assim como outros EPackages (Brambilla et al., 2017; Steinberg et al., 2009). No caso do repositório ModelSet, alguns dos metamodelos apresentam o elemento EGenericType, o qual é um elemento genérico, que pode ser parametrizado com outros elementos.

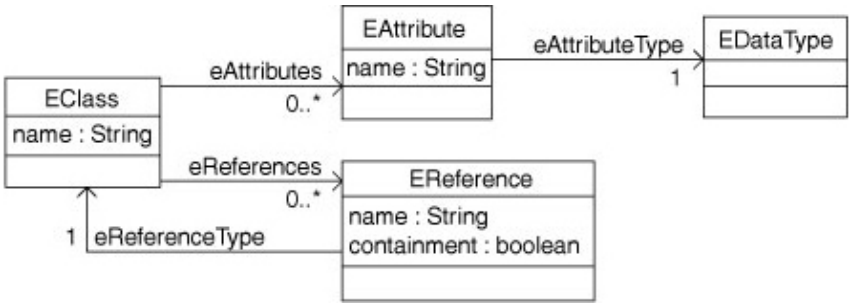


Figura 2.4: Elementos Principais da Arquitetura de um Metamodelo Ecore

Fonte: Steinberg et al. (2009).

2.5 CLASSIFICAÇÃO E REUSABILIDADE DE METAMODELOS NA ENGENHARIA DE SOFTWARE DIRIGIDA POR MODELOS

As organizações que adotam a metodologia de MDSE estão procurando a automatização consistente de seus processos de engenharia de software (Brambilla et al., 2017; Molina et al., 2012) e, por sua vez, estimulando a otimização no uso de seus recursos (financeiros e temporais) e de suas capacidades humanas (Domingo, 2022; Barangi et al., 2021). Deste modo, os modelos, não são apenas artefatos de documentação, são os elementos centrais que dirigem o processo (Barangi et al., 2021; Brambilla et al., 2017; Da Silva, 2015; Molina et al., 2012).

Neste contexto, uma boa prática para armazenar, rastrear, integrar (Di Rocco et al., 2015), gestionar (Bucchiarone et al., 2020) e disponibilizar os modelos e suas formas recursivas, como os metamodelos, é usando os repositórios de software. Neste repositório, os modelos

e metamodelos deveriam estar documentados e classificados segundas suas características, funcionalidades ou sua estrutura. Ainda, não todos os metamodelos são criados e documentados pelos próprios desenvolvedores. Além disso, é comum procurar metamodelos em repositórios de terceiros, alguns deles sob as licenças do software livre ou de código aberto, como por exemplo: SourceForge, Bitbucket, GenMyModel e o GitHub.

Para a classificação de metamodelos é preciso reconhecer seu domínio, segunda sua sintaxe abstrata, gramática e sua semântica estática (Couto, 2023; López et al., 2022; Couto et al., 2020; Nguyen et al., 2021, 2019). Assim, é possível classificá-los em forma manual ou automaticamente. A classificação manual é um processo heurístico, o qual pode ser sistematizado, dependendo do treinamento ou da perícia -incluindo a intuição- das pessoas (modeladores ou classificadores), ainda seus resultados são bastante subjetivos e criticáveis. Em contrapartida, à classificação automática, o qual é mais eficiente respeito ao uso do tempo, dos recursos computacionais e das capacidades humanas, além de ser menos custosas e com resultados mais eficazes (López et al., 2022; Nguyen et al., 2021, 2019).

Na MDSE, um dos mecanismos que agiliza o processo da engenharia de software -diminuindo seu custo, tempo, esforço e risco- é a reusabilidade ou reúso de todos os tipos de modelos (Domingo, 2022; Khalilipour et al., 2022; Bucchiarone et al., 2020; Yousaf et al., 2019; Brambilla et al., 2017). Neste contexto, a reusabilidade permite que os modelos possam se adaptar a diferentes contextos ou a novas plataformas e, também, conseguindo otimizar e estender suas funcionalidades (Khalilipour et al., 2022; Bucchiarone et al., 2020); melhorando sua capacidade de reparação e refatoração (Bucchiarone et al., 2020; Da Silva, 2015) e sua interoperabilidade com outros artefatos de softwares (Brambilla et al., 2017). Desta forma, pode se acreditar que o reúso de modelos acrescenta seu valor, principalmente, em termos de capacidade e maturidade, incluindo sua evolução.

No contexto da pesquisa, a classificação -baseada no reconhecimento do domínio- tentará predizer a categoria ou a classe do metamodelo, isto é, prever se o metamodelo é, por exemplo, do tipo: Máquina de Estados, Rede de Petri, Biblioteca, Modelagem, Diagrama de Classes, etc. E, por consequência, pode se acreditar que os metamodelos classificados coerentemente acrescentarão sua reusabilidade (López et al., 2022; Nguyen et al., 2021, 2019).

2.6 *COGNIFYING* APLICADO À ENGENHARIA DE SOFTWARE DIRIGIDA POR MODELOS

Em 2016, Kelly em seu livro *The Inevitable*, propôs o termo *Cognifying*, aludindo aos produtos ou serviços que façam uso do conhecimento inferido pelas distintas formas de inteligência artificial tentando conseguir mais efetividade nas atividades humanas e, na sua vez, considerando-o como uma das doze forças tecnológicas que mudarão nosso mundo. Baseada nesta ideia, Cabot et al. (2018) aplicaram o conceito do *Cognifying* à engenharia de software dirigida por modelos para referir-se ao uso dos abordagens ligados à inteligência artificial, inteligência coletiva, *crowdsourcing* e outros relacionados na melhoria do desempenho e impacto dos processos.

Neste sentido, pode se conceitualizar à inteligência artificial ou na língua inglesa *artificial intelligence* (AI) como a habilidade dos computadores para imitar as funções cognitivas humanas, como o aprendizado, a percepção, o raciocínio e a resolução de problemas (Jiang, 2021) de um modo coerente ao contexto específico; dita imitação é feita por meio do regras pré determinadas ou por padrões baseados em dados.

Assim, o *cognifying* é considerado um grande desafio na área do MDSE, o que permite acrescentar a automatização do processo de desenvolvimento de software, incluindo as

soluções de manutenção, com uma visão quantificável e perceptível (Bucchiarone et al., 2020). Consequentemente, este abordagem pode ser aplicado aos *bots* de modelagem, inferências de modelos, geradores de código, revisores de modelos em tempo real, modelagem autoexplicativo, modelagem escalável com mecanismos de fusão de dados (Cabot et al., 2018), reparação do modelos (Barriga, 2021), classificação de modelos (Couto, 2023; López et al., 2022; Couto et al., 2020; Nguyen et al., 2021, 2019), etc. Sendo o aprendizado de máquina a técnica mais usada no processo de *cognifycation* (Barriga, 2021).

2.7 APRENDIZADO DE MÁQUINA OU APRENDIZAGEM MÁQUINA

O aprendizado de máquina, chamado também na língua inglesa *machine learning* (ML), termo cunhado por Arthur Samuel, pesquisador da IBM, em 1959 (Jiang, 2021; Negro, 2021; Stamile et al., 2021), é uma área da inteligência artificial (veja a Figura 2.5), que pode ser conceptualizada como a aplicação de algoritmos de aprendizado pelos quais os computadores possam melhorar automaticamente sua habilidade em qualquer tarefa específica por meio da experiência (Jiang, 2021). Do mesmo modo, pode se referir aos métodos computacionais que usam a experiência para melhorar o desempenho ao fazer previsões (Mohri et al., 2018). Também, é possível referir-se como ML aos algoritmos que procuram padrões ou modelos em dados (Lee, 2019), sem ser programados explicitamente para tal habilidade (Negro, 2021; Stamile et al., 2021; Samuel, 1959).

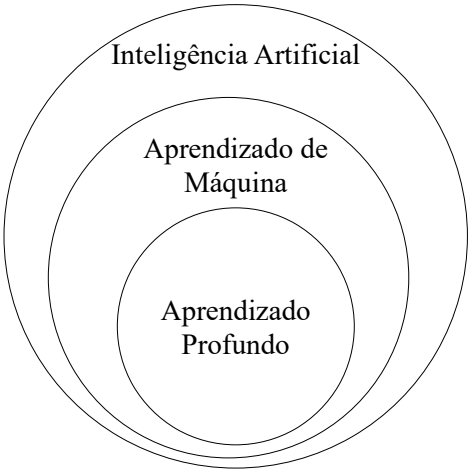


Figura 2.5: Relações entre Inteligência Artificial, Aprendizado de Máquinas e Aprendizado Profundo

Fonte: Elaboração Própria.

Neste contexto, a "experiência" concerne ao uso dos dados passados disponíveis, os quais são usados como elementos imprescindíveis (dados de treinamento) na entrada para o sistema de ML e são processados pelos mecanismos de "aprendizado", que adaptam automática e continuamente seus algoritmos para descobrir relações e regularidades entre eles ou estimar medidas de similaridade (Berzal, 2018). Isto último, pode ser atingido por meio de otimização matemática ou estatística (Stamile et al., 2021), ainda sempre com um erro associado, provocado pela tentativa de generalizar o modelo. Consequentemente, o padrão é aplicado aos novos dados (dados de teste) a fim de predizer ou dar uma resposta coerente sob contexto específico, o que permite avaliar e verificar seu desempenho ou eficácia.

Do anterior, pode se reconhecer a importância dos dados de treinamento e de teste no processo do aprendizado de máquina, os quais podem ser gerados por meio da engenharia

de características ou na língua inglesa *feature engineering*. Especificamente, a engenharia de características extrai representações, principalmente do tipo numéricas, relevantes, compactas e significativas de cada instância do conjunto de dados iniciais e constrói os chamados vetores de características (Géron, 2023; Stamile et al., 2021; Zheng e Casari, 2018).

Na pesquisa, pode se afirmar que o aprendizado máquina usa algoritmos que dão aos computadores a habilidade de aprender e melhorar automaticamente por meio do reconhecimento de padrões nos dados, sem ser explicitamente programado para tal fim (Géron, 2023; Jiang, 2021; Stamile et al., 2021; Lee, 2019; Mohri et al., 2018). Ou em poucas palavras, o ML é a habilidade computacional de aprender da experiência -do passado- para predizer o futuro.

2.8 TIPOS DE SISTEMAS DE APRENDIZADO DE MÁQUINA

Os sistemas de aprendizado de máquina podem ser classificados de acordo com o tipo de supervisão que recebem durante o treinamento e são: o aprendizado supervisionado, não supervisionado, semissupervisionado e o aprendizado por reforço (Géron, 2023; Jiang, 2021; Mohri et al., 2018).

2.8.1 Aprendizado Supervisionado

No aprendizado supervisionado os dados de treinamento são fornecidos ao sistema de ML incluindo as soluções desejadas, os quais são chamados rótulos, alvos, anotações ou na língua inglesa *labels*, *targets* (Géron, 2023; Jiang, 2021; Lee, 2019; Berzal, 2018; Mohri et al., 2018) ou *annotations* (Stamile et al., 2021) respectivamente. O objetivo, na fase de treinamento, consiste em ajustar os parâmetros do sistema para que seja capaz de reproduzir uma saída o mais próxima possível à desejada. Ainda o mais importante, é a capacidade de generalização do modelo para predizer resultados com dados novos ou de teste e validar sua efetividade (Stamile et al., 2021; Berzal, 2018).

As tarefas típicas do aprendizado supervisionado são a regressão e a classificação. A regressão, procura a estimação de valores contínuos e pode ser aplicado a avaliação de riscos, predição de pontuação, geração de imagens, etc. Por sua parte, a classificação, que tenta prever as categorias ou classes -como resultados discretos- do conjunto de dados ou instâncias disponibilizadas, os quais são adequados para classificação de imagens, detecção de fraude, reconhecimento de fala, etc. (Géron, 2023; Alpaydın, 2021; Jiang, 2021; Lee, 2019). O aprendizado supervisionado é um dos tipos mais frequentes do aprendizado de máquina (Alpaydın, 2021; Stamile et al., 2021; Lee, 2019). Não obstante, requer um supervisor -geralmente humano- para coletar e rotular os dados de treinamento, o que pode fazer que este tipo de aprendizado seja caro na prática, mas dependendo da quantidade dos dados de treinamento pode garantir altos níveis de desempenho (Jiang, 2021).

2.8.2 Aprendizado Não Supervisionado

No aprendizado não supervisionado, chamado também aprendizagem observacional (Berzal, 2018), os dados de treinamento não tem rótulos ou classificação conhecida (Géron, 2023; Jiang, 2021; Lee, 2019; Berzal, 2018; Mohri et al., 2018). Assim, o sistema de ML tenta aprender sozinho (sem um supervisor) construindo descrições, hipóteses ou teorias a partir do conjunto de fatos ou observações, sem a existência de informações sobre os resultados esperados (Berzal, 2018) ou nenhuma função de ganho.

As tarefas típicas do aprendizado não supervisionado são: o agrupamento de dados ou na língua inglesa *clustering* e a redução da dimensionalidade. O *clustering* detecta grupos

assumindo que os dados que pertençam a um mesmo grupo possuem atributos semelhantes, é usado nos sistemas de recomendação, segmentação de clientes, etc. E, de outra parte, a redução da dimensionalidade, na qual o objetivo é simplificar os dados sem perder muita informação, feito por meio da mescla de várias características correlacionadas, pode ser aplicado na elicitação de características, descoberta de estruturas, visualização de grandes quantidades de dados, compreensão de significado, etc. (Géron, 2023; Jiang, 2021; Berzal, 2018; Mohri et al., 2018).

2.8.3 Aprendizado Semissupervisionado

O aprendizado semissupervisionado combina as vantagens do aprendizado supervisionado e do não supervisionado, aqui os dados de treinamento estão parcialmente rotulados, o que permite reduzir a necessidade de dados rotulados e aproveitar as grandes quantidades de dados disponíveis sem etiquetas (Géron, 2023; Jiang, 2021; Mohri et al., 2018). Assim, o sistema de ML usa os dados rotulados para inferir o padrão e, seguidamente, rotular o resto dos dados (Mohri et al., 2018), conseguindo reduzir o tempo e os recursos humanos, computacionais e até financeiros.

Este tipo de aprendizado é comum em ambientes onde os dados não rotulados são facilmente acessíveis, mas os dados rotulados são caros para obter. Pode se aplicar às tarefas de classificação, regressão, elicitação, etc. (Mohri et al., 2018).

2.8.4 Aprendizado por Reforço

O aprendizado por reforço, chamado também aprendizagem por recompensas, apresenta um agente que observa o ambiente, seleciona e executa ações para depois receber recompensas e punição (Géron, 2023; Jiang, 2021; Berzal, 2018; Mohri et al., 2018), como um mecanismo de tentativa e erro (Berzal, 2018), com fases de treinamento e de teste bastante interativas (Mohri et al., 2018), a fim de ajustar a estratégia para receber a melhor recompensa, mas sem uma supervisão forte do ambiente (Jiang, 2021).

Este tipo de aprendizado é comum em ambientes incertos e potencialmente complexos (Jiang, 2021), tal como: sistemas de decisões em tempo real, navegação de robôs, aquisição de habilidades, jogos digitais, etc.

2.9 REDES NEURAIS ARTIFICIAIS E APRENDIZADO PROFUNDO

As redes neurais artificiais ou na língua inglesa *artificial neural networks* (ANNs) são modelos matemáticos inspirados nos neurônios biológicos do cérebro e suas conexões (Géron, 2023; Jiang, 2021). Sendo o neurônio artificial a unidade básica de processamento, que recebe sinais ou dados de entrada, os quais são somados ponderada e linearmente, para em seguida serem transformados conforme a funções de ativação não lineares e, finalmente, produzir uma saída ou predição (como pode se ver na Figura 2.6). Assim, as ANNs são modeladas como uma rede distribuída de uma ou várias camadas, onde cada camada tem diferentes números de neurônios, formando uma arquitetura versátil, poderosa e escalável, capaz de lidar com tarefas extremamente complexas (Géron, 2023; Chollet, 2021; Jiang, 2021). Em termos gerais, pode se acreditar que, a capacidade da ANN depende principalmente de como seus neurônios estão ligados e da força dessa conexão (Jiang, 2021).

O aprendizado profundo ou na língua inglesa *deep learning* (DL) é parte dos métodos do aprendizado de máquina, portanto, mantém o mesmo objetivo de aprender e melhorar automaticamente por meio do reconhecimento de padrões nos dados, mas neste caso, usando algoritmos baseados em redes neurais como funções de aproximação (Géron, 2023; Jo, 2021;

Berzal, 2018); permitindo a aprendizagem sob estruturas hierárquicas e níveis de representação e de abstração (Chollet, 2021; Berzal, 2018), o que possibilita a extração automática dos vetores de características -cada vez mais significativos e mais complexos- como parte do processo do DL (Negro, 2021; Berzal, 2018). Além de acrescentar o desempenho na solução de problemas mais complexos e com uma grande quantidade de dados em relação ao aprendizado de máquina tradicional (Géron, 2023; Jiang, 2021).

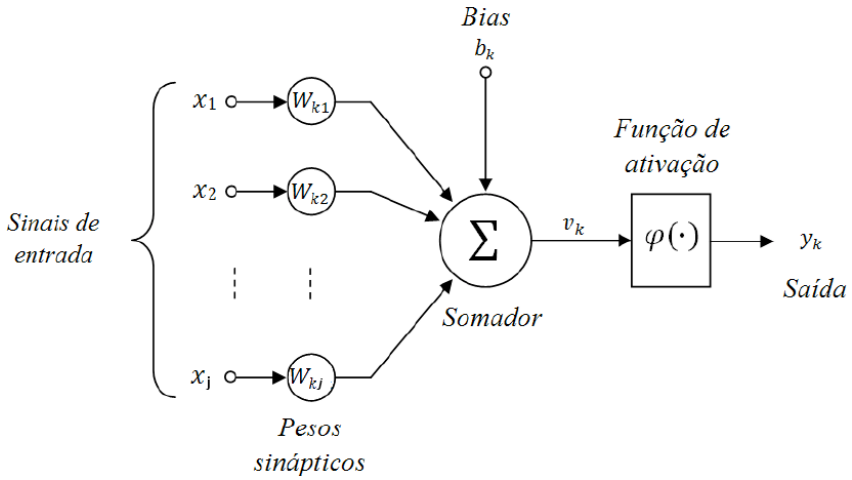


Figura 2.6: Neurônio Artificial

Fonte: Adaptado de Jiang (2021).

Alguns tipos de redes neurais artificiais mais referenciadas na literatura são: as redes neurais recursivas (baseadas em estruturas hierárquicas ou em vetores de composição), redes neurais recorrentes (projetadas para aprender padrões sequenciais ou temporais usando mecanismos de realimentação, memória ou de atenção), redes neurais convolucionais (focadas principalmente à percepção visual) e as redes generativas profundas (com a capacidade de criar ou gerar novas amostras). Todas elas aplicáveis efetivamente ao processamento de sinais, de áudio e de voz, reconhecimento de imagens, ao processamento de linguagem natural entre outras tarefas (Chollet, 2021; Jiang, 2021; Berzal, 2018; Pouyanfar et al., 2018).

2.10 APRENDIZADO DE MÁQUINA EM GRAFOS

O aprendizado de máquina em grafos ou na língua inglesa *graph machine learning* (Graph ML) ou *machine learning on graphs* é uma extensão do aprendizado de máquina, que possibilita o uso dos grafos como uma estrutura de dados ubíqua, altamente complexa e não euclidiana (Hamilton, 2020; Liu e Zhou, 2020). Desta forma, o ML em grafos pode detectar, interpretar e representar automaticamente padrões latentes recorrentes diretamente nos grafos com mais eficiência e eficácia que os abordagens tradicionais (Negro, 2021; Stamile et al., 2021). Além disso, segundo (Negro, 2021) os grafos são uma maneira mais natural de organizar, analisar e processar dados para os sistemas de ML.

Um grafo $\mathcal{G}$ é conceitualizado como um conjunto de vértices $\mathcal{V}$ e arestas $\mathcal{E}$ (da língua inglesa *edges*). Os vértices ou nós representam entidades ou objetos, e as arestas denotam as relações entre ditas entidades. Desta maneira, formalmente um grafo pode se expressar como $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (Wu et al., 2022; Stamile et al., 2021; Negro, 2021; Hamilton, 2020; Liu e Zhou, 2020). Segundo (Hamilton, 2020) as principais vantagens dos grafos estão em sua abordagem nas relações entre as entidades -em vez das propriedades dos objetos individuais- e na sua

generalidade, conseguintemente, os grafos podem ser usados para modelar relacionamentos complexos entre uma coleção de entidades.

Segundo o tipo dos vértices e das arestas, os grafos podem ser homogêneos ou heterogêneos; os grafos homogêneos apresentam seus elementos de um mesmo tipo e, de outra parte, os grafos heterogêneos têm seus vértices e as arestas de diferentes tipos. Respeito à direção das arestas, os grafos são classificados como dirigidos ou não dirigidos; são grafos dirigidos ou digrafos quando suas arestas denotam direção entre os nós, por sua vez, são grafos não dirigidos se suas arestas não têm direção ou são bidirecionais (Wu et al., 2022; Stamile et al., 2021; Negro, 2021; Hamilton, 2020; Liu e Zhou, 2020).

2.11 *EMBEDDING* DE GRAFOS

Todos os sistemas de aprendizado de máquina precisam de dados como fontes de entradas, os quais podem ser gerados por meio da engenharia de características como parte do processo do ML tradicional. Porém, esta técnica pode não ser a solução ótima no aprendizado de máquina em grafos, devido a que os grafos não têm uma estrutura bem definida para todos os casos (Stamile et al., 2021). Além disso, os vetores de características -no aprendizado de máquina tradicional- são feitos a mão e estão limitadas por sua inflexibilidade e alto custo computacional (Liu e Zhou, 2020).

Uma alternativa para o aprendizado de máquina em grafos são os métodos de *embeddings*. Em termos gerais, o *embedding* pode se conceituar como uma representação ou projeção vetorial -de alguma coisa complexa-, em um espaço latente de baixa dimensionalidade, que permite extrair informações ou características úteis, melhorando sua escalabilidade (Hamilton, 2020; Goyal e Ferrara, 2018) e, principalmente, manter suas propriedades topológicas. Especificamente, os *embeddings* de grafos, *embedding* de redes ou representação do aprendizado -na língua inglesa *graph embedding*, *network embedding* ou *representation learning* respectivamente-, geram automaticamente representações ou espaços vetoriais, sob um abordagem de redução de dimensionalidade e de otimização, capturando as informações estruturais, dependências e as semelhanças do grafo (Chami et al., 2022; Yang et al., 2021).

Neste sentido, o *embedding* de grafo permite converter um grafo ou parte dele -de domínio não euclidiano, de alta dimensão e de numéricos discretos- em uma representação vetorial de baixa dimensionalidade, compacta, densa e de valores numéricos contínuos (Chami et al., 2022; Lavrač et al., 2021; Stamile et al., 2021; Cai et al., 2018). Pode ser expressado com uma função de mapeio $f : \mathcal{G} \rightarrow \mathbb{R}^{\delta}$, onde $\mathcal{G}$ é um grafo, e $\mathbb{R}^{\delta}$ a representação vetorial de δ dimensões. Na Figura 2.7, apresentam-se as granularidades do *embeddings* do grafo em espaços bidimensionais, ao nível de nó, aresta, subgrafo e de grafo inteiro.

2.12 TAXONOMIA DOS MÉTODOS DE *EMBEDDING* DE GRAFOS

Atualmente, a taxonomia dos métodos codificadores de *embeddings* de grafos é uma linha de pesquisa, e não há uma classificação comumente aceitada. Na pesquisa, se usou a proposta do Chami et al. (2022) e do Stamile et al. (2021), quem classificam os métodos de *embeddings* de grafos como: *embeddings* superficiais, de codificação automática, de agregação da vizinhança, e de regularização de grafos.

2.12.1 Métodos de *Embeddings* Superficiais

Os *embeddings* superficiais ou na língua inglesa *shallow embedding methods* são métodos que usam o aprendizado transdutivo e só podem retornar uma representação vetorial

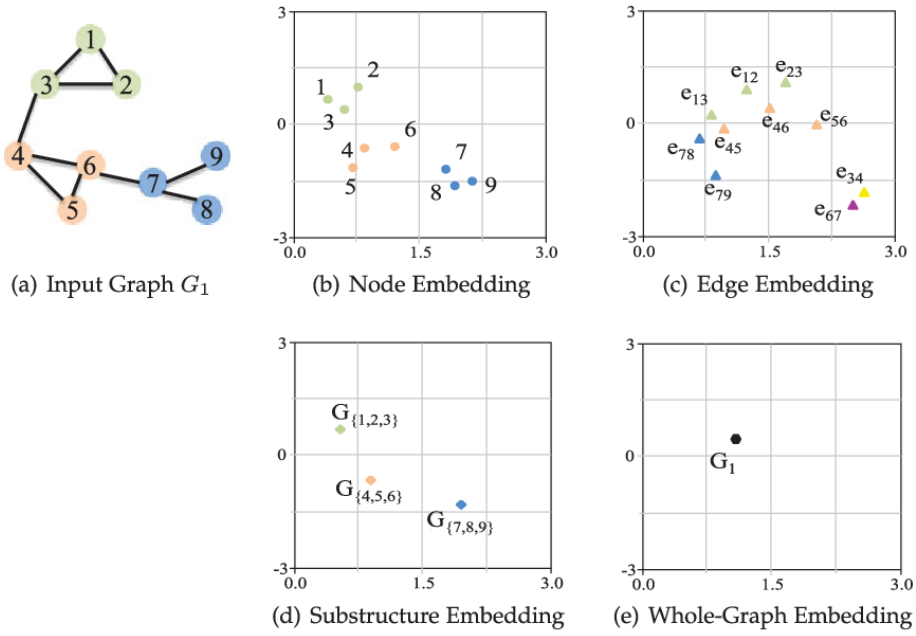


Figura 2.7: Granularidades de *Embeddings* de Grafo em Espaços Bidimensionais

Fonte: Cai et al. (2018).

dos dados que aprenderam durante o processo de ajuste; portanto, não é possível obter uma representação para dados não vistos (Chami et al., 2022; Stamile et al., 2021; Hamilton, 2020). Neste sentido, estos métodos não podem ser aplicados diretamente em configurações indutivas onde a estrutura das instâncias não são fixas (Chami et al., 2022; Hamilton, 2020). Alguns algoritmos que implementam este tipo de *embedding* são: Node2Vec (Grover e Leskovec, 2016), Edge2Vec (Wang et al., 2020a) e Graph2Vec (Narayanan et al., 2017).

2.12.2 Métodos de Codificação Automática

Os métodos de codificação automática ou na língua inglesa *graph autoencoding methods* geram representações vetoriais usando uma função de mapeio mais geral baseada em redes neurais, conseguindo preservar a maioria das informações dos grafos e reduzindo sua dimensionalidade, com a habilidade de representar instâncias não vistas (Stamile et al., 2021; Wang et al., 2020b). Isto é, uma vez que o modelo é ajustado ao conjunto de grafos de treinamento, é possível usá-lo para gerar as representações de novos grafos não vistos.

2.12.3 Métodos de Agregação da Vizinhança

Respeito aos métodos de agregação da vizinhança ou na língua inglesa *neighborhood aggregation methods*, incluindo os métodos convolucionais em grafos, são codificadores que usam a estrutura do grafo para propagar informações e extrair representações ao nível do grafo, onde os nós têm rótulos com algumas propriedades. Ao igual que os métodos de codificação automática, este método também pode gera uma função de mapeamento geral (Chami et al., 2022; Stamile et al., 2021).

2.12.4 Métodos de Regularização de Grafos

Os métodos de regularização de grafos ou na língua inglesa *graph regularization methods* são ligeiramente diferente aos tipos listados anteriormente. Aqui, o codificador ignora a estrutura do grafo e só aprende das características dos nós e da sua interação para regularizar o processo, sob a suposição de que os nós próximos em um grafo provavelmente terão os mesmos rótulos (Chami et al., 2022; Stamile et al., 2021).

3 TRABALHOS RELACIONADOS

Neste capítulo são apresentados trabalhos segundo o mapeamento sistemático da literatura, destacando os processos, métodos e ferramentas relacionados à classificação supervisionada de modelos e metamodelos sob uma abordagem de engenharia de software dirigida por modelos e do aprendizado de máquina em grafos.

3.1 CLASSIFICAÇÃO DE MODELOS DESESTRUTURADOS USANDO UM PERCEPTRON MULTICAMADA E O APRENDIZADO DE MÁQUINA EM GRAFOS

Na engenharia de software dirigida por modelos, os modelos e metamodelos apresentam uma relação de conformidade restrita a domínios específicos. No entanto, há um aumento de formatos de dados desestruturados (semiestruturados, não estruturados ou livres de esquema), como as representações orientadas a documentos, os quais são geralmente criadas como documentos JSON. Embora, estos documentos desestruturados não possuem um esquema de metamodelo explícito, eles podem ser classificados ao descobrir seus domínios e estabelecer certa conformidade -total ou parcial- com um determinado metamodelo previamente especificado.

Neste sentido, Couto et al. (2020) propuseram uma abordagem para analisar e classificar modelos desestruturados usando metamodelos especificados junto a um perceptron multicamada ou na língua inglesa *multiple layer perceptron* (MLP), o que foi complementado por Couto (2023) sob um enfoque de aprendizado de máquina em grafos por meio da arquitetura chamada MCGML, siglas baseadas em *Model Classification using Graph Machine Learning*.

Em ambos os casos, cada modelo e metamodelo é analisado como uma estrutura composta de elementos como classes, atributos e referências. Assim, na primeira etapa desta abordagem, como pode se ver na Figura 3.1, foi usado o Extrator de Metamodelos (definido como *Extraction Function* na literatura original) para distribuir os elementos dos metamodelos a uns *datasets* específicos tais como: $d_{s(0)}, d_{s(1)} ... d_{s(n)}$, os quais, por sua vez, são parte do *dataset* $\mathcal{D}_s$ e onde *n* representa a quantidade total de elementos. Consequentemente, cada um dos *datasets* específicos foram convertidos a vetores de características, incluindo seus rótulos, usando o Codificador com a técnica chamada *one-hot encoding* (OHE). Desta forma, cada elemento do metamodelo foi representado por um número binário (excluindo as diferenças entre classes, atributos e referências) e armazenado em uma estrutura de pares chave-valor no arquivo JSON. Especificamente, Couto (2023) e Couto et al. (2020) usaram quatro metamodelos do tipo MySQL, Java, UML e KM3, disponibilizados pela *ATL Transformations* da *The Eclipse Foundation*, os quais foram convertidos a vetores de características de setenta e dois dimensões.

Coerentemente ao processo deste trabalho, foi implementado um perceptron multicamada composto na camada de entrada por setenta e duas unidades ou sinais -quantidade alinhada ao número de dimensões dos vetores de características dos metamodelos, os quais são os recursos de entrada para esta MLP-, na parte intermediária apresenta três camadas ocultas com três neurônios cada, e, finalmente, um neurônio na camada de saída que retorna o valor predito da classe. Cada neurônio apresenta uma função de ativação do tipo sigmoide, um viés ou *bias* e conexões densas (como pode se ver na Figura 3.2). Assim, a MLP foi executada com inicialização aleatória convencional e retropropagação.

Na segunda parte deste trabalho, usando um processo muito similar ao Extrator de Metamodelos e ao Codificador descritos anteriormente, Couto (2023) e Couto et al. (2020) conseguiram representar os modelos desestruturados -disponibilizados em formato JSON- em

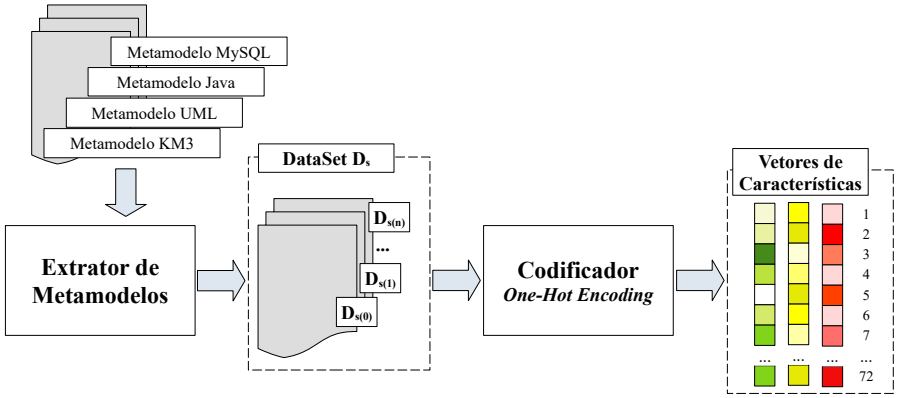


Figura 3.1: Extrator e Codificador de Metamodelos

Fonte: Elaboração própria baseada em Couto (2023) e Couto et al. (2020).

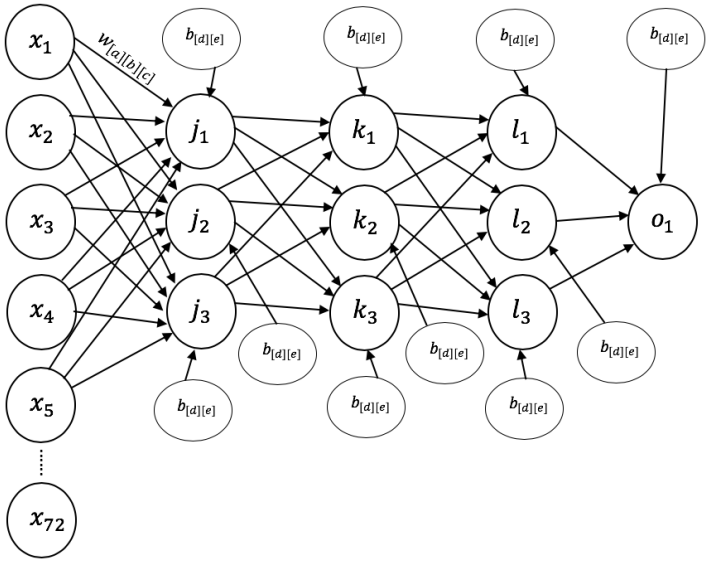


Figura 3.2: Perceptron Multicamada para a Classificação de Modelos

Fonte: Couto (2023) e Couto et al. (2020).

vetores de características compatíveis com a entrada requerida para a MLP. Adicionalmente, foram gerados automaticamente modelos com cinquenta e outros com cem elementos, onde cada elemento (classes, atributos e referências) pertence a um metamodelo previamente estabelecido do tipo MySQL, Java, UML ou KM3. Além disso, cada modelo apresenta um grau diferente de conformidade com respeito a seus metamodelos; alguns modelos com cem por cento de conformidade e outros modelos com classes, atributos e referências mistas entre diferentes metamodelos na proporção de 80%-20%, 60%-40% e 50%-50%, isto é, por exemplo, modelos com 80% dos elementos em conformidade com um metamodelo especificado e 20% em conformidade com um outro metamodelo.

Deste modo, foi feita a classificação por meio de dois perceptrons multicamadas com configurações similares à definida anteriormente, mas diferem no número de camadas intermediárias, uma MLP apresenta três camadas e a outra MLP possui cinco camadas intermediárias. Os resultados dos experimentos mostraram que a classificação de modelos que posseiam conformidade total com seus metamodelos apresentou uma acurácia igual ao cem por cento em todos

os casos (nas duas MLP e com os modelos de cinquenta e de cem elementos). De outra parte, para modelos mistos, a MLP de três camadas intermediárias atingiu a acurácia máxima de 96,3% usando modelos de cinquenta elementos, enquanto a MLP de cinco camadas intermediárias alcançou a acurácia máxima de 97,6% com modelos de cem elementos; nos dois casos, para modelos com o 80% deles do tipo MySQL e 20% em conformidade com o tipo KM3. Em geral, segundo os resultados, pode se acreditar que o perceptron multicamada de cinco camadas intermediárias é a configuração mais efetiva ao processar os modelos mistos.

Como mencionado anteriormente, Couto (2023) ampliou sua proposta com a arquitetura MCGML, na qual, além de considerar ao modelo ou metamodelo como uma estrutura composta de elementos (classes, atributos e referências), eles cumprem as especificações do formato JSON. Assim, nesta perspectiva, como pode se ver na Figura 3.3, o Tradutor de Metamodelos permitiu mapear os elementos do metamodelo em uma estrutura de grafo dirigido como se fossem vértices e arestas, supondo que um grafo inteiro é documento com subgrafos enraizados ao redor de cada nó, os quais, por sua vez são visualizados como palavras que compõem o documento.

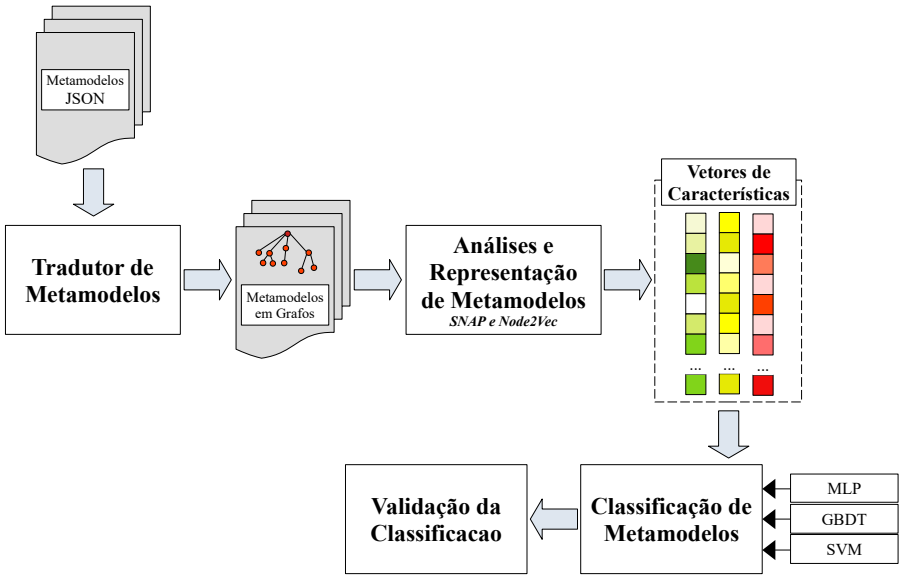


Figura 3.3: Arquitetura do *Model Classification using Graph Machine Learning*

Fonte: Elaboração própria baseada em Couto (2023).

Deste modo, foi realizado o processo de Análises e Representação de Metamodelos através do arcabouço chamado SNAP (*Stanford Network Analysis Project*) suportada pela *Stanford University* e, especialmente, pelo algoritmo *Node2Vec*, parte do SNAP, desenvolvido por (Grover e Leskovec, 2016). Neste contexto, primeiro, os grafos foram manipulados para determinar suas propriedades estruturais e, por conseguinte, analisar suas similaridades, o que envolveu usar as dependências dos papéis que os vértices desempenham em cada grafo, descobertas por meio de caminhadas aleatórias tendenciosas ou na língua inglesa *biased random walk*. Consequentemente, na segunda parte, usou-se o *Node2Vec* como método semissupervisionado de *embedding* superficial para gerar representações vetoriais dos nós de cada grafo (metamodelo) em espaços de baixa dimensionalidade, sob um enfoque que otimiza uma função objetivo personalizada mediante a descida de gradiente estocástico ou na língua inglesa *stochastic gradient descent* (SGD).

Para executar a Classificação dos Metamodelos foram usadas as representações vetoriais como recurso de entrada (vetores de caraterísticas) para o perceptron multicamada de três

camadas intermediarias (parte da arquitetura MCGML e especificado anteriormente nesta secção). Adicionalmente, para fins de comparação do desempenho, foram implementados dois algoritmos de classificação: árvores de decisão impulsionadas por gradiente ou na língua inglesa *gradient boosted decision trees* (GBDT) e o máquina de vetores de suporte ou na língua inglesa *support vector machine* (SVM).

Finalmente, na Validação da Classificação foram calculados os indicadores de desempenho: acurácia, precisão, *recall* e o *f1-score* para todos os experimentos usando a técnica de validação cruzada do tipo *tenfolds* com os mesmos recursos de entrada para os três algoritmos e configurações para classificação binária e multiclasse de quatro, seis e oito categorias. Os resultados mostraram que a arquitetura MCGML obteve uma acurácia máxima de 93,20% para a classificação binária frente ao GBDT e SVM com pontuações de 91,05% e de 72,15% respectivamente; estimações similares foram atingidas nas classificações de quatro e oito classes. No entanto, na classificação com seis classes, o GBDT alcançou métricas mais elevadas, seguidas do MCGML e SVM.

3.2 CLASSIFICAÇÃO AUTOMÁTICA DE REPOSITÓRIOS DE MODELOS E METAMODELOS USANDO MECANISMOS DO APRENDIZADO DE MÁQUINA

No contexto da engenheira de software dirigida por modelos, o problema da classificação é assinar um rótulo ao modelo ou metamodelo não rotulado. Nesse sentido, muitos estudos foram realizados para classificar ou categorizar modelos automaticamente usando principalmente abordagens ligados ao *cognifying*, como o aprendizado de máquina supervisionado, o que permite aproveitar a experiência baseada em modelos e metamodelos rotulados.

Neste sentido, Nguyen et al. (2019) propuseram uma ferramenta chamada AURORA, siglas baseadas em *AUtomated classification of metamodel RepOsitories using a neuRAl network*, o qual é um classificador supervisionado de modelos e metamodelos baseado em redes neurais. Consequentemente, a categorização de modelos facilita sua busca nos repositórios de software, promovendo seu reúso. A ferramenta foi desenvolvida sob a visão do metamodelo como um grafo, composto por um conjunto de classes, atributos e seus rótulos.

A solução envolve duas partes, a primeira versa sobre a transformação dos metamodelos do repositório em vetores de características; e a segunda parte aborda a classificação *per se*. Especificamente, como pode se ver na Figura 3.4, a transformação dos metamodelos usa um *Data Extractor* ou Extrator de Dados para gerar um vetor de características de cada metamodelo -incluindo sua classe- em um formato próprio da AURORA. Assim, nesta parte da proposta, baseada no processamento de linguagem natural, são empregados três esquemas de codificação (*uni-gram*, *bi-gram* e *n-gram*) para representar diferentes granularidades de informação sobre os termos extraídos de cada metamodelo nomeado. O *uni-gram*, o qual usa uma coleção simples de termos que não reflete a estrutura do metamodelo. O *bi-gram* ou digrama que representa a estrutura parcial do metamodelo reconhecendo as relações de contenção entre dois elementos nomeados (por exemplo, classes vs. características estruturais, pacotes vs. classes, etc.). E, finalmente, o *n-gram* que exibe a estrutura do metamodelo usando os digramas com propriedades adicionais (por exemplo: tipagem de informações, cardinalidade e multiplicidade de referência, etc.).

Além do anterior, o Extrator de Dados usa três mecanismos de normalização de termos conhecidos como: *stemming*, *lemmatization*, e *stop words*. O *stemming* permite extrair uma palavra raiz (por exemplo: *fish*, *fishes* e *fishing* são derivados de *fish*). O *lemmatization* que visa a extração de uma palavra raiz considerando seu vocabulário (por exemplo: *good*, *better* e *best* é lematizado em *good*). E, por último, o *stop words* que remove palavras que não têm significado

ou têm menos significado em comparação com outras palavras-chave (por exemplo, da frase *"How to develop a chatbot using Python"* pode se remover as palavras: *how*, *to*, *a*, e *using* para ficar com as palavras chaves: *develop*, *chatbot*, *Python*).

Ao final desta primeira parte, o Extrator de Dados implementou um *Feature Vector Generator* para gerar vetores de características com termos que têm uma frequência significativa. Neste caso, a frequência é limitada pelos pontos de corte ou na língua inglesa *cut-off*, os quais definem o número mínimo de ocorrências de um termo e, desta forma, evitam singularidades nos metamodelos. Considerando que valores de *cut-off* mais altos dão lugar a problemas de precisão.

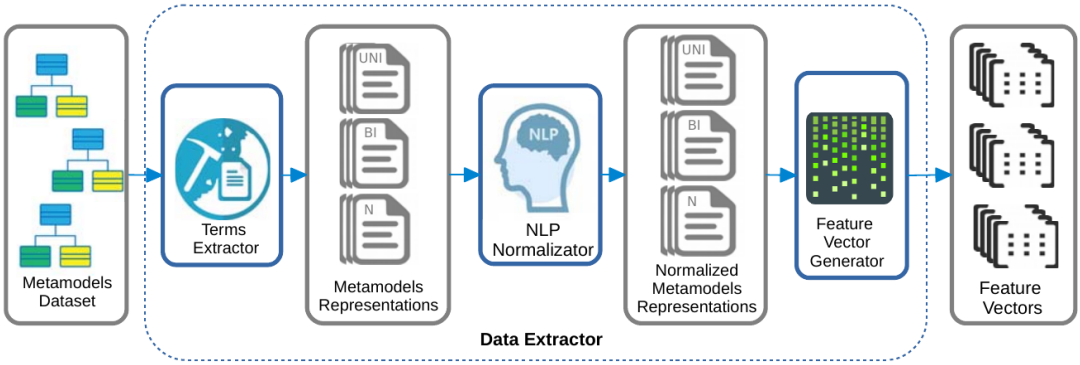


Figura 3.4: Processo de Extração de Dados da Ferramenta AURORA

Fonte: Nguyen et al. (2019).

Na segunda parte da proposta, Nguyen et al. (2019) abordaram a classificação dos metamodelos. Como pode se ver na Figura 3.5, na secção *Building*, os vetores de características -resultado do processo da primeira parte- e seus rótulos são usados pelo *Weights Calculator* para calcular os pesos e limiares ou *bias*. Assim, na secção *Deployment*, estos pesos e limiares foram usados para treinar o classificador, o que é uma rede neural do tipo *feed-forward* com conexões densas e funções de ativação do tipo sigmoide. Finalmente, a ferramenta AURORA foi avaliada empregando um conjunto de dados composto por 555 metamodelos recuperados do *GitHub* classificados em nove categorias, conseguindo uma acurácia máxima de 95,45% calculada com a técnica de validação cruzada do tipo *tenfolds*, reconhecendo como esquema de codificação ao *uni-gram* e ao valor de *cut-off* igual a dois como a configuração mais eficaz.

3.3 COMPARAÇÃO DE TÉCNICAS DE APRENDIZADO DE MÁQUINA PARA A CODIFICAÇÃO E CLASSIFICAÇÃO DE MODELOS E METAMODELOS DE RÓTULO ÚNICO

O aprendizado de máquina disponibiliza um conjunto de algoritmos para a classificação, incluindo alguns algoritmos de regressão usados como classificadores (Géron, 2023; Jiang, 2021). Complementarmente, cada sistema de aprendizado de máquina precisa de dados adequados como fontes de entradas, os quais podem ser gerados por meio das técnicas de codificação como parte do processo do ML. Nesta perspectiva, López et al. (2022) apresentaram um estudo no qual compararam diferentes técnicas de codificação e de classificação de modelos e metamodelos, empregando a abordagem *cognifying* baseado no aprendizado de máquina supervisionado para rotular automaticamente modelos armazenados em repositórios, e desta maneira, possibilitando seu reúso.

De acordo com o processo tradicional do aprendizado de máquina, a comparação sistemática foi desenvolvida por meio de quatro etapas: a coleta ou escolha de dados, a preparação

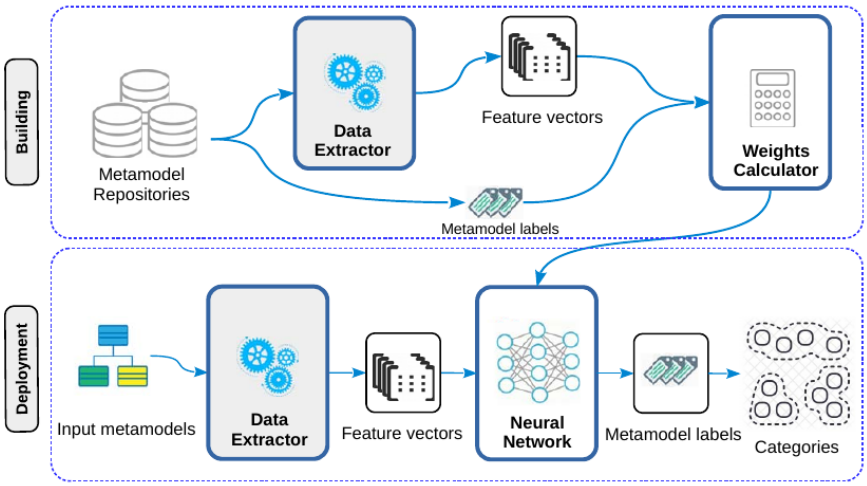


Figura 3.5: Arquitetura da Ferramenta AURORA

Fonte: Nguyen et al. (2019).

de dados, geração de vetores de características e, finalmente, o treinamento e avaliação do modelo. Neste sentido, como pode se ver a Figura 3.6, na primeira etapa, a coleta de dados ou na língua inglesa *Data Gathering* apresentou-se o conjunto de dados chamado ModelSet composto por cerca de cinco mil metamodelos Ecore e outros cinco mil modelos UML, rotulados com a categoria que reflete o domínio de aplicação do modelo. Seguidamente, na segunda etapa, a preparação de dados ou na língua inglesa *Data Preparation* possibilitou a detecção e o deslocamento dos modelos duplicados para diminuir o viés na avaliação das técnicas de ML; ainda, o conjunto de dados original foi mantido para comparar o efeito dos duplicados. Além disso, foram filtrados as categorias com menos de dez modelos.

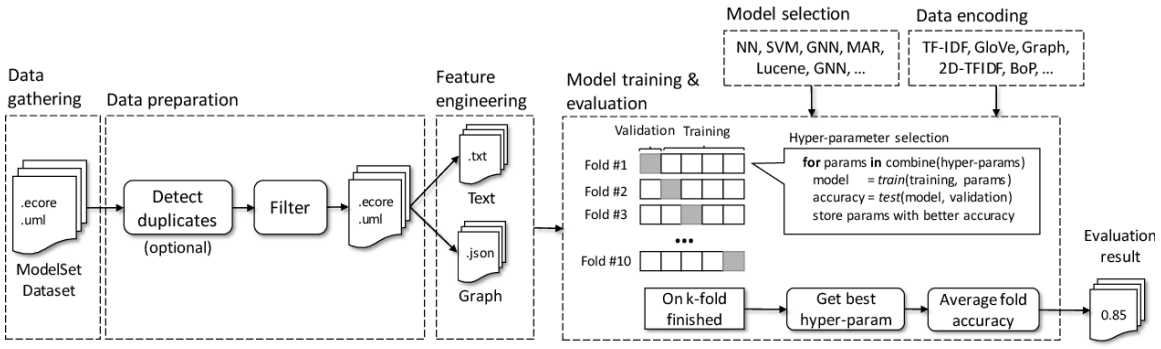


Figura 3.6: Metodologia para a Comparação de Técnicas de Aprendizado de Máquina

Fonte: López et al. (2022).

Na terceira etapa, a engenharia de características o na língua inglesa *Feature Engineering* foi usada para codificar os modelos e gerar os vetores de características para cada classificador supervisionado de ML, incluindo k-vizinhos mais próximos ou na língua inglesa *K-nearest neighbors* (KNN), bayes naive, máquina de vetores de suporte ou na língua inglesa *support vector machine* (SVM) e redes neurais como: *feedforward neural network* (FFNN), *convolutional neural network* (CNN) e *graph neural network* (GNN). Neste sentido, a geração foi efetuada sob duas abordagens: umo baseada na técnica *bags of words* (BoW) e, outro, que representa os modelos em grafos. Assim, conforme a primeira abordagem, foram extraídos um conjunto de

palavras de cada modelo, considerando a frequência de aparecimento cada palavra, com os quais foram gerados os vetores de valores reais usando três algoritmos:

- *Term Frequency - Inverse Document Frequency* (TF-IDF) onde cada conjunto de palavras é representado por um vetor de alta dimensão (uma dimensão por palavra diferente do vocabulário global).
- *Global Vectors for Word Representation* (GloVe) é um método de *embeddings* de palavras, em que cada palavra do conjunto tem um vetor associado, considerando que palavras com semânticas semelhantes têm vetores geometricamente próximos. Assim, cada conjunto de palavras é representado em um vetor de características com dimensões que contêm a média dos vetores associados às palavras do conjunto.
- 2D TF-IDF onde os vetores gerados pelo TF-IDF são divididos em sequências de células consecutivas, os quais são empilhados verticalmente para gerar matrizes 2D.

Igualmente, nesta terceira etapa foi implementada uma segunda abordagem, acreditando que a representação típica de um modelo é semelhante ao grafo. Desta maneira, cada objeto do modelo foi representado como um nó (incluindo um identificador do objeto e o nome de sua metaclasse) e cada referência foi mapeada como uma aresta do grafo. Armazenando esta nova representação em um arquivo com formato JSON, o que permitiu manter as propriedades dos modelos na topologia do grafo gerado.

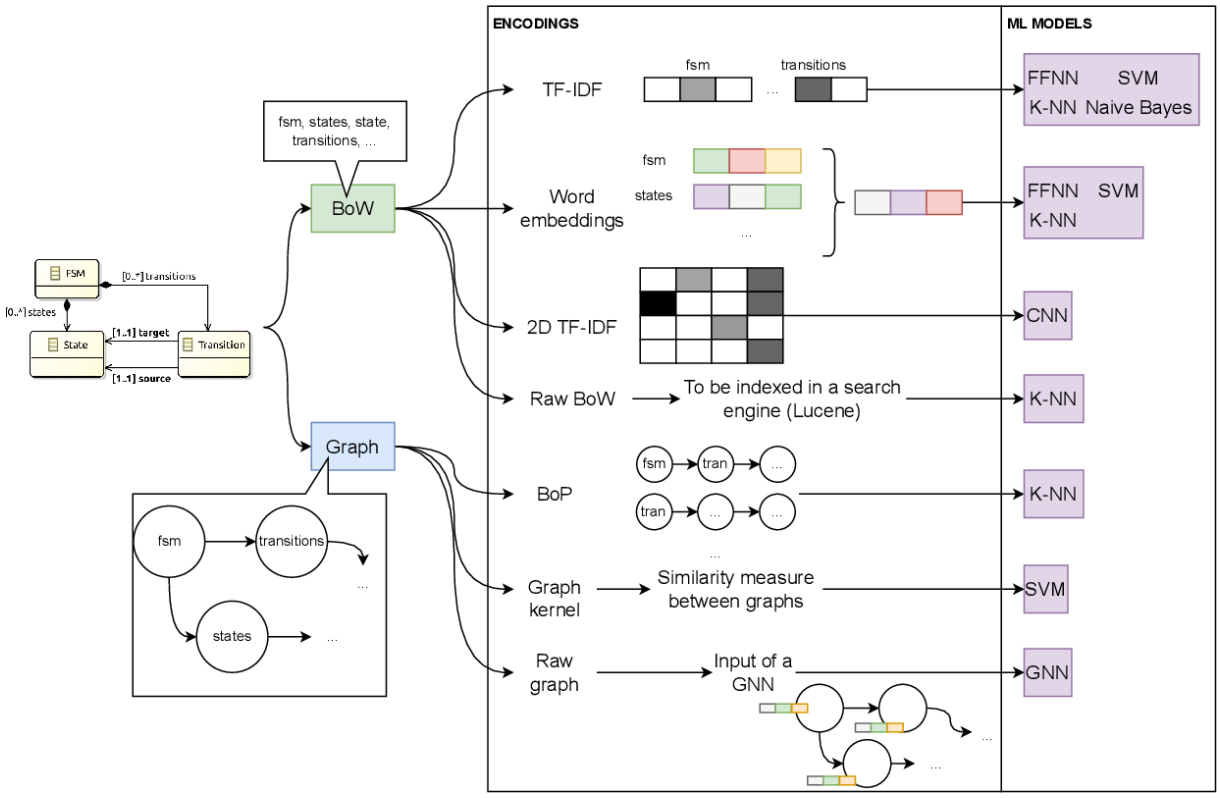


Figura 3.7: Combinações das Técnicas de Codificação e de Classificação de Modelos

Fonte: López et al. (2022).

Por último, na quarta etapa, o treinamento e avaliação do modelo ou na língua inglesa *Model Training and Evaluation* foram desenvolvidas sistematicamente, considerando as

combinações das técnicas de codificação e de classificação de modelos e metamodelos baseadas na Figura 3.7, e junto a técnica de validação cruzada do tipo *tenfolds* com o indicador de acurácia balanceada. Neste contexto, os resultados demostraram que as técnicas de codificação de major sucesso foram o TF-IDF seguido do GloVe, igualmente, os melhores classificadores foram o FFNN e SVM. A combinação mais eficaz para o conjunto de dados de metamodelos Ecore, com e sem duplicados, foi o FFNN junto com TF-IDF, com acurácias balanceadas de 89,8511% e 82,4972% respectivamente. Do mesmo modo, para os modelos UML, a melhor combinação foi o SVM com GloVe, em ambos os casos, com e sem duplicados, com um desempenho de 87,3906% e 77,5893% respectivamente. Além disso, pode se acreditar que os experimentos com o conjunto de dados ModelSet com modelos duplicados alcançaram melhores resultados.

4 PROPOSTA DA PESQUISA

Neste capítulo são apresentados a abordagem e arquitetura da proposta de pesquisa e as estrategias usadas para atingir os objetivos específicos e, consequentemente, atingir o objetivo geral, que visa classificar metamodelos de um repositório usando uma abordagem de aprendizado de máquina em grafos.

4.1 ABORDAGEM E ARQUITETURA DA PROPOSTA

O método proposto nesta dissertação de mestrado, permite predizer a categoria ou a classe dos metamodelos do repositório, isto é, prever se o metamodelo é, por exemplo, do tipo: Máquina de Estados, Rede de Petri, Biblioteca, Modelagem, Diagrama de Classes, etc. Para tal fim, como pode se ver na Figura 4.1, os metamodelos do repositório, mapeados como grafos dirigidos em formato JSON (*JavaScript Object Notation*), foram representados em espácios vetoriais de baixa dimensionalidade por meio de um método de *embedding* de grafos superficiais chamado Graph2Vec, para depois classificá-los usando três algoritmos de aprendizado de máquina supervisionados: árvores de decisão impulsionadas por gradiente, máquina de vetores de suporte e o perceptron multicamada. Coerentemente, ao processo do aprendizado máquina, todos os experimentos foram avaliados através da validação cruzada do tipo *tenfolds*.

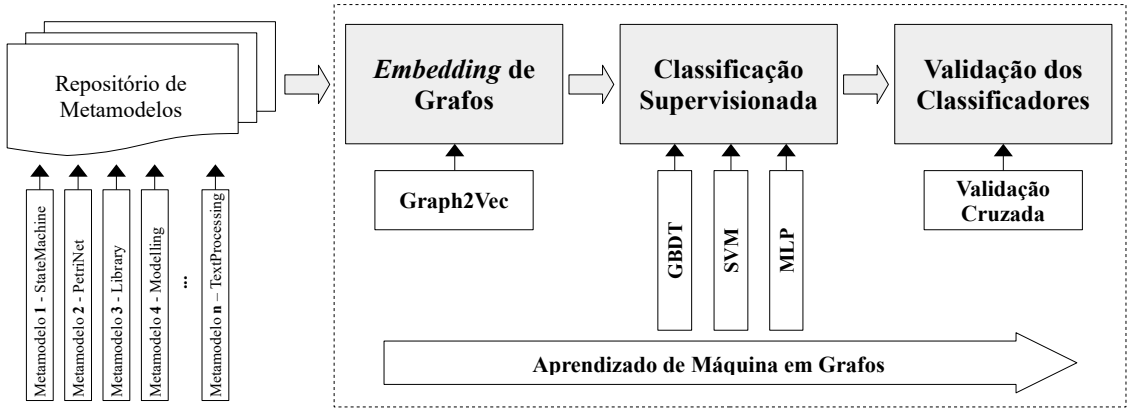


Figura 4.1: Arquitetura da Proposta

Fonte: Elaboração própria.

4.2 REPOSITÓRIO DE METAMODELOS SOB UMA VISÃO DE GRAFOS

Para atingir os objetivos desta pesquisa, foram usados os metamodelos do repositório chamado ModelSet, em sua versão 0.9.3, disponibilizado por López et al. (2021, 2022). O ModelSet é um conjunto de metamodelos coletados, que foram manualmente rotulados e curados para possibilitar a aplicação de métodos de aprendizado de máquina para resolver problemas próprios da MDSE. Neste repositório, os domínios de aplicação dos metamodelos estão relacionados à modelagem conceitual e DSLs, categorizados, em classes desbalanceadas, como por exemplo: Máquina de Estados, Rede de Petri, Biblioteca, Modelagem, Diagrama de Classes, etc.

O ModelSet disponível no endereço eletrônico `http://modelset.github.io`, acessado em 24 de julho de 2023, está composto de 5466 modelos Ecore (coletados do GitHub) e 5120 modelos UML (coletados do GenMyModel), perfazendo um total de 10586 modelos. Todos os metamodelos estão armazenados em três formatos de arquivos: texto, XMI (XML Metadata Interchange) e JSON. O formato tipo JSON foi adicionado ao ModelSet no 2022 e contêm os mapeamentos dos metamodelos em forma de grafos, obtidas visando que um metamodelo é um conjunto de classes, atributos e referências. Desta forma, cada metamodelo foi transformado em uma coleção de pares chave-valor ou na língua inglesa *key-value*, o que permitiu preservar as propriedades estruturais e relacionais dos metamodelos (López et al., 2022).

Desta forma, no contexto do aprendizado de máquina em grafos, os metamodelos do repositório ModelSet serviram como fonte de entrada para o processo de *embedding* desenvolvido por meio do método Graph2Vec.

4.3 *EMBEDDING* DE GRAFOS A REPRESENTAÇÕES VETORIAIS DE LONGITUDE FIXA

No contexto da pesquisa, o método de *embedding* de grafos permitiu mapear automaticamente os metamodelos -no formato JSON- em representações ou espaços vetoriais de baixa dimensionalidade, composto por vetores compactos, densos e de valores numéricos contínuos. Dos métodos de *embeddings* descritos anteriormente e segundo os trabalhos analisados, os métodos superficiais ou na língua inglesa *shallow embedding methods* são os algoritmos mais usados na classificação ou agrupamento de modelos no contexto da MDSE.

Assim, na pesquisa, se usou o algoritmo Graph2Vec, proposto por (Narayanan et al., 2017), o qual é um método que usa redes neurais para aprender representações distribuídas de maneira não supervisionada, transdutiva e, também, agnóstica em relação à tarefa. Desta forma, o Graph2Vec permite gerar vetores de baixa dimensão e de longitude fixa, que preservem as propriedades topológicas do grafo inteiro (relacionadas à estrutura do grafo e as interconexões entre os nós). Segundo Liu e Zhou (2020) as representações distribuídas possibilita que objetos linguísticos com distribuições semelhantes têm significados semelhantes, favorecendo a aprendizagem semântica. A visão agnóstica do algoritmo é devido a que não se aproveita nenhuma informação específica da tarefa (por exemplo, rótulos de classe dos modelos), portanto, as representações fornecidas são genéricas, o que propicia seu uso nas tarefas analíticas que envolvem grafos inteiros.

Segunda a proposta do Narayanan et al. (2017), como pode se ver no Algoritmo 1, o objetivo é tentar aprender representações ou espaços vetoriais de dimensão δ de todos os grafos no conjunto de dados $\mathcal{G}$ em ϵ épocas, considerando que cada grafo G é apresentado como (N, E, λ), onde N simboliza os nós do grafo, E as arestas e λ representa uma função que procura um rótulo único para cada nó $n \in N$. Assim, para atingir o objetivo, a estrategia do Graph2Vec é gerar recursivamente subgrafos enraizados ao redor de cada nó do grafo, atribuindo a cada subgrafo um novo rótulo exclusivo usando o método de Weisfeiler-Lehman (como pode se ver no Algoritmo 2). Coerente ao modelo Skip-Gram (Mikolov et al., 2013) com amostragem negativa, os rótulos dos subgrafos extraídos são usados para predizer o contexto semântico de cada palavra em relação às outras; e, desta forma, baseado no Doc2Vec (Le e Mikolov, 2014) o documento é analisado como um grafo inteiro composto de subgrafos enraizados a fim de conseguir suas representações ou espaços vetoriais densos, de baixa dimensão e de longitude fixa.

Algoritmo 1 Graph2Vec ($\mathcal{G}$, D, δ, ϵ, α):

Require: $\mathcal{G} = \{G_1, G_2, .., G_n\}$: Conjunto de grafos de forma que cada grafo $G_i = (N_i, E_i, \lambda_i)$
D: Grau máximo de subgrafos enraizados a ser considerados para o processo de *embeddings*, o que produzirá um vocabulário ao nível de subgrafos, $SG_{vocab} = \{sg_1, sg_2, ..\}$ de todos os grafos em $\mathcal{G}$
δ: Número de dimensões ou tamanho de *embedding*
ϵ: Número de *epochs* ou épocas
α: *Learning rate* ou taxa do aprendizado
Ensure: Matriz de representações ou espaços vetoriais dos grafos $\Phi \in \mathbb{R}^{|\mathcal{G}| \times \delta}$

Inicialização: Amostra Φ de $\mathbb{R}^{|\mathcal{G}| \times \delta}$
for $e = 1$ **to** ϵ **do**
  $\mathbb{G}$ = SHUFFLE ($\mathcal{G}$)
  for each $G_i \in \mathbb{G}$ **do**
    for each $n \in N_i$ **do**
      for $d = 0$ **to** D **do**
        $sg_n^{(d)} := GetWLSubGraph(n, G_i, d)$
        $J(\Phi) = -\log Pr(sg_n^{(d)} | \Phi(G))$
        $\Phi = \Phi - \alpha \frac{\partial J}{\partial \Phi}$
      end for
    end for
  end for
end for
return Φ

Algoritmo 2 GetWLSubGraph (n, G, d):

Require: n: Nó que atua como a raiz do subgrafo
$G = (N, E, \lambda)$: Grafo do qual o subgrafo vai ser extraído
d: Grau dos vizinhos a serem considerados para extrair o subgrafo
Ensure: $sg_n^{(d)}$: Subgrafo enraizado de grau d ao redor do nó n

$sg_n^{(d)} = \{\}$
if $d = 0$ **then**
  $sg_n^{(d)} := \lambda(n)$
else
  $\mathcal{N}_n := \{n' | (n, n') \in E\}$
  $M_n^{(d)} := \{GetWLSubGraph(n', G, d-1) | n' \in \mathcal{N}_n\}$
  $sg_n^{(d)} := sg_n^{(d)} \cup GetWLSubGraph(n, G, d-1) \oplus sort(M_n^{(d)})$
end if
return $sg_n^{(d)}$

O Graph2Vec apresenta melhorias significativas na eficiência e eficacia nas tarefas de classificação e agrupamento -ao nível do grafo inteiro- em relação a outras abordagens de *embeddings* baseadas em subestruturas ou em *kernels* de grafos. Além disso, este algoritmo pode ser usado sob abordagens de aprendizado supervisionado e não supervisionado, com grafos de tamanhos arbitrários (Narayanan et al., 2017).

Desta forma, nesta pesquisa, as representações ou espaços vetoriais dos metamodelos, gerados por meio do Graph2Vec, foram usados como fontes de entrada para os algoritmos

classificadores, conforme ao contexto de aprendizado de máquina em grafos, visando a predição da categoria ou a classe correspondente de cada metamodelo.

4.4 ALGORITMOS DE APRENDIZADO DE MÁQUINA PARA A CLASSIFICAÇÃO SUPERVISIONADA

Coerentemente aos objetivos da pesquisa, as representações vetoriais dos metamodelos foram usadas como dados de treinamento para os classificadores, sob a visão do aprendizado de máquina supervisionado, tentando predizer, ainda com um erro associado, a categoria ou a classe única do metamodelo, isto é, prever se o metamodelo é do tipo: Máquina de Estados, Rede de Petri, Biblioteca, Modelagem, Diagrama de Classes, etc. Assim, no contexto da pesquisa, foram usados três algoritmos de classificação supervisionados: as árvores de decisão impulsionadas por gradiente, a máquina de vetores de suporte e o perceptron multicamada, os quais são os algoritmos mais frequentemente aplicados na classificação de modelos e metamodelos no contexto da MDSE (Couto, 2023; López et al., 2022; Couto et al., 2020; Nguyen et al., 2021, 2019).

As árvores de decisão impulsionadas por gradiente ou na língua inglesa *gradient boosted decision trees* (GBDT), proposto por Friedman no 2001 (Nguyen et al., 2021), é um modelo de *ensemble* baseado em várias árvores de decisão superficiais treinadas sequencialmente para potencializar a capacidade de predição do algoritmo. Assim, como pode se ver na Figura 4.2, cada árvore é treinada para predizer e tentar minimizar os erros residuais da árvore anterior usando o gradiente (Grigoriev, 2021; Sarker, 2021). Consequentemente, o GBDT tem uma capacidade de predição igual à soma ponderada das previsões feitas pelas árvores do *ensemble* (Ishfaq et al., 2022). Neste sentido, o GBDT tende a cometer cada vez menos erros e consegue um bom desempenho global, em termos de eficiência, interpretabilidade e generalização (Grigoriev, 2021; Ke et al., 2017), além de evitar o *overfitting* por meio da técnica de parada antecipada ou na língua inglesa *early stopping*.

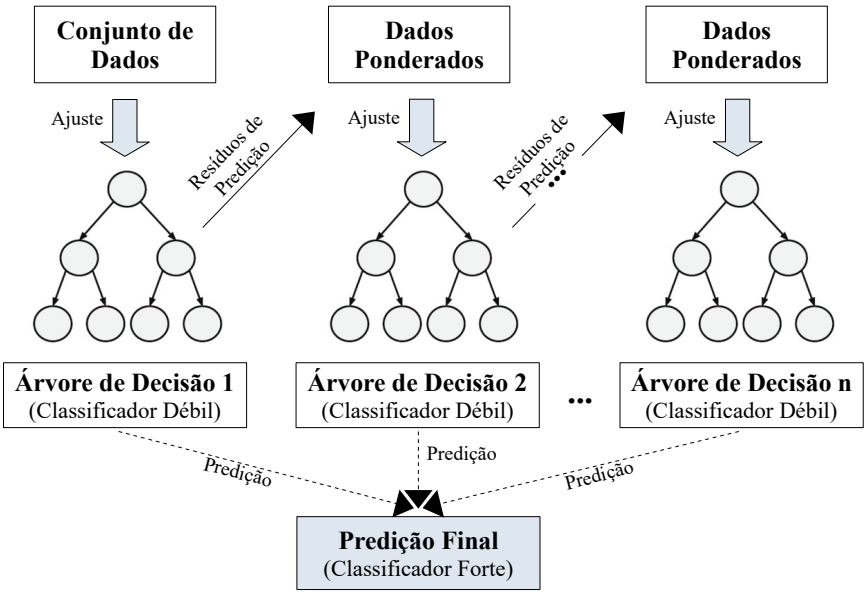


Figura 4.2: Árvores de Decisão Impulsionadas por Gradiente

Fonte: Elaboração própria baseada em Grigoriev (2021).

A máquina de vetores de suporte ou na língua inglesa *support vector machine* (SVM), proposta por Cortes e Vapnik no 1995 (Izbicki e dos Santos, 2020), está baseada na teoria do aprendizado estatístico e na análise discriminante (Berzal, 2018). O algoritmo representa os dados de treinamento como pontos rotulados em um espaço multidimensional, onde o número de dimensões varia de acordo com o número de características dos vetores de entrada. Assim, o SVM pode construir um ou mais hiperplanos que melhor dividem os pontos entre as classes (como pode se ver na Figura 4.3). Conceitualizando ao hiperplano como uma superfície de decisão, que optimizado permite maximizar a margem entre os pontos. Desta forma, pode se acreditar que quanto maior a margem, menor o erro da generalização do classificador (Jiang, 2021; Nguyen et al., 2021; Sarker, 2021). O SVM é eficaz em espaços de alta dimensão e pode se comportar de forma diferente com base em diferentes funções matemáticas conhecidas como *kernel* (Géron, 2023; Sarker, 2021).

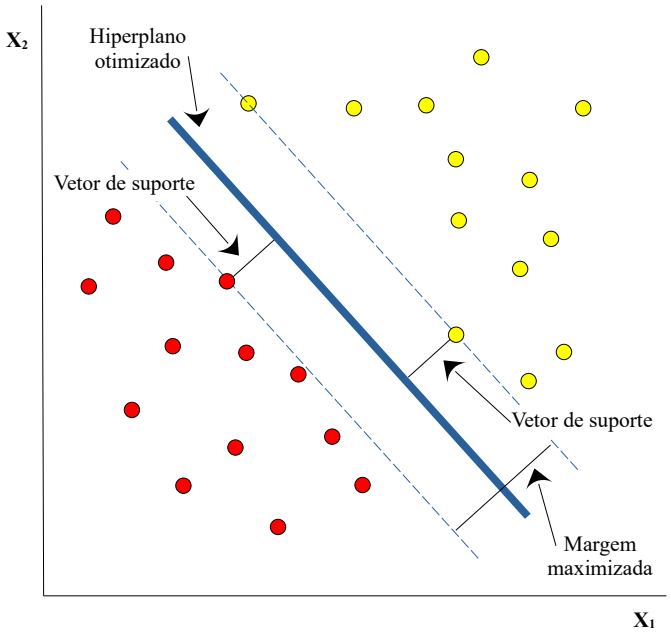


Figura 4.3: Máquina de Vetores de Suporte

Fonte: Elaboração própria baseada em Jiang (2021).

O perceptron multicamada ou na língua inglesa *multiple layer perceptron* (MLP), baseada na proposta de Rosenblatt no 1957 (Géron, 2023), é uma das arquiteturas mais simples de uma rede neural artificial. A estrutura de uma MLP está definida por camadas interconectadas de neurônios, incluindo uma camada de entrada, uma ou mais camadas ocultas e uma camada de saída (Abhishek e Abdelaziz, 2023; Géron, 2023). Por sua vez, cada neurônio realiza uma operação com suas entradas ponderadas usando uma função de ativação e o resultado é transmitido a todos os neurônios da próxima camada. A função de ativação esta baseada em uma percepção não linear, o que permite que a MLP possa aprender e modelar relações muito complexas. O fluxo da informação em um MLP é da camada inicial até a final, por meio de um algoritmo conhecido como *feed forward propagation*, enquanto os ajustes dos pesos e dos vieses para minimizar a função de perda são feitas da camada final às camadas anteriores conforme a um algoritmo chamado *back propagation* (Berzal, 2018).

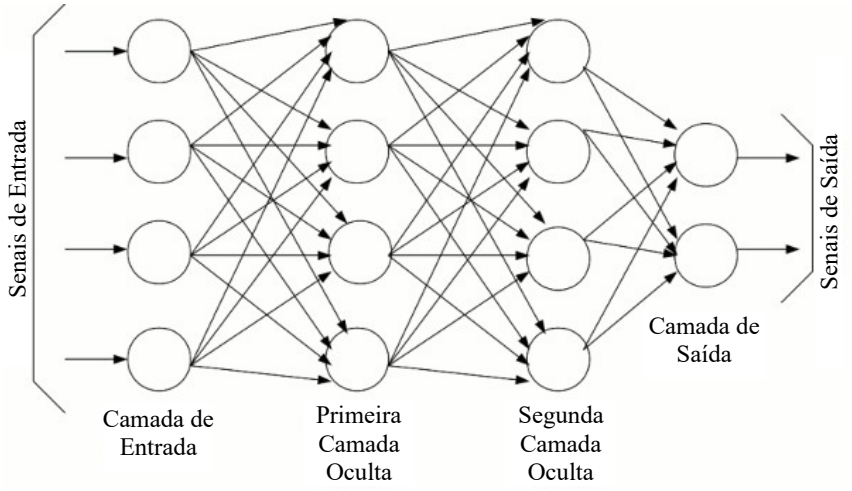


Figura 4.4: Perceptron Multicamada

Fonte: Adaptada de Berzal (2018).

4.5 VALIDAÇÃO DA PROPOSTA

Como última parte da pesquisa e, coerentemente, ao processo do aprendizado máquina em grafos, foram avaliados os desempenhos dos três classificadores por meio da validação cruzada do tipo *tenfolds* ou na língua inglesa *tenfolds cross-validation*, o qual é uma técnica baseada em *k-folds* para $k = 10$, sendo k o número de subconjuntos, partes ou *folds* em que o conjunto de dados é dividido. Pode se acreditar que a validação cruzada do tipo *tenfolds* é a técnica mais frequentemente usada nos trabalhos analisados, como em Couto (2023) e Couto et al. (2020), López et al. (2022) e em Nguyen et al. (2021, 2019). Além disso, a validação cruzada pode ser usada para evitar o *overfitting* e, também, para comparar o desempenho de diferentes algoritmos de aprendizado de máquina com o mesmo conjunto de dados (Lee, 2019), e o tipo *tenfold* apresenta a maneira mais eficiente de obter uma boa estimativa em relação a outros valores para k (Berzal, 2018).

Assim, na pesquisa, as representações vetoriais dos metamodelos foram divididos aleatoriamente em dez subconjuntos de tamanhos similares, de forma que, cada algoritmo classificador teve dez iterações, onde cada iteração cria um preditor usando uma combinação diferente de nove partes para treinamento e uma parte para validação (como pode se ver na Figura 4.5). Considerando que o ModelSet é um repositório com classes desbalanceadas, por sugestão do López et al. (2022, 2023), o desempenho do classificador foi medida pela métrica chamada acurácia balanceada ou na língua inglesa *balanced accuracy*. Ao final da execução, a estimativa da métrica é a média das pontuações obtidas em suas interações.

4.6 SINGULARIDADES DA PROPOSTA

Dos trabalhos relacionados foram extraídos algumas contribuições e boas práticas referidas à classificação supervisionada de modelos e metamodelos sob uma perspectiva de engenharia de software dirigida por modelos e do aprendizado de máquina em grafos, os quais permitiram estruturar e contextualizar esta proposta de pesquisa com major clareza.

Do trabalho relacionado de Couto (2023), que visa sobre a classificação de modelos desestruturados usando metamodelos específicos e aprendizado de máquina, foi considerado a possibilidade de mapear -em espaços vetoriais- os nós dos grafos (modelo) por meio do método

Figura 4.5: Validação Cruzada do tipo *TenFolds*

Fonte: Elaboração própria baseada em López et al. (2022) e Grigoriev (2021).

de *embedding Node2Vec* proposto por Grover e Leskovec (2016) como método de *embedding*. Não obstante, nesta pesquisa foi implementado o algoritmo *Graph2Vec* de Narayanan et al. (2017) como método não supervisionado de *embedding* superficial, o qual permitiu conseguir representações distribuídas de cada metamodelo sob um abordagem de grafo inteiro.

Em relação aos trabalhos de Nguyen et al. (2021, 2019), onde propuseram e avaliaram a ferramenta AURORA (*AUtomated classification of metamodel RepOsitories using a neuRAl network*), é possível destacar o processo ou na língua inglesa *pipeline* da classificação de metamodelos e, consequentemente, a viabilidade do projeto. Desta maneira, o processo nesta pesquisa foi proposto em duas etapas: a primeira referida ao *embedding* ou representação de metamodelos em um espaço vetorial de baixa dimensão e, a segunda etapa, que visa sobre a classificação *per se* usando três algoritmos supervisionados, incluindo a avaliação dos desempenhos através da técnica de validação cruzada do tipo *tenfolds*.

Congruentemente com esta linha de pesquisa, o trabalho de López et al. (2022), o qual comparou diferentes técnicas de codificação e de classificação de modelos e metamodelos em um contexto de aprendizado de máquina supervisionada, permitiu acreditar que a forma típica de analisar um modelo ou metamodelo é como uma estrutura conformada de classes ou objetos, atributos e referências; de esta forma, os modelos podem ser representados sob um abordagem em grafos. Deste trabalho, também foi usado o repositório chamado ModelSet, proposto por López et al. (2021), conformado por 5466 metamodelos curados do tipo Ecore, representados em formato de grafos e armazenados em estruturas JSON.

5 EXPERIMENTOS

Neste capítulo são apresentados os resultados dos experimentos que validam a efetividade da proposta para classificar os metamodelos de um repositório. São mostrados os detalhes do repositório, o uso do método de *embedding* superficial chamado *Graph2Vec* e os três algoritmos de aprendizado de máquina em grafos, os quais foram avaliados por meio da validação cruzada.

5.1 DESCRIÇÃO DO REPOSITÓRIO

Como descrito no Capítulo IV, nesta pesquisa foi usado o repositório de metamodelos chamado ModelSet proposto por López et al. (2021, 2022). Especificamente, para os experimentos, usaram-se apenas os metamodelos do tipo Ecore do repositório. Segundo López et al. (2021), no ModelSet, os metamodelos Ecore apresentam maior variabilidade em termos de especificação de domínios e de propósito, além de ser os metamodelos mais usados em contextos da MDSE em comparação ao formato UML.

Segundo López et al. (2022), todos os metamodelos do repositório ModelSet foram analisados como uma estrutura composta de elementos como classes, atributos e referências. Desta forma, cada metamodelo foi representado como um multigrafo dirigido no ModelSet; onde cada objeto ou classe do modelo foi mapeado como um nó com seus atributos (incluindo um identificador de nó, um tipo do elemento Ecore e um nome se disponível) e, igualmente, cada referência foi especificada como uma aresta do grafo. No Anexo A pode se ver um exemplo da representação de um metamodelo como um grafo.

Nesta perspectiva, é essencial destacar que a convenção de nomes dos metamodelos Ecore e, por conseguente, das representações em grafos armazenadas em formato JSON, consideram a sensibilidade entre maiúsculas e as minúsculas. Portanto, as analises das representações em grafos conservaram essa distinção, a fim de manter as propriedades originais dos modelos.

Originalmente, o repositório ModelSet contém 5466 metamodelos do tipo Ecore representados em grafos, classificados em 222 classes ou categorias, estatisticamente desbalanceadas. Por sugestão do López et al. (2022, 2023), excluíram-se dos experimentos os metamodelos as classes *Dummy* e *Unknown*, por conter, respectivamente, modelos criados com fins de teste ou com pouca certeza do seu domínio. No mesmo sentido, foram retiradas as classes que possuem menos de dez metamodelos. Na Figura 5.1 apresentam-se as primeiras classes do ModelSet.

Na tabela 5.1, pode-se observar a descrição do repositório ModelSet, depois das exclusões especificadas. O repositório com duplicados tem 4155 metamodelos divididos em 67 categorias ou classes e, de outra parte, o repositório sem duplicados (com modelos únicos) apresenta 2067 metamodelos classificados em 48 categorias. A diferença entre as quantidades de metamodelos evidencia que o ModelSet contém um 50.25% (1-2067/4155) de modelos duplicados e quase duplicados. Coerentemente, a variação do número de classes pressupõe que há metamodelos duplicados que suas ocorrências por categoria diminuíram para menos de dez, portanto, não foram considerados nas análises.

Para determinar a duplicidade ou quase duplicidade de metamodelos, López et al. (2022) compararam as representações em texto entre metamodelos a través da adaptação do algoritmo proposto por Allamanis (2019). Este algoritmo usa a técnica de processamento de linguagem natural conhecida como tokenização (para criar multiconjuntos de palavras para cada metamodelo) e o coeficiente de Jaccard (para a detecção de similitude).

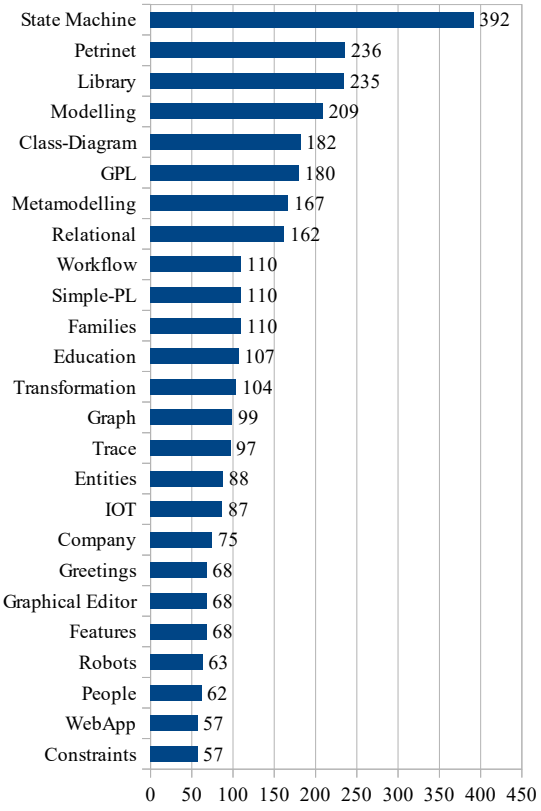


Figura 5.1: Frequência das Primeiras Classes do Repositório ModelSet

Fonte: Elaboração própria baseada no ModelSet (López et al., 2022)

Tabela 5.1: Descrição do Repositório ModelSet

	Repositório com Duplicados	**Repositório sem Duplicados**
Número de metamodelos	4155	2067
Número de categorias o classes	67	48
Média de elementos	218.96	142.62
Média de classes	28.72	19.16
Média de atributos	16.68	12.68
Média de referências	29.26	21.66
Média de pacotes	1.50	1.34

Fonte: Elaboração própria baseada no ModelSet (López et al., 2022).

5.2 AMBIENTE DE EXPERIMENTAÇÃO

Todos os experimentos foram desenvolvidos acessando remotamente nas servidoras do Departamento de Informática da UFPR. Especialmente, foi usada a servidora Zara, que tem um processador Intel®Xeon®CPU E5-2670 de arquitetura de 64 bits com 16 núcleos físicos e memória RAM de 126 GB sob o sistema operacional Linux Mint LMDE (Linux Mint Debian Edition) na versão 5 Elsie. Além disso, foi usado um ambiente virtual de programação com a linguagem Python 3.10.12 junto com as bibliotecas NumPy 1.26.4, Pandas 2.2.2, Scikit-Learn

1.5.1, NetworkX 3.3, ModelSet-Py 0.2.1, Karate Club 1.3.4, XGBoost 2.1.1, SciPy 1.10, GenSim 4.2.0 e o editor GNU nano 5.4.

5.3 EXPERIMENTOS DE COMPREENSÃO

Nesta secção são apresentados os experimentos iniciais para compreender a implementação (funcionamento e uso) do método de *embedding* superficial chamado Graph2Vec juntamente com o algoritmo de aprendizado de máquina supervisionado conhecido como máquina de vetores de suporte (SVM).

Das muitas codificações do algoritmo Graph2Vec no Github, na pesquisa implementou-se a proposta de Rozemberczki et al. (2020), por ser a mais usada e atualizada. Esta codificação do Graph2Vec faz parte do *framework* de código aberto chamado Karate Club, o qual está disponível no endereço eletrônico `https://github.com/benedekrozemberczki/karateclub` acessado em 24 de outubro de 2023. O Karate Club, desenvolvido em Python integrado com a biblioteca NetworkX, implementa uma variedade de métodos de aprendizado de máquina não supervisionados aplicáveis a estruturas de dados em grafos. Especialmente, o *framework* se destaca na detecção de comunidades e *embeddings* de nós e de grafos inteiros.

Especificamente, o Graph2Vec do Karate Club é codificado como método de *embedding* ao nível de grafo inteiro baseado na proposta de Narayanan et al. (2017). O algoritmo cria recursivamente subestruturas -em forma de arvore- para cada nó do grafo através do algoritmo de Weisfeiler-Lehman. Utilizando essas subestruturas, uma matriz de coocorrência do grafo inteiro é decomposta para gerar representações vectoriais densas e de longitude fixa em espaços latentes de baixa dimensionalidade. Além de usar a estrutura do grafo, o Graph2Vec pode usar um atributo específico dos nós.

Do anterior, pode-se acreditar que os principais parâmetros no Graph2Vec do Karate Club são:

- *wl_iterations*: Número de iterações para desestruturar o grafo usando Weisfeiler-Lehman. O valor predefinido é dois, embora poderia ser usado o grau de centralidade do grafo.
- *dimensions*: Dimensão ou tamanho do vetor que representa ao grafo inteiro. O valor predefinido é igual a 128.
- *use_node_attribute*: Possibilita o uso de um atributo do nó. O valor predefinido é *None*. Em Python, *None* é um tipo especial que representa a ausência de um valor.

Adicionalmente, a fim de melhorar o desempenho do processo de *embedding*, o Graph2Vec do *Karate Club* pode ser implementado com os seguintes hiperparâmetros:

- *workers*: Número de núcleos ou unidades de processamento. Valor predefinido é 4. Em todos os experimentos, foram usados todos núcleos disponíveis da servidora Zara do Departamento de Informática.
- *epochs*: Número de *epochs* ou épocas, refere-se ao número de iterações em que o algoritmo processa o conjunto de dados completo. Valor predefinido é 10.
- *learning_rate*: Taxa de aprendizado, representa a rapidez com que os pesos do algoritmo são atualizados em cada iteração. Valor predefinido é 0.025. Em geral, este valor predefinido atingiu resultados mais ótimos e foi usado em todos os experimentos.

- *erase_base_features*: Remove as características originais dos nós do grafo. Valor predefinido é *False*. Em todos os experimentos, usou-se o valor predefinido por ter demonstrado bons desempenhos.

Uma limitação ao implementar o Graph2Vec do Karate Club, como método de *embedding*, é a falta de uma métrica do desempenho, seja como uma função de utilidade ou de custo. Então, não é possível medir a eficacia em que os vetores gerados preservam as propriedades especificadas dos grafos originais (metamodelos Ecore do ModelSet). Não obstante, pode-se acreditar que uma adequada representação acrescenta a eficiência computacional, a capacidade de generalização e, até, a interpretação e visualização de dados.

Na pesquisa, considerando que as representações vetoriais dos metamodelos serão usadas como dados de treinamento para o processo de aprendizado de máquina, o que, por sua vez, envolve a aplicação de um mecanismo de avaliação de desempenho. Desta forma, com o objetivo de conseguir uma métrica de desempenho para o método de *embedding*, implementou-se o Graph2Vec do Karate Club junto com o algoritmo de classificação *SVC*[1] de Scikit-Learn, o qual usa a máquina de vetores de suporte para classificação. Então, através da validação cruzada do tipo *tenfolds* foi possível obter a pontuação da métrica chamada acurácia balanceada, o qual é usada como um indicador indireto do desempenho do *embedding*.

Todos os experimentos desta secção foram implementados usando seus hiperparâmetros com valores predefinidos. Incluindo as especificações para o Graph2Vec, o qual não usa os atributos dos nós do grafo. Só no caso da máquina de vetores de suporte, devido a que o ModelSet é um repositório com classes desbalanceadas e por sugestão do López et al. (2022, 2023), decidiu-se usar o algoritmo *SVC* com a estrategia de classificação multiclasse chamada Um Contra Um ou na língua inglesa *One versus One*.

A fim de analisar o impacto do número de metamodelos por classes do ModelSet no processo de *embedding*, foram estabelecidos segmentos no repositório. Como se pode observar na Tabela 5.2, a segmentação incluiu: classes com cem ou mais metamodelos, com 50 ou mais, etc. Com isso, pode se entender que, quanto menor o número de metamodelos por classes e maior o número de classes, maior foi o número de metamodelos processados.

Tabela 5.2: Metamodelos por Classe do Repositório ModelSet

Metamodelos por Classe	**Repositório com Duplicados**		**Repositório sem Duplicados**	
	Classes	Metamodelos	Classes	Metamodelos
De 100 a mais	13	2304	5	666
De 50 a mais	25	3193	14	1236
De 40 a mais	31	3446	18	1418
De 30 a mais	39	3704	22	1562
De 20 a mais	46	3869	33	1845
De 10 a mais	67	4155	48	2067

Fonte: Elaboração própria baseada no ModelSet (López et al., 2022).

Na Tabela 5.3, pode-se observar os resultados dos experimentos segunda as especificações previas. Os valores exibidos correspondem à métrica de acurácia balanceada expressada em escala percentual, considerando que os valores mais altos são melhores. Nos dois casos (com o repositório com duplicados e sem duplicados), se evidencia que, conforme o número de metamodelos por classes decresce e o número de classes aumenta, os valores da acurácia

[1]https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

balanceada diminuíram consideravelmente. De modo que, na provas com o repositório com duplicados, o desempenho do processo de *embedding* descendeu de 38.8212% (usando 13 classes e 2304 metamodelos) para 21.2761% (com 67 classes e 4155 metamodelos). Igualmente, no caso das provas com o repositório sem duplicados, o indicadores passaram de 20.7167% (usando 5 classes e 666 metamodelos) para 5.8912% (com 48 classes e 2067 metamodelos).

Tabela 5.3: Resultados dos Experimentos de Compreensão

Metamodelos por Classe	**Repositório com Duplicados**	**Repositório sem Duplicados**
De 100 a mais	38,8212	20,7167
De 50 a mais	27,4110	13,3430
De 40 a mais	24,0368	10,8759
De 30 a mais	23,6517	9,3487
De 20 a mais	22,3838	6,8818
De 10 a mais	21,2761	5,8912

Fonte: Elaboração própria.
Nota: Pontuações correspondentes à acurácia balanceada em escala percentual.

Adicionalmente, foram feitos alguns experimentos para avaliar os efeitos das técnicas de escalonamento (normalização e padronização de dados) das representações vetoriais no processo de *embedding*. Não obstante, esses resultados não apresentaram diferenças significativas em relação aos dos experimentos realizados sem o uso das técnicas referidas. Por outra parte, com o propósito de manter as propriedades originais dos metamodelos do tipo Ecore, principalmente em referencia à sensibilidade entre os caracteres, nos experimentos não foram convertidos os nomes ou valores das propriedades dos grafos em minúsculas.

5.4 EXPERIMENTOS DE *EMBEDDINGS*

Coerentemente ao primeiro objetivo específico da pesquisa, nesta secção são apresentados os experimentos que mostram as capacidades do método de *embedding* chamado Graph2Vec ao usar, além da estrutura, um atributo do grafo para conseguir as representações vetoriais. Depois, considerando o atributo que exibe maior acurácia balanceada, foram feitos experimentos adicionais para conseguir uma ótima representação vetorial através do ajuste dos parâmetros do Graph2Vec do Karate Club.

Todos os experimentos foram feitos usando o repositório ModelSet com suas exclusões especificadas anteriormente, para o caso do repositório com duplicados usou-se suas 67 classes e 4155 metamodelos e, igualmente, para o caso do repositório sem duplicados usou-se suas 48 classes e 2067 metamodelos. Assim, se implementou o algoritmo de classificação supervisionada SVC do Scikit-Learn -com seus hiperparâmetros predefinidos- para avaliar indiretamente o desempenho do processo de por meio da métrica chamada acurácia balanceada.

Considerando os elementos que podem ser parte dos metamodelos do tipo Ecore no contexto da MDSE. Alguns metamodelos do ModelSet apresentam nós especiais formados somente por um identificador e um atributo do tipo de nó chamado EClass com valor igual a EGenericType (elemento genérico parte do Ecore). Na Tabela 5.4, pode-se verificar o percentagem deste tipo de nós por cada metamodelo. No caso do repositório com duplicados há uma média de 42,9311% deste tipo de nós especiais. Similarmente, o repositório sem duplicados tem uma média de 42,0687% por cada metamodelo.

Tabela 5.4: Nós do tipo *EGenericType* por Metamodelo do Repositório ModelSet

	Repositório com Duplicados	**Repositório sem Duplicados**
Número de metamodelos	4155	2067
Média	**42,9311**	**42,0687**
Desvio Padrão	6,6906	5,9531
Mínimo	0,0000	0,0000
Primeiro Quartil (25%)	40,0000	39,1304
Mediana (50%)	43,4783	42,4710
Terceiro Quartil (75%)	46,6543	45,8096
Máximo	75,5187	75,5187

Fonte: Elaboração própria baseada no ModelSet (López et al., 2022).
Nota: Os valores dos parâmetros estatísticos foram calculados sob os percentagens dos nós do tipo *EGenericType* estimados em cada metamodelo.

Dos experimentos de López et al. (2022) se pode concluir que -para o processo de *embedding*- o atributo mais importante é o *name* ou nome dos nós. Não obstante, para processar todos os nós do grafo, os nós do tipo EGenericType tiveram que ser pré-processados para imputar o atributo *name*. Para tal fim, foram usadas as seguintes estrategias:

- Adicionar o atributo *name* com o valor ***None*** ou Nulo. Em Python, o *None* é um tipo especial que representa a ausência de um valor.
- Adicionar o atributo *name* com o valor ***Unknown***. Dada a versatilidade do elemento EGenericType. *Unknown* é uma palavra que representa um valor genérico.
- Criar um atributo chamado *mergedName* com o valor do atributo *EClass* fusionado com o valor do atributo *name* separados por um espaço. No caso dos nós com o elemento *EGenericType*, o valor do atributo *name* foi ***Unknown***.
- Excluir os nós com o elemento EGenericType.

A Tabela 5.5, mostra os resultados, sob a métrica chamada acurácia balanceada, da exploração do impacto de cada atributo usado como parâmetro no processo de *embeddings* implementado com o método Graph2Vec com seus hiperparâmetros predefinidos. Nos dois casos, repositório com duplicados e sem duplicados, as implementações do Graph2Vec sem o uso de um atributo dos nós apresentam pontuações de 21,2761% e 5,8912% respectivamente, os quais são considerados como os resultados mais baixos, demonstrando que a estrutura dos grafos que representam aos metamodelos do tipo Ecore são insuficientes para classificá-los. Similarmente, o uso do atributo EClass (tipo de nó) exibe valores de 22,1472% e 6,0065% para cada caso, o que pode ser devido à pouca variabilidade de dados (somente são cinco tipos de nó nos metamodelos: EPackage, EClass, EAttribute, EReference e EGenericType).

Os melhores valores de desempenho foram atingidas pelas estrategias de imputação do atributo *name* para os nós do tipo EGenericType. A imputação para *None* alcançou 48,5619% para o repositório com duplicados e 21,1764% para o repositório sem duplicados. Bastante próximos, a imputação para *Unknown* obteve um 48,0295% e 19,9152% para cada cenário respeito aos repositórios. A estrategia de fusão de atributos conseguiu estimações de 47,7949% e de 17,5168% respectivamente para repositório, considerando que foram feitas provas com uniões sem espaços em brancos entre os atributos, mas os resultados foram muito similares aos apresentados. A exclusão dos nós do tipo EGenericType estimou 21,8530% e 7,5327% para cada

repositório, nesta parte, o decremento do desempenho é devido a que também foram suprimidas as relações (arestas do grafo) deste tipo de nós.

Das analises anteriores, reconhece-se que as métricas das imputações do atributo *name* para *None* e para *Unknown*, além de ser muito próximas, são as pontuações mais relevantes para cada caso. Embora, é possível acreditar que a maior eficiência foi obtida pela imputação para *None*. Isto, devido a que, neste contexto do processo de *embedding*, o Python usa menos recursos ao trabalhar com o tipo especial *None* em comparação com uma cadena de texto do tipo *str* ou *string*.

Tabela 5.5: Atributos usados no Processo de *Embedding* com o Graph2Vec

Atributo usado no Graph2Vec	**Repositório com Duplicados**	**Repositório sem Duplicados**
Sem atributos, usando só a estrutura	21,2761	5,8912
EClass (tipo de nó)	22,1472	6,0065
name, imputados para ***None***	**48,5619**	**21,1764**
name, imputados para ***Unknown***	48,0295	19,9152
mergedName	47,7949	17,5168
name, excluindo os nós do tipo *EGenericType*	21,8530	7,5327

Fonte: Elaboração própria.
Nota: Pontuações correspondentes à acurácia balanceada em escala percentual.

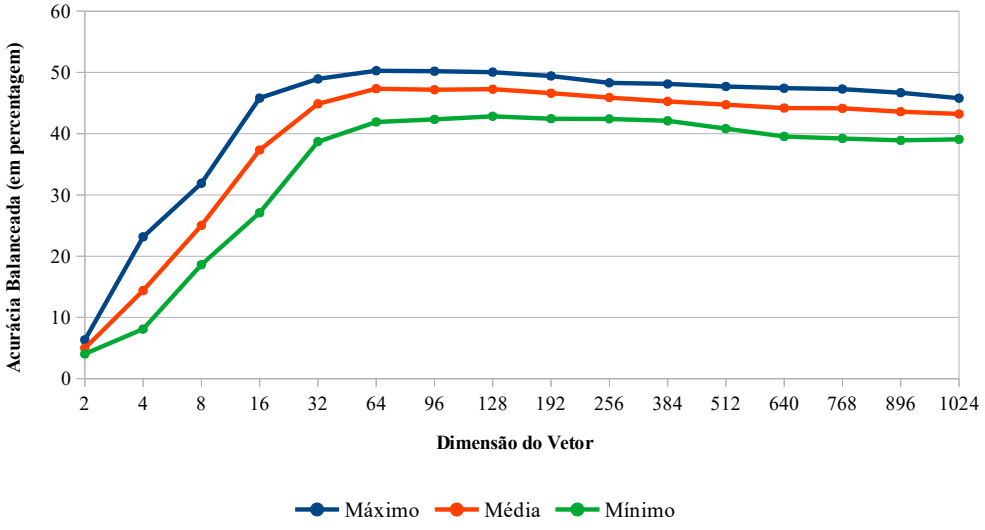
Focados principalmente no processo de *embedding* através do método chamado Graph2Vec do Karate Club. Nos experimentos seguintes foram executados a fim de explorar o impacto dos hiperparâmetros deste método para conseguir uma representação vetorial ótima. Desta forma, foi implementado o Graph2Vec usando o atributo *name* imputado para *None*. Igualmente, estabeleceu-se um espaço de busca formado por números que são potências de dois e alguns valores adicionais. Assim, foram testados dois hiperparâmetros do Graph2Vec, um com respeito à dimensão do vetor usando um intervalo de valores de dois até 1024 e outro hiperparâmetro acerca do número de iterações Weisfeiler-Lehman com valores de dois até 64. Mantendo os demais hiperparâmetros com seus valores predefinidos.

Na Figura 5.2, são mostrados os resultados para o repositório ModelSet com duplicados, medidos com a métrica acurácia balanceada. Nas duas figuras é possível notar que as linhas das pontuações máximas, médias e mínimas apresentam inclinações similares; por conseguinte, pode-se verificar que os dados seguem um padrão comum, conforme a uma distribuição homogênea e sem presencia de valores atípicos. Especificamente, na Figura 5.2(a) em relação ao hiperparâmetro dimensão do vetor, percebe-se uma inclinações ascendentes com dimensões de 2 até 32, com uma média de 4,9609% até 44,8756% e uma pontuação máxima de 48,9405%. Seguido dos valores maiores com dimensões no espaço de 64 até 192, médias entre 47,3419% e 46,6079% e um valor máximo de 50,2806%. Finalmente, as pontuações apresentam um leve decréscimo com dimensões de 256 a 1024, com médias que vão de 45,9022% a 43,2255% e um valor máximo de 48,3112%.

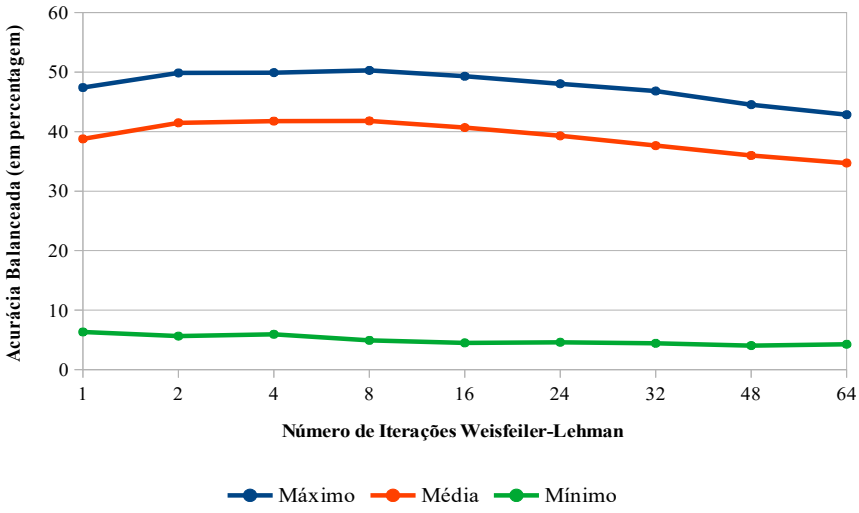
Neste mesmo sentido, a Figura 5.2(b) descreve o impacto do hiperparâmetro número de iterações Weisfeiler-Lehman no processo de *embedding*. Fixando este hiperparâmetro igual a um o desempenho é regular, com um valor médio de 38,7792% e um máximo de 47,4029%. Pode-se considerar que as pontuações maiores estão na faixa de 2 até 8, com uma média de 41,4792% até 41,7961% e um valor máximo de 50,2806%. Na última parte, os valores denotam uma diminuição no intervalo de 16 até 64, com médias que vão de 40,6804% a 34,7222% e

um valor máximo de 49,3003%. Considerando que a linha com as pontuações mínimas está direitamente relacionada com o vetor gerado com duas dimensões.

Portanto, para este caso perante ao ModelSet com duplicados, a fim de obter uma representação vetorial ótima, foi selecionada um espaço de busca formado por uma faixa de 32 até 256 para o hiperparâmetro relacionado com a dimensão do vetor e outra faixa de 2 até 8 para o hiperparâmetro número de iterações Weisfeiler-Lehman.



(a) Dimensão do Vetor



(b) Número de Iterações Weisfeiler-Lehman

Figura 5.2: Analises dos hiperparâmetros do Graph2Vec usando o ModelSet com Duplicados

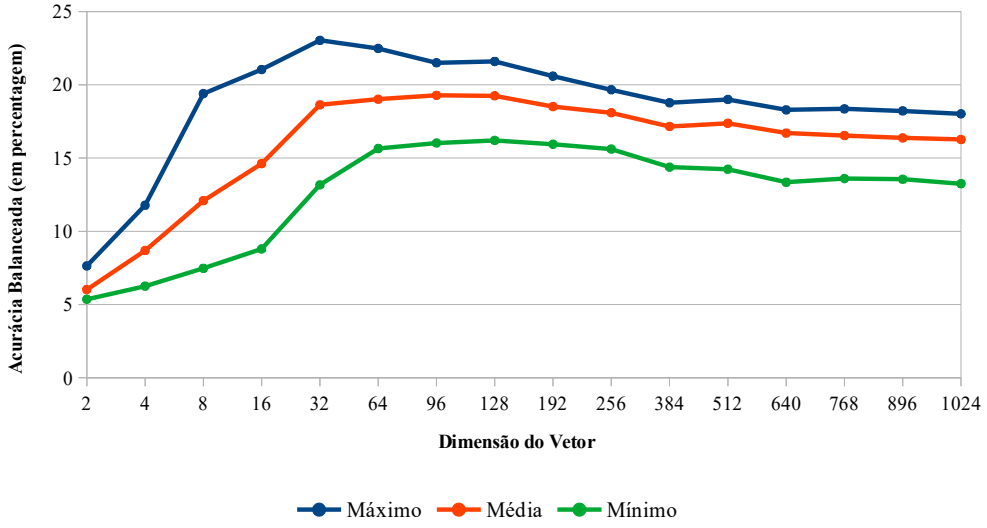
Fonte: Elaboração própria baseada nos dados da Tabela A.1

Do mesmo modo, foram feitos experimentos com o repositório ModelSet sem duplicados, a Figura 5.3 mostra os resultados medidos com a métrica acurácia balanceada. Em ambas as figuras, pode-se perceber que as linhas das pontuações máximas, médias e mínimas exibem inclinações bastantes similares, conforme a uma distribuição homogênea e sem presencia de valores atípicos. Respeito ao hiperparâmetro dimensão do vetor, na Figura 5.3(a), percebe-se inclinações ascendentes com dimensões de 2 até 16, com uma média de 6,0197% até 14,6219% e uma pontuação máxima de 21,0436%. Seguido dos valores maiores com dimensões no espaço de 32 até 128, médias entre 18,6334% e 19,2499% e um valor máximo de 23,0342%. Finalmente,

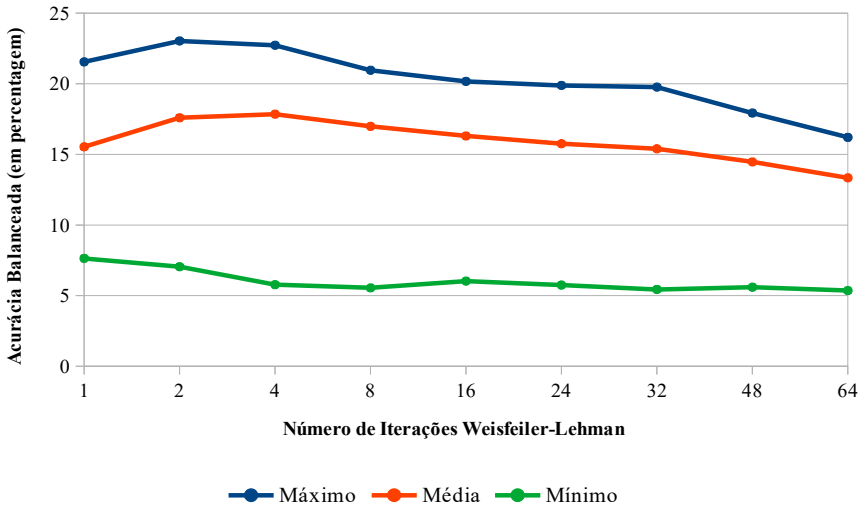
as pontuações apresentam um leve decréscimo com dimensões de 192 a 1024, com médias que vão de 18,5118% a 16,2697% e um valor máximo de 19,0023%.

Coerente com as análises, a Figura 5.3(b) descreve o impacto do hiperparâmetro número de iterações Weisfeiler-Lehman no processo de *embedding*. Fixando este hiperparâmetro igual a um o desempenho é regular, com um valor médio de 15,5406% e um máximo de 21,5516%. Pode-se considerar que as pontuações maiores estão na faixa de 2 até 4, com uma média de 17,5932% até 17,8505% e um valor máximo de 23,0342%. Na última parte, os valores denotam uma diminuição no intervalo de 8 até 64, com médias de 16,9862% a 13,3424% e um valor máximo de 20,9591%. Considerando que a linha com as pontuações mínimas esta direitamente relacionada com o vetor gerado com duas dimensões.

Neste contexto, sob o ModelSet sem duplicados, com o propósito de obter uma representação vetorial ótima, foi selecionado um espaço de busca formado por uma faixa de 32 até 256 para o hiperparâmetro relacionado com a dimensão do vetor e outra faixa de 2 até 4 para o hiperparâmetro número de iterações Weisfeiler-Lehman.



(a) Dimensão do Vetor



(b) Número de Iterações Weisfeiler-Lehman

Figura 5.3: Analises dos hiperparâmetros do Graph2Vec usando o ModelSet sem Duplicados

Fonte: Elaboração própria baseada nos dados da Tabela A.2

Mantendo o objetivo de conseguir uma representação vetorial ótima dos metamodelos em grafos do ModelSet, através da implementação do método de *embedding* chamado Graph2Vec parte do Karate Club, também foram feitos experimentos com o hiperparâmetro *epoch* considerando as faixas de valores referenciais para os hiperparâmetro especificados nos parágrafos anteriores. Não obstante, devido ao desbalanço dos metamodelos, ao incrementar o número de *epochs* a valores maiores de dez, os desempenhos mais relevantes foram achados para o hiperparâmetro número de iterações Weisfeiler-Lehman estabelecido a dois. Assim, para valores maiores a dois deste hiperparâmetro os desempenhos diminuíram significativamente.

Além disso, baseado nos experimentos usando o repositório com e sem duplicados, pode-se afirmar que implementar o Graph2Vec com seu hiperparâmetro número de iterações Weisfeiler-Lehman com valores elevados gera um modelo de aprendizado do *embedding* altamente complexo capaz de aprender sistematicamente particularidades irrelevantes dos metamodelos do ModelSet. Isto, principalmente, devido ao desbalanço do repositório, o modelo de aprendizado se mantém focado nas classes com maior frequência. O que produz um viés muito baixo, o que é compensado com uma variância muito alta, diminuindo a capacidade de representação do Graph2Vec.

Consequentemente, os seguintes experimentos foram executados para explorar o impacto do hiperparâmetro *epoch* no processo de *embedding*, usando o método Graph2Vec, medidos pela métrica acurácia balanceada. Assim, para todos os casos, estabeleceu-se um valor de dois para o hiperparâmetro número de iterações Weisfeiler-Lehman, igualmente, usou-se uma faixa de 32 até 256 para o hiperparâmetro dimensão do vetor e outra faixa de 10 até 5000 para o hiperparâmetro *epoch*.

Em particular, na Tabela 5.6 são apresentados os resultados para o caso do repositório ModelSet com duplicados. Em relação, ao hiperparâmetro *epoch*, percebe-se pontuações ascendentes na faixa de 10 até 100, com uma média entre 48,4119% e 80,4554% e um valor máximo de 81,2100%. Seguido das pontuações mais relevantes, que estão dados para o número de *epochs* igual a 500, com uma média de 81,6392% e um valor máximo de 82,2809%. Ao final, as pontuações exibem uma leve diminuição no intervalo de 1000 até 5000, com médias que passam de 81,2347% a 80,3195% e um valor máximo de 81,9975%. Da mesma forma, em relação ao hiperparâmetro dimensão do vetor, pode-se ver que as pontuações mostram pouca variabilidade, as médias estão entre 73,5855% e 76,0702%, considerando que os valores mais significativos são 82,2809% e 82,2584% para as dimensões de 96 e 160, respetivamente.

Tabela 5.6: Impacto da Dimensão do Vetor e do Número de Épocas do Graph2Vec na Representação Vetorial do ModelSet com Duplicados

Épocas	**Dimensão do Vetor**								**Máximo**	**Média**	**Mínimo**
	32	**64**	**96**	**128**	**160**	**192**	**224**	**256**			
10	48,9405	49,8510	49,1240	48,5619	48,0861	47,8966	47,4630	47,3719	49,8510	48,4119	47,3719
30	72,5969	73,9410	74,6315	74,5268	74,6438	74,2095	73,9554	74,1929	74,6438	74,0872	72,5969
50	75,9789	78,4860	78,8286	78,4914	77,8761	78,7182	78,6745	77,7990	78,8286	78,1066	75,9789
70	77,3323	79,7668	80,5666	80,1541	80,2669	80,3436	79,7857	79,7579	80,5666	79,7467	77,3323
100	77,4106	80,5570	81,0314	80,5939	81,2100	81,0596	80,9417	80,8392	81,2100	80,4554	77,4106
500	79,1694	82,1803	**82,2809**	81,7348	**82,2584**	82,0239	81,7996	81,6662	**82,2809**	**81,6392**	**79,1694**
1000	78,7541	81,4102	81,2782	81,4276	81,9975	81,5962	81,7493	81,6647	81,9975	81,2347	78,7541
5000	78,5010	80,8907	80,8206	80,5524	80,2188	80,4600	80,3942	80,7187	80,8907	80,3195	78,5010
Máximo	79,1694	82,1803	**82,2809**	81,7348	**82,2584**	82,0239	81,7996	81,6662			
Média	73,5855	75,8854	**76,0702**	75,7554	**75,8197**	75,7885	75,5954	75,5013			
Mínimo	48,9405	49,8510	**49,1240**	48,5619	**48,0861**	47,8966	47,4630	47,3719			

Fonte: Elaboração própria.

Nota: Pontuações correspondentes à acurácia balanceada em escala percentual. Sublinhado os resultados com hiperparâmetros predefinidos do Graph2Vec e, em negrito, os melhores resultados.

Para o caso do repositório ModelSet sem duplicados, como pode-se ver na Tabela 5.7, em relação, ao hiperparâmetro *epoch*, percebe-se pontuações ascendentes na faixa de 10 até 100, com uma média entre 20,4236% e 72,4070% e um valor máximo de 73,1207%. Seguido das pontuações mais relevantes, que estão dados para o número de *epochs* igual a 500, com uma média de 72,5716% e um valor máximo de 73,3260%. Ao final, as pontuações exibem uma leve diminuição no intervalo de 1000 até 5000, com médias que passam de 71,1838% a 69,1590% e um valor máximo de 71,7970%. Da mesma forma, em relação ao hiperparâmetro dimensão do vetor, pode-se ver que as pontuações mostram pouca variabilidade, as médias estão entre 63,3298% e 64,5701%, considerando que os valores mais significativos são 73,3260% e 73,0263% para as dimensões de 96 e 128, respetivamente.

Tabela 5.7: Impacto da Dimensão do Vetor e do Número de Épocas do Graph2Vec na Representação Vetorial do ModelSet sem Duplicados

Épocas	**Dimensão do Vetor**								**Máximo**	**Média**	**Mínimo**
	32	**64**	**96**	**128**	**160**	**192**	**224**	**256**			
10	23,0342	22,4828	21,3497	21,1764	19,1820	19,4907	18,4471	18,2256	23,0342	20,4236	18,2256
30	64,2156	64,6208	64,9677	64,3221	64,7304	64,8274	64,5748	64,4060	64,9677	64,5831	64,2156
50	69,8093	70,5416	70,4211	69,9703	69,9736	70,3415	70,2618	70,5342	70,5416	70,2317	69,8093
70	71,2245	71,4240	71,3094	71,3095	70,9771	71,4613	71,5934	72,5114	72,5114	71,4763	70,9771
100	71,2668	72,0873	73,1207	72,5848	72,0037	72,7198	72,7619	72,7111	73,1207	72,4070	71,2668
500	70,7170	72,3526	**73,3260**	**73,0263**	72,5676	73,1480	72,9830	72,4523	**73,3260**	**72,5716**	**70,7170**
1000	69,2484	71,3123	71,7970	71,3833	71,6009	71,7361	71,3683	71,0239	71,7970	71,1838	69,2484
5000	67,1227	68,7902	70,2695	69,8077	69,3549	69,4257	69,2448	69,2569	70,2695	69,1590	67,1227
Máximo	71,2668	72,3526	**73,3260**	**73,0263**	72,5676	73,1480	72,9830	72,7111			
Média	63,3298	64,2014	**64,5701**	**64,1976**	63,7988	64,1438	63,9044	63,8902			
Mínimo	23,0342	22,4828	**21,3497**	**21,1764**	19,1820	19,4907	18,4471	18,2256			

Fonte: Elaboração própria.
Nota: Pontuações correspondentes à acurácia balanceada em escala percentual. Sublinhado os resultados com hiperparâmetros predefinidos do Graph2Vec e, em negrito, os melhores resultados.

Atingindo o primeiro objetivo específico, que visa a implementação de um método de *embedding* de grafos sob uma abordagem de grafos inteiros para mapear os metamodelos de um repositório em representações vetoriais densas, de baixa dimensionalidade e de longitude fixa; baseado nos experimentos desenvolvidos, em um balanço entre a média e os valores máximos das acurácias balanceadas obtidas, além de considerar a dimensionalidade dos vetores de características sob o principio de parcimônia, pode-se afirmar que os hiperparâmetros ótimos do Graph2Vec para conseguir as representações do ModelSet são os mesmos para os dois casos (repositório com e sem duplicados). Assim, a configuração do Graph2Vec ajustou-se com o hiperparâmetro número de iterações Weisfeiler-Lehman estabelecido em dois, a dimensão do vetor em 96 e número de *epochs* em 500.

5.5 EXPERIMENTOS DE CLASSIFICAÇÃO

Respeito ao segundo objetivo da pesquisa, nesta secção são mostrados os resultados da classificação usando as representações vetoriais geradas pelo método de *embedding* chamado Graph2Vec do Karate Club explicadas na secção anterior. As classificações foram feitas por três algoritmos de aprendizado de máquina supervisionado e avaliadas pela validação cruzada do tipo *tenfolds* usando a métrica chamada acurácia balanceada.

Especificamente, na pesquisa foram usadas as implementações de três algoritmos de classificação: o primeiro, o XGBClassifier parte da biblioteca XGBoost[2], acrônimo de *eXtreme Gradient Boosting*, que implementa as árvores de decisão impulsionadas por gradiente. O segundo

[2]https://xgboost.readthedocs.io/en/stable/

algoritmo, o SVC[3] de Scikit-Learn, o qual usa a máquina de vetores de suporte. E, terceiro, o MLPClassifier[4], igualmente de Scikit-Learn, que implementa uma rede neural conhecida como perceptron multicamada.

O XGBoost, originalmente apresentado por Chen e Guestrin (2016) e suportado pela comunidade de desenvolvedores, é considerado o estado da arte dos métodos de *ensemble* baseados em *boosting* por gradiente de árvores de decisão (Kumar e Jain, 2020). Assim, o XGBoost é reconhecido pelo seu alto desempenho em termos de bons resultados e uso eficiente dos recursos computacionais ao permitir o processamento paralelo, distribuído e uso de GPUs (Kunapuli, 2023; Wade e Glynn, 2020). O objetivo do XGBoost é otimizar uma função de perda com estatísticas de gradiente e adicionar um termo de regularização à função objetivo para regular a complexidade do modelo e evitar o *overfitting* (Sarker, 2021).

No XGBoost, todos os hiperparâmetros são opcionais. Embora, no contexto do XGBClassifier, o único hiperparâmetro obrigatorio é o *objective*. O *objective* permite especificar a função objetivo, em relação a uma tarefa ou um objetivo do aprendizado. O valor predefinido é *binary:logistic*, que é usada para classificação binaria. Para o caso de classificação multiclasse ou multinomial, os valores mais ótimos são *multi:softmax* ou *multi:softprob*, além de especificar o número de classes através do *num_classes*. Nesta pesquisa foi usado o *multi:softmax* como função objetivo e o número de classes estabeleceu-se segunda a Tabela 5.1. Para o contexto da pesquisa, segundo Owen (2022), Grigoriev (2021) e Wade e Glynn (2020) os principais hiperparâmetros deste classificador são:

- *booster*: Tipo de modelo base, usado para estabelecer a estrutura do modelo final, pode ser:
  - *gbtree*: Usa árvores de decisão tradicionais. É o valor predefinido.
  - *gblinear*: Usa funções lineares para construir um modelo final mais simples e rápido para treinar. Ainda é um *booster* não determinista.
  - *dart*: Semelhante ao *gbtree*, mas incorpora a técnica chamada *drop-out* como mecanismo de regularização.
- *n_estimators*: Número de iterações do *boosting* (árvores de decisão). Quanto maior esse valor, mais complexo é o modelo e mais demorado é o treinamento. É importante notar que um grande número de árvores pode levar ao *overfitting* do modelo. O valor predefinido é cem.
- *max_depth*: Profundidade máxima de cada árvore. O valor predefinido é igual a 6.
- *learning_rate* ou *eta*: Taxa de aprendizado. Esta é a taxa de aprendizado do algoritmo de otimização de descida de gradiente. Deve-se considerar que quanto menor é a taxa de aprendizado, mais probabilidades de atingir previsões ótimas. O valor predefinido é 0,3.
- *subsample*: Proporção de subamostragem das instâncias que são usadas no treinamento de cada árvore. O valor predefinido é um.

De outra parte, o SVC siglas de *Support Vector Classification* é a implementação da biblioteca LibSVM (Géron, 2023). A máquina de vetores de suporte (SVM) faz parte do grupo dos algoritmos baseados em distância Owen (2022). O uso do *kernel* na SVM permite elevar as

[3]https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

[4]https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

dimensões do conjunto de dados não separáveis linearmente a um espaço superior, de maneira que exista uma margem divisória clara entre as classes de dados, sob uma perspectiva linear (Owen, 2022; Jiang, 2021). Assim, para o contexto da pesquisa, segundo Géron (2023) e Owen (2022) os principais hiperparâmetros para o classificador SVC são:

- *C*: Parâmetro de regularização. Quanto menor o valor, mais forte o impacto da regularização no modelo e, portanto, maior a possibilidade de evitar o *overfitting*. O valor predefinido é um.
- *kernel*: Tipo do *kernel*, pode ser:
  - *linear*: Supõe que os dados são linearmente separáveis e não faz transformações nas dimensões.
  - *poly*: Da língua inglesa *polynomial*. Adiciona características polinomiais de diferentes graus. O grau é especificado pelo hiperparámetro *degree*, o qual apresenta um valor predefinido de 3.
  - *rbf*: Da língua inglesa *radial basis function*, também conhecido como *kernel* gaussiano. Usa o método de características de similaridade para medir a distancia entre dois pontos de dados em dimensões infinitas. É usado quando a natureza da relação entre as dimensões é desconhecida. É o valor predefinido.
  - *sigmoid*: Transforma o conjunto de dados usando uma função tangente hiperbólica. Os limites de decisão tendem a ser curvos e irregulares, ajustando-se a uma forma sigmoide.
    Adicionalmente, pode se estabelecer um *kernel* personalizado.
- *gamma*: Controla a influência de cada amostra individual no limite de decisão. É usado pelos *kernels*: *poly*, *rbf* e *sigmoid*. Pode-se designar valores como: *auto*, *scale* ou um número em ponto flutuante, mas o valor predefinido é *scale*.
- *class_weight*: Atribui pesos para cada classe durante o treinamento, é usado como estrategia ao trabalhar com conjuntos de dados não balanceados, pode ser:
  - *balanced*: Ajusta automaticamente os pesos inversamente proporcionais às frequências de classe nos dados de entrada.
  - *None*: indica que todas as classes têm o mesmo peso igual ao um. É o valor predefinido.
    Também pode se usar um dicionario a fim de atribuir os pesos manualmente para cada classe.
- *decision_function_shape*: Especifica a forma da função de decisão usada para classificação multiclasse. Pode optar por valores como *ovr* (*one-versus-the-rest*) para comparar uma classe contra as outras ou *ovo* (*one-versus-one*) que compara um par de classes. Ao igual que os experimentos anteriores. O valor predefinido *ovr* foi mudado para *ovo* por recomendação de López et al. (2022).

Em relação ao MLPClassifier, que é uma implementação que pode usar métodos de otimização como LBFGS (do inglês *Limited-memory Broyden–Fletcher–Goldfarb–Shanno*), SGD (do inglês *Stochastic Gradient Descent*) ou ADAM (do inglês *Adaptive Moment Estimation*). Esses métodos ajustam os pesos e desvios da rede neural durante o processo de treinamento para minimizar a função de perda logarítmica. Para o contexto da pesquisa, segundo Géron (2023) e Owen (2022) os seguintes são os principais hiperparâmetros para este classificador:

- *hidden_layer_sizes*: Representa o número de neurônios para cada camada oculta do MLP. O valor predefinido é (100,).
- *activation*: Função de ativação não lineares para as camadas ocultas, pode ser:
  - *identity*: Função identidade. Retorna o mesmo valor de entrada.
  - *logistic*: Função sigmoide logística. Produz uma saída entre 0 e 1, similar a uma probabilidade.
  - *tanh*: Função tangente hiperbólica. Tem uma saída entre -1 e 1, mais ou menos normalizada, ou seja, centrada em torno de zero.
  - *relu*: Acrônimo de *Rectified Linear Unit* ou Unidades Lineares Retificadas. Retorna o mesmo valor de entrada se for positivo e zero em caso contrário. É o valor predefinido.
- *solver*: Otimizador de pesos, pode ser:
  - *lbfgs*: Pertence aos métodos quase-Newton.
  - *sgd*: Acrônimo de *Stochastic Gradient Descent* ou Descida de Gradiente Estocástico.
  - *adam*: Acrônimo de *Adaptive Moment Estimation* ou Estimativa de Momento Adaptativo. É um otimizador baseado no gradiente estocástico. É o valor predefinido.
- *learning_rate_init*: Taxa de aprendizado inicial, somente é usado com os *solvers* do tipo *sgd* ou *adam*. O valor predefinido é 0,001.
- *max_iter*: Número máximo de iterações. O valor predefinido é 200.
- *early_stopping*: O valor predefinido é *False*.

Os primeiros experimentos desta secção foram executados usando os hiperparâmetros predefinidos para cada classificador. A Tabela 5.8 mostra os resultados medidos pela acurácia balanceada e o desvio padrão. Nos dois casos, as melhores pontuações foram obtidas pelo algoritmo MLPClassifier, seguidos dos SVC e XGBClassifier. Exatamente, o repositório com duplicados alcançou o resultado máximo de 85,5554% e, por outra parte, o repositório sem duplicados atingiu um máximo de 75,9017%.

Tabela 5.8: Resultados da Classificação usando os Valores Predefinidos dos Hiperparâmetros para cada Algoritmo

	Repositório com Duplicados	**Repositório sem Duplicados**
XGBClassifier	78,0905±2,7189	60,2651±4,5377
SVC	82,2809±3,1024	73,3260±3,0445
MLPClassifier	85,5554±2,8289	73,9017±4,9044

Fonte: Elaboração própria.
Nota: Pontuações correspondentes à acurácia balanceada em escala percentual.

5.5.1 Classificação usando o algoritmo XGBClassifier

A fim de procurar resultados mais ótimos ao classificar os *embeddings* dos metamodelos do repositório ModelSet, foram feitos experimentos mais detalhados com cada algoritmo. No caso do XGBClassifier, previamente provou-se o hiperparâmetro *booster* usando suas três opções: *gbtree*, *gblinear* e *dart*. Assim, o repositório com duplicados manteve sua acurácia balanceada inicial igual a 78,0905% para os *boosters gbtree* e *dart*; não obstante, usando a alternativa *gblinear* o desempenho baixou significativamente ao 9.7449%. De outra parte, o repositório sem duplicados atingiu um desempenho superior igual a 70,8363% com a opção *gblinear* e os outros *boosters* não superaram a pontuação inicial de 60,2651%. Por conseguinte, os seguintes experimentos usaram o hiperparâmetro *booster* igual a *gbtree* para o repositório com duplicados e *gblinear* para o repositório sem duplicados.

Igualmente foram testados os impactos dos hiperparâmetros *n_estimator* e *max_depth* na classificação, usando faixas de 50 até mil (com intervalos de 50 e cem) e de um até dez, respectivamente. Embora de serem os hiperparâmetros mais conhecidos para classificar dados desbalanceados, as variações destes hiperparâmetros não produziram melhoras significativas na acurácia balanceada em relação aos resultados do parágrafo anterior, tanto para o repositório com duplicados quanto para o repositório sem duplicados. Isso demonstra a flexibilidade do XGBoost, que permite se adaptar quase automaticamente ao conjunto de dados. Além disso, o XGBClassifier pode substituir alguns hiperparâmetros com valores estabelecidos pelo usuário por valores que o algoritmo considera mais ótimos (incluindo seus valores predefinidos).

Para analisar os impactos dos outros hiperparâmetro do XGBClassifier e procurar melhores resultados na classificação, estabeleceram-se espaços de busca para cada hiperparâmetro, como são apresentados na Tabela 5.9.

Tabela 5.9: Espaços de Busca para os Hiperparâmetros do XGBClassifier

Hiperparâmetros	**Espaços de Busca**
learning_rate	0,001, 0,01, 0,02, 0,03, 0,04, 0,05, 0,1, 0,15, 0,2, 0,25, 0,3, .., 0,5, 1
subsample	0,1, 0,2, .., 0,9, 1

Fonte: Elaboração própria.
Nota: Sublinhado o valor predefinido de cada hiperparâmetro.

Dessa forma, foi testado o hiperparâmetro *learning_rate* usando os espaços de busca estabelecidos na Tabela 5.9 e mantendo os outros hiperparâmetros da tabela com seus valores predefinidos. Como pode se ver na Tabela 5.10, nos dois casos, as pontuações são significativamente ascendentes até o ponto mais alto e depois diminuem um pouco, observando pontuações ínfimas aos extremos com *learning_rate* iguais a 0,001 e um. O repositório com duplicados alcançou a acurácia balanceada máxima de 79,2349% com *learning_rate* igual a 0,25. De outra parte, o repositório sem duplicados atingiu uma pontuação máxima de 76,2195% conforme a *learning_rate* igual a 0,02. Em ambos os casos, os valores do *learning_rate* de maior impacto são considerados baixos, isso evidencia que o XGBClassifier controlou o *overfitting* (Owen, 2022).

Igualmente, avaliou-se o impacto do hiperparâmetro *subsample* na classificação dos *embeddings* do ModelSet, variando o *subsample* segunda a Tabela 5.9 e mantendo os outros hiperparâmetros da tabela com seus valores predefinidos. Como pode se ver na Tabela 5.11, o repositório com duplicados alcançou o resultado máximo de 81,3224% com *subsample* igual a 0,4, isto significa, que só foram usadas 40% dos dados no treinamento a fim de evitar o *overfitting*. Para o repositório sem duplicados, a acurácia balanceada mais alta foi 70,8363% conforme a

Tabela 5.10: Impacto do Hiperparâmetro *learning_rate* do XGBClassifier na classificação

Hiperparâmetro *learning_rate*	**Repositório com Duplicados**	**Repositório sem Duplicados**
0,001	47,9369	16,2635
0,01	68,8935	74,9088
0,02	72,6665	**76,2195**
0,03	74,4749	75,6519
0,04	75,4640	75,4268
0,05	76,5708	75,3504
0,1	78,5377	74,6193
0,15	78,8005	73,1470
0,2	78,6219	72,3693
0,25	**79,2349**	71,9199
0,3	78,0905	70,8363
0,35	77,8759	70,6176
0,4	76,7645	70,3381
0,45	75,8884	69,7822
0,5	74,3838	69,7829
1	65,8482	2,0833

Fonte: Elaboração própria.
Nota: Pontuações correspondentes à acurácia balanceada em escala percentual. Sublinhado os resultados com hiperparâmetros predefinidos na Tabela 5.9 e, em negrito, os melhores resultados.

subsample igual a um, que corresponde ao mesmo resultado alcançado usando os hiperparâmetros com valores predefinidos. Neste último caso, as variações do hiperparâmetro não produziram melhoras significativas na classificação, além de reconhecer que o XGBClassifier limitou o uso do hiperparâmetro *subsample* durante o treino, mas conseguiu controlar o *overfitting*.

Tabela 5.11: Impacto do Hiperparâmetro *subsample* do XGBClassifier na classificação

Hiperparâmetro *subsample*	**Repositório com Duplicados**	**Repositório sem Duplicados**
0,1	72,5272	70,1190
0,2	79,4285	70,0561
0,3	80,6324	70,2442
0,4	**81,3224**	70,0277
0,5	80,0375	70,4765
0,6	80,1213	70,3240
0,7	79,3076	70,2460
0,8	79,1485	70,0781
0,9	78,7836	70,3266
1	78,0905	**70,8363**

Fonte: Elaboração própria.
Nota: Pontuações correspondentes à acurácia balanceada em escala percentual. Sublinhado os resultados com hiperparâmetros predefinidos na Tabela 5.9 e, em negrito, os melhores resultados.

A fim de melhorar o desempenho na classificação dos *embeddings* do ModelSet, os seguintes experimentos foram feitos usando o GridSearchCV[5] do Scikit-Learn, o qual é uma técnica de otimização de hiperparâmetros, que usa a validação cruzada e mitiga o risco de *overfitting* (Géron, 2023; Owen, 2022). Na Tabela 5.12 são apresentados os resultados do GridSearchCV com os espaços de busca especificados na Tabela 5.9. O repositório com duplicados atingiu a acurácia balanceada máxima de 81,3224% sob uma abordagem do *booster* tipo *xgbtree*, com o hiperparâmetro *learning_rate* igual a 0,3 e *subsample* de 0,4. Do mesmo modo, o repositório sem duplicados alcançou um desempenho ótimo de 76,2195% conforme ao *booster* igual a *xgblinear*, com o hiperparâmetro *learning_rate* igual a 0,02 e *subsample* de um.

Tabela 5.12: Resultados da Classificação usando o XGBClassifier com Hiperparâmetros Otimizados

Hiperparâmetros Otimizados	**Repositório com Duplicados**	**Repositório sem Duplicados**
booster	*xgbtree*	*xgblinear*
learning_rate	0,3	0,02
subsample	0,4	1
Acurácia balanceada	**81,3224±3,0457**	**76,2195±4,6940**

Fonte: Elaboração própria.
Nota: A acurácia balanceada é apresentada em escala percentual.

5.5.2 Classificação usando o algoritmo SVC

No SVC, foi testado primeiro o hiperparâmetro *class_weight* com valores *None* e *balanced*. Devido ao desbalanço do ModelSet, na Tabela 5.13, pode se evidenciar que a opção *balanced* obteve melhores resultados nos *kernels*: *poly*, *rbf* e *sigmoid*. O repositório com duplicados mostra um incremento máximo de 11,8846% e o repositório sem duplicados registrou um aumento de até 7,4431%. Não obstante, o *kernel linear* apresenta resultados similares. Portanto, os seguintes experimentos usaram o hiperparâmetro *class_weight* igual a *balanced*.

Tabela 5.13: Hiperparâmetro *class_weight* no SVC

Kernel	***None***	***balanced***	**Incremento**
Repositório com Duplicados			
linear	87,9481	88,0717	0,1236
poly	81,5234	85,3792	3,8557
rbf	82,2809	87,8503	5,5694
sigmoid	74,0567	85,9413	11,8846
Repositório sem Duplicados			
linear	77,2415	77,3513	0,1097
poly	66,6863	74,1294	7,4431
rbf	73,3260	78,9595	5,6335
sigmoid	72,2154	78,0515	5,8361

Fonte: Elaboração própria.
Nota: Pontuações correspondentes à acurácia balanceada em escala percentual. O incremento é a diferença entre as colunas *balanced* menos *None*. Sublinhado os resultados com hiperparâmetros predefinidos.

[5]https://scikit-learn.org/dev/modules/generated/sklearn.model_selection.GridSearchCV.html

Além disso, foram feitos experimentos usando os espaços de busca para cada hiperparâmetro do SVC, os quais são especificados na Tabela 5.14.

Tabela 5.14: Espaços de Busca para os Hiperparâmetros do SVC

Hiperparâmetros	Espaços de Busca
C	0,0001, 0,001, 0,01, 0,1, 1, 10, 100
	1000, 10000
kernel	*linear*, *poly*, *rbf*, *sigmoid*
degree	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
gamma	0,0104, 0,0212, 0,03, 0,04, 0,05, 0,06
	0,07, 0,08, 0,09, 0,1, 0,5, 1

Fonte: Elaboração própria.
Nota: Sublinhado o valor predefinido de cada hiperparâmetro. Os valores para *auto* o *scale* do hiperparâmetro *gamma* estão perto de 0,0104 e 0,0212 respectivamente.

Neste contexto, foram analisados os impactos do hiperparâmetro *C* em cada *kernel* do SVC, variando o *C* segunda a Tabela 5.14 e mantendo os outros hiperparâmetros da tabela com seus valores predefinidos. Como pode se ver na Tabela 5.15, em todos os casos, as pontuações são significativamente ascendentes até o ponto mais alto e depois diminuem um pouco. A excepção do *kernel linear*, que mostra valores ínfimos para *C* igual a 0,0001 e/ou 0,001, os outros *kernels* apresentam resultados muito baixos com valores de *C* entre 0,001 e 0,01. Não obstante, os *kernels* que atingiram valores mais altos foram o *kernel linear* e o *rbf*. No caso do repositório com duplicados o *kernel linear* alcançou a pontuação máxima de 88,5195% com *C* igual a 0,1, seguido do *kernel rbf* com um resultado de 87,8503% conforme o *C* igual a um. Por sua parte, o repositório sem duplicados o *kernel rbf* atingiu um valor máximo de 79,0576% com *C* igual a dez, seguido do *kernel linear* com um resultado de 78,8924% conforme o *C* igual a 0,01. Assim, com os valores de maior impacto de *C*, considerados como valores intermediários (nem altos nem baixos), é possível evidenciar que a classificação é feita com um equilíbrio razoável ao maximizar a margem e minimizar os erros (Géron, 2023; Lee, 2019).

Igualmente, com o objetivo de analisar os impactos do hiperparâmetro *degree* -usado exclusivamente pelo *kernel poly* do SVC- na classificação, foram feitos experimentos variando o grau do polinômio segundo o espaço de busca estabelecido na Tabela 5.14 e os outros hiperparâmetros da tabela mantiveram seus valores predefinidos. A Tabela 5.16 mostra que, para ambos os casos, as pontuações maiores são para o *degree* igual a um, depois as pontuações baixam significativamente. Para o caso do repositório com duplicados a acurácia balanceada mais alta é 88,2889% e para o repositório sem duplicados o melhor valor é 79,3726%. Nesta perspectiva, com um *degree* igual a um, pode se evidenciar que a classificação tende a ser linear.

Outro hiperparâmetro do SVC analisado foi o *gamma* -usado exclusivamente pelo *kernels: poly*, *rbf* e *sigmoid* do SVC-. Os experimentos foram feitos variando o *gamma* segunda a Tabela 5.14 e os outros hiperparâmetros da tabela mantiveram seus valores predefinidos. A Tabela 5.17 mostra que em todos os casos, antes (se houvesse) e depois das pontuações máximas, os valores diminuem um pouco, atendendo que o *kernel sigmoid* baixa até valores ínfimos com *gamma* de 0.5 e um. Para o repositório com duplicados, o *kernel rbf* alcançou a pontuação máxima de 88,4175% com *gamma* igual a 0,0104, seguido do *kernel poly* com um resultado de 87,1124% conforme o *gamma* igual a 0,05. De outra parte, para o repositório sem duplicados, o *kernel rbf* atingiu um valor máximo de 79,6906%, seguido do *kernel sigmoid* com um resultado de 78,6142%, ambos dois valores superiores obtidos com *gamma* igual a 0,0104. Neste abordagem, considerando que valores de *gamma* baixos impactaram mais significativamente no desempenho,

Tabela 5.15: Impacto do Hiperparâmetro *C* em cada *kernel* do SVC

Hiperparâmetro	***Kernels***			
C	***linear***	***poly***	***rbf***	***sigmoid***
Repositório com Duplicados				
0,0001	2,6617	1,9403	2,2388	2,6617
0,001	63,1452	1,4925	1,6418	1,6045
0,01	88,0738	1,4925	1,4925	1,4925
0,1	**88,5195**	65,5139	74,9791	70,7690
1	88,0717	85,3792	**87,8503**	85,9413
10	87,7631	**87,1440**	87,4866	**86,0719**
100	87,8180	86,6113	87,4581	85,5039
1000	87,8180	86,5212	87,4084	84,3282
10000	87,8180	86,5212	87,4084	83,9963
Repositório sem Duplicados				
0,0001	4,0625	3,7153	4,2361	4,0972
0,001	7,0347	3,1250	3,9965	3,9583
0,01	**78,8924**	3,2986	3,5069	3,4722
0,1	78,3849	29,6112	37,4384	46,6800
1	77,3513	**74,1294**	78,9595	**78,0515**
10	77,1167	74,0624	**79,0576**	75,7296
100	77,1167	73,1738	78,5858	73,9505
1000	77,1167	73,0349	78,5560	73,3773
10000	77,1167	72,9667	78,5560	72,9242

Fonte: Elaboração própria.
Nota: Pontuações correspondentes à acurácia balanceada em escala percentual. Sublinhado os resultados com hiperparâmetros predefinidos na Tabela 5.14 e, em negrito, os melhores resultados.

Tabela 5.16: Impacto do Hiperparâmetro *degree* do SVC na classificação

Hiperparâmetro ***degree***	**Repositório com Duplicados**	**Repositório sem Duplicados**
1	**88,2889**	**79,3726**
2	87,7638	79,0784
3	85,3792	74,1294
4	81,0006	61,9187
5	72,4709	44,8420
6	65,9877	31,6674
7	58,8605	21,0319
8	54,1826	16,3758
9	50,8083	14,1323
10	48,6257	12,2730

Fonte: Elaboração própria.
Nota: Pontuações correspondentes à acurácia balanceada em escala percentual. Sublinhado os resultados com hiperparâmetros predefinidos na Tabela 5.14 e, em negrito, os melhores resultados.

pode se evidenciar que todos os *embeddings* influenciaram na geração de uma fronteira de decisão mais suave e que não produzem *overfitting* (Géron, 2023; Lee, 2019).

Tabela 5.17: Impacto do Hiperparâmetro *gamma* em cada *kernel* do SVC

Hiperparâmetro	***Kernels***		
gamma	***poly***	***rbf***	***sigmoid***
Repositório com Duplicados			
0,0104 (*auto*)	77,9753	**88,4175**	**86,9789**
0,0212 (*scale*)	85,3792	87,8503	85,9277
0,03	86,8078	84,6479	80,7543
0,04	87,0252	82,3426	74,8874
0,05	**87,1124**	79,7368	69,6916
0,06	86,8579	77,5890	63,8202
0,07	86,6787	75,5159	57,5258
0,08	86,6675	74,4403	51,1338
0,09	86,6113	72,9664	44,7326
0,1	86,6222	70,4077	39,1743
0,5	86,5212	47,8610	1,6247
1	86,5212	41,6604	1,4925
Repositório sem Duplicados			
0,0104 (*auto*)	43,8930	**79,6906**	**78,6142**
0,0212 (*scale*)	74,0997	78,9595	78,0515
0,03	**75,2207**	76,0893	75,9782
0,04	74,1200	71,8525	72,5832
0,05	74,0363	66,6429	70,4975
0,06	74,1012	60,7785	68,0422
0,07	74,0362	54,2630	65,8630
0,08	73,7196	49,6335	63,5360
0,09	73,5091	45,4431	60,6399
0,1	73,1999	42,2850	57,4132
0,5	72,9667	9,4875	2,6389
1	72,9667	6,3214	2,2083

Fonte: Elaboração própria.
Nota: Pontuações correspondentes à acurácia balanceada em escala percentual. Sublinhado os resultados com hiperparâmetros predefinidos na Tabela 5.14 e, em negrito, os melhores resultados.

Para atingir um desempenho ótimo na classificação dos *embeddings* do ModelSet usando o SVC, foi usado o GridSearchCV com os espaços de busca especificados na Tabela 5.14. Como pode se ver na Tabela 5.18, o repositório com duplicados atingiu a acurácia balanceada máxima de 88,7378% sob uma perspectiva de classificação linear através do *kernel poly* de primeiro grau, com o hiperparâmetro *C* igual a um e *gamma* de 0,04. Do mesmo modo, o repositório sem duplicados alcançou um desempenho ótimo de 79,6906% conforme ao método de características de similaridade achadas conforme o *kernel rbf* com o hiperparâmetro *C* igual a um e *gamma* de 0,0104.

Tabela 5.18: Resultados da Classificação usando o SVC com Hiperparâmetros Otimizados

Hiperparâmetros Otimizados	Repositório com Duplicados	Repositório sem Duplicados
C	1	1
kernel	*poly*	*rbf*
degree	1	Não aplica
gamma	0,04	0,0104
Acurácia balanceada	**88,7378±2,8957**	**79,6906±4,0130**

Fonte: Elaboração própria.
Nota: A acurácia balanceada é apresentada em escala percentual.

5.5.3 Classificação usando o algoritmo MLPClassifier

Por último, foi testado o algoritmo MLPClassifier a fim de procurar outra perspectiva ao classificar os *embeddings* dos metamodelos do repositório ModelSet. Deve-se reconhecer que este algoritmo, usa de forma predefinida o hiperparâmetro *shuffle* estabelecido como *True*, o que facilitou ao MLPClassifier lidar com o desbalanço do ModelSet ao reorganizar aleatoriamente os *embeddings* para cada iteração dos experimentos.

Neste contexto, provou-se o hiperparâmetro *max_iter* usando duas faixas: a primeira de 10 até 200 com intervalos de dez e a segunda de 300 até cinco mil com intervalos de cem e mantendo os outros hiperparâmetros do MLPClassifier com seus valores predefinidos. Embora este hiperparâmetro representa o número de iterações, além de ter um comportamento similar ao *epochs*, principalmente com os otimizadores do tipo *sgd* e *adam*, as variações do *max_iter* não produziram melhoras significativas na acurácia balanceada em relação aos resultados da Tabela 5.8, tanto para o repositório com duplicados quanto para o repositório sem duplicados. Por conseguinte, os seguintes experimentos não usaram o hiperparâmetro *max_iter*.

Para analisar os impactos dos outros hiperparâmetro do MLPClassifier e procurar melhores resultados na classificação, estabeleceram-se espaços de busca para cada hiperparâmetro, como são detalhados na Tabela 5.19.

Tabela 5.19: Espaços de Busca para os Hiperparâmetros do MLPClassifier

Hiperparâmetros	Espaços de Busca
hidden_layer_sizes	100, 200, .., 1400 , 1500
activation	*identity*, *logistic*, *tanh*, *relu*
solver	*lbfgs*, *sgd*, *adam*
learning_rate_init	0,0001, 0,0002, ..0,0005, 0,001, 0,002 .., 0,005, 0,01, 0,02, ..0,05, 0,1

Fonte: Elaboração própria.
Nota: Sublinhado o valor predefinido de cada hiperparâmetro.

Assim, foi testado o hiperparâmetro *hidden_layer_sizes* usando os espaços de busca propostos na Tabela 5.19 e mantendo os outros hiperparâmetros da tabela com seus valores predefinidos. Como pode se ver na Tabela 5.20, o repositório com duplicados alcançou o resultado máximo de 87,1451% com *hidden_layer_sizes* igual a 1200. Por sua parte, o repositório sem duplicados atingiu uma acurácia balanceada máxima de 77,7805% conforme o *hidden_layer_sizes* igual a 1000. Além disso, foram feitos experimentos com dois, três e quatro camadas ocultas usando combinações com os valores de *hidden_layer_sizes* mais relevantes, tais como: (1200, 900), (1200, 900, 1200), (900, 1200, 1200, 900), etc. para o caso do repositório com duplicados

e, de (1000, 500), (1000, 500, 1000), (500, 1000, 1000, 500) e outras variações para o caso do repositório sem duplicados. Não obstante, as combinações testadas não conseguiram melhorias significativas nas acurácias balanceadas com respeito às pontuações obtidas com configurações de uma única camada oculta. Por tanto, se evidencia que uma arquitetura de uma camada oculta é eficiente para classificar os *embeddings* e prevenir o *overfitting*.

Tabela 5.20: Impacto do Hiperparâmetro *hidden_layer_sizes* do MLPClassifier na classificação

Hiperparâmetro ***hidden_layer_sizes***	**Repositório com Duplicados**	**Repositório sem Duplicados**
(100,)	85,5554	73,9017
(200,)	86,6896	76,2960
(300,)	86,3607	77,0063
(400,)	86,5809	75,6314
(500,)	86,7614	77,1051
(600,)	86,6553	76,8034
(700,)	86,7223	76,3563
(800,)	86,8486	76,4647
(900,)	86,9409	76,6421
(1000,)	86,6305	**77,7805**
(1100,)	86,8364	76,5997
(1200,)	**87,1451**	76,7650
(1300,)	87,0738	77,1783
(1400,)	87,0929	77,3395
(1500,)	86,6383	76,9442

Fonte: Elaboração própria.
Nota: Pontuações correspondentes à acurácia balanceada em escala percentual. Sublinhado os resultados com hiperparâmetros predefinidos na Tabela 5.19 e, em negrito, os melhores resultados.

Da mesma forma, testou-se o impacto do hiperparâmetro *activation* na classificação dos *embeddings* do ModelSet, usando os espaços de busca propostos segunda a Tabela 5.19 e mantendo os outros hiperparâmetros da tabela com seus valores predefinidos. A Tabela 5.21 mostra que, em todos os casos, os impactos mais significativos nas classificações foi usando *activation* com seu valor predefinido (*logistic*). Assim, o caso do repositório com duplicados a acurácia balanceada mais alta é 86,1514%, seguido do 85,5554% logrado conforme à função de ativação *relu*. E, de outro modo, para o repositório sem duplicados o maior resultado é 76,4943%, seguido do 75,3211% atingido conforme à função não linear *tanh*. Atendendo que a função de ativação do tipo sigmoide logística obteve melhores resultados em ambos os casos, pode se evidenciar a capacidade do MLPClassifier ao trabalhar classificações multiclasses usando distribuições binomiais.

Outro hiperparâmetro do MLPClassifier analisado foi o *solver*. Neste sentido, foram feitos experimentos variando o *solver* com suas opções segunda a Tabela 5.19 e mantendo os outros hiperparâmetros com seus valores predefinidos. Na Tabela 5.22 pode se ver que, no dois casos, o impacto mais significativo nas classificações foi usando *solver* com seu valor predefinido (*adam*), seguido do otimizador *lbfgs*. Assim, no caso do repositório com duplicados a acurácia balanceada alcançada mais alta é de 86,1514%, seguida de 84,3727%. Igualmente, neste ordem, analisando o impacto do *solver*, o repositório sem duplicados conseguiu um resultado máximo de 73,9017%, seguido de 71,1683%. Desta forma, em ambos os casos, pode se evidenciar a

Tabela 5.21: Impacto do Hiperparâmetro *activation* do MLPClassifier na classificação

Hiperparâmetro *activation*	**Repositório com Duplicados**	**Repositório sem Duplicados**
identity	85,4151	74,7165
logistic	**86,1514**	**76,4943**
tanh	85,4793	75,3211
relu	85,5554	73,9017

Fonte: Elaboração própria.
Nota: Pontuações correspondentes à acurácia balanceada em escala percentual. Sublinhado os resultados com hiperparâmetros predefinidos na Tabela 5.19 e, em negrito, os melhores resultados.

importância do otimizador *adam* no MLPClassifier ao lidar com o desbalanço do ModelSet, adaptando a taxa de aprendizado de acordo com o desempenho da classificação.

Tabela 5.22: Impacto do Hiperparâmetro *solver* do MLPClassifier na classificação

Hiperparâmetro *solver*	**Repositório com Duplicados**	**Repositório sem Duplicados**
lbfgs	84,3727	71,1683
sgd	64,5182	56,4481
adam	**85,5554**	**73,9017**

Fonte: Elaboração própria.
Nota: Pontuações correspondentes à acurácia balanceada em escala percentual. Sublinhado os resultados com hiperparâmetros predefinidos na Tabela 5.19 e, em negrito, os melhores resultados.

Neste abordagem, também foi analisado o impacto do hiperparâmetro *learning_rate_init* na classificação, variando o *learning_rate_init* segunda os espaços de busca da Tabela 5.14 e mantendo os outros hiperparâmetros da tabela com seus valores predefinidos. Como pode se ver na Tabela 5.23, em todos os casos, as pontuações são ascendentes até o ponto mais alto e depois diminuem um pouco, observando pontuações ínfimas aos extremos com *learning_rate_init* iguais a 0,0001 e 0,1. Assim, o repositório com duplicados alcançou a acurácia balanceada máxima de 86,0964% com a taxa de aprendizado inicial igual a 0,0005. E, de outra parte, o repositório sem duplicados conseguiu uma pontuação máxima de 74,9069% conforme o *learning_rate_init* igual a 0,003. Os valores ótimos do hiperparâmetro *learning_rate_init*, considerados como ligeiramente baixos, permitiram que o MLPClassifier convergisse rapidamente, sem comprometer muito a precisão.

Tabela 5.23: Impacto do Hiperparâmetro *learning_rate_init* do MLPClassifier na classificação

Hiperparâmetro ***learning_rate_init***	**Repositório com Duplicados**	**Repositório sem Duplicados**
0,0001	80,4685	68,5587
0,0002	85,5697	74,2484
0,0003	85,8326	74,3410
0,0004	85,9947	74,5148
0,0005	**86,0964**	74,3673
0,001	85,5554	73,9017
0,002	85,6939	73,6722
0,003	85,3684	**74,9069**
0,004	85,4287	74,5403
0,005	85,1967	74,6464
0,01	85,0125	74,8639
0,02	83,3774	74,8298
0,03	81,4721	74,3518
0,04	79,9509	73,4230
0,05	80,7210	72,0800
0,1	63,5298	52,4405

Fonte: Elaboração própria.
Nota: Pontuações correspondentes à acurácia balanceada em escala percentual. Sublinhado os resultados com hiperparâmetros predefinidos na Tabela 5.19 e, em negrito, os melhores resultados.

Similar aos algoritmos de classificação anteriores, foi usado o GridSearchCV, com os espaços de busca especificados na Tabela 5.19, para obter os hiperparâmetros mais ótimos do MLPClassifier. Na Tabela 5.24 pode se ver os resultados da classificação, o repositório com duplicados atingiu a acurácia balanceada máxima de 87,7533% sob uma arquitetura de uma camada oculta estabelecida conforme o *hidden_layer_sizes* igual 1300, com o hiperparâmetro *activation* igual a *relu*, com o *solver* de *sgd* e o *learning_rate_init* igual a 0,05. De outro lado, o repositório sem duplicados alcançou um desempenho ótimo de 78,7852% com *hidden_layer_sizes* igual a 1400, *activation* igual a *logistic*, *solver* de *adam* e com o hiperparâmetro *learning_rate_init* igual a 0,0002.

Tabela 5.24: Resultados da Classificação usando o MLPClassifier com Hiperparâmetros Otimizados

Hiperparâmetros Otimizados	**Repositório com Duplicados**	**Repositório sem Duplicados**
hidden_layer_sizes	1300	1400
activation	*relu*	*logistic*
solver	*sgd*	*adam*
learning_rate_init	0,05	0,0002
Acurácia balanceada	**87,7533±2,9799**	**78,7852±3,7011**

Fonte: Elaboração própria.
Nota: A acurácia balanceada é apresentada em escala percentual.

5.6 DISCUSSÕES DOS RESULTADOS

Neste capítulo, foram apresentados os experimentos realizados para avaliar a proposta de classificação os metamodelos de um repositório sob uma abordagem de aprendizado de máquina em grafos. A proposta apresenta a singularidade de usar o método de *embedding* de grafos superficiais chamado Graph2Vec desenvolvido por Narayanan et al. (2017). Neste contexto, o método foi usado para obter representações vetoriais dos metamodelos do tipo Ecore do repositório conhecido como ModelSet (López et al., 2021, 2022), para posteriormente classificá-los usando três algoritmos supervisionados: o XGBClassifier, baseado em árvores de decisão impulsionadas por gradiente; o SVC, conforme à máquina de vetores de suporte; e, o MLPCLassifier, fundado no perceptron multicamada.

Das várias implementações do Graph2Vec, na pesquisa, optou-se pela versão do Rozemberczki et al. (2020). Uma fortaleza desta versão, é que, além de usar a estrutura do grafo, o Graph2Vec pode usar um atributo dos nós no processo de *embedding*. De outra parte, uma limitação do Graph2Vec é a falta de uma métrica do desempenho. Assim, nos experimentos de *embedding* implementou-se o Graph2Vec junto com o SVC (com a estrategia de classificação multiclasse *One versus One* e os demais hiperparâmetros com seus valores predefinidos). Todos os experimentos foram avaliados por meio da validação cruzada do tipo *tenfolds*, como foi usada nos trabalhos de Couto (2023), López et al. (2022), Nguyen et al. (2021). Neste contexto, também utilizou-se a métrica nomeada como acurácia balanceada, seguindo a recomendação de Abhishek e Abdelaziz (2023) e usada em López et al. (2022).

Usando o Graph2Vec com seus hiperparâmetros predefinidos, isto é, usando só a estrutura do grafo, a acurácia balanceada foi de 21,2761% e 5,8912% para o ModelSet com e sem duplicados respectivamente. Outros experimentos foram executados adicionando o atributo dos nós nomeados como *name* imputados para o valor *None* de Python, assim, o repositório com metamodelos duplicados atingiu 48,5619% de acurácia balanceada e, de outro lado, o repositório sem duplicados alcançou um desempenho de 21,1764%. O que evidencia a importância da incorporação do atributo *name* no processo de *embedding*, atributo que também foi usado no estudo do López et al. (2022). Os resultados superiores do repositório com duplicados poderiam supor a presença de certo tipo de viés, como é sugerido por López et al. (2022). No entanto, o uso da validação cruzada contribuiu a controlar a compensação entre a variância e o viés.

A fim de obter uma representação vetorial ótima dos metamodelos, foram analisados os parâmetros do Graph2Vec número de iterações Weisfeiler-Lehman e a dimensão ou tamanho do vetor que representa ao grafo inteiro. Além disso, também foi usado o hiperparâmetro *epochs*. Sob o principio de parcimônia, pode-se evidenciar que os parâmetros mais ótimos do Graph2Vec para conseguir as representações vetoriais do ModelSet são os mesmos para os dois casos (repositório com e sem duplicados). Especificamente, o parâmetro número de iterações Weisfeiler-Lehman estabeleceu-se a dois, a dimensão do vetor em 96 e o hiperparâmetro número de *epochs* em 500.

Tendo em conta, a diferença entre os resultados, medidos pela acurácia balanceada, entre o repositório com e sem duplicados; e, de outra parte, considerando que a configuração do Graph2Vec são os mesmos para cada caso. Pode se acreditar que dita diferença é produto da classificação usando o SVC (implementado para medir o desempenho do Graph2Vec, como foi explicado anteriormente), mas não do processo de *embedding per se*. O que é justificado pelo caráter não supervisionado, transdutivo e agnóstico em relação à tarefa do Graph2Vec.

Como parte do processo do aprendizado de máquina em grafos, as representações vetoriais geradas foram usadas com os algoritmos classificadores especificados. O primeiro foi o XGBClassifier, que com seus hiperparâmetros predefinidos alcançou acurácia balanceada de

78,0905% e 60,2651% para o ModelSet com e sem duplicados respectivamente. O principal hiperparâmetro do XGBoost é o *booster*. O repositório com duplicados atingiu um resultado ótimo de 81,3224% conforme ao *booster* de tipo *xgbtree* que usa árvores de decisão e junto com os hiperparâmetros *learning_rate* igual a 0,3 e *subsample* de 0,4 conseguiram controlar o desbalanço dos dados. Paralelamente, o repositório sem duplicados obteve um desempenho de 76,2195% usando o *booster* de tipo *xgblinear* baseado em funções lineares e associados aos hiperparâmetros *learning_rate* igual a 0,02 e *subsample* de um. Deste último caso, pode se assumir que o repositório sem duplicados apresenta certa relação linear entre as dimensões dos vetores gerados e as classes alvo.

Sobre o uso dos modelos de *ensemble*, especificamente os baseados em árvores de decisão como o GBDT, como são apresentados nos estudos de Couto (2023) e de Nguyen et al. (2021), os quais usaram um repositório diferente ao ModelSet, formado por 555 metamodelos distribuídos em oito classes e, também, diferentes métodos de representação dos metamodelos como Node2Vec e o esquema de codificação do tipo *n-grams*, conseguiram atingir acurácias máximas de 90,84% e 92,92% respectivamente. Considerando que o estudo de López et al. (2022) usa o ModelSet, mas não algoritmos relacionados ao GBDT. Mesmo que os resultados achados nesta pesquisa, são menores aos outros estudos apresentados, pode se advertir sua coerência e eficacia empírica e, de outra parte, sua originalidade ao não encontrar estudos similares que implementam o GBDT junto com o ModelSet.

O segundo classificador usado foi o SVC, que estabeleceu-se com seu hiperparâmetro *class_weight* igual a *balanced* para controlar o desbalanço das classes do ModelSet junto com a estrategia OvO, o qual usou-se nestos experimentos. O SVC com seus outros hiperparâmetros predefinidos alcançou um desempenho de 87,8503% e 78,9595% para o repositório com e sem duplicados respectivamente. Os hiperparâmetros do SVC mais usados são o *C* e o tipo de *kernel*, conforme usados nos em trabalhos relacionados de Couto (2023), López et al. (2022) e Nguyen et al. (2021). Desta forma, o repositório com duplicados atingiu a acurácia balanceada ótima de 88,7378% sob uma classificação linear, isto é, com *kernel poly* de primeiro grau, com o hiperparâmetro *C* igual a um e *gamma* igual a 0,04. E, de outra parte, o repositório sem duplicados logrou um desempenho ótimo de 79,6906% conforme ao método de características de similaridade, isto é, por meio do *kernel rbf*, com o hiperparâmetro *C* igual a um e *gamma* de 0,0104.

Em comparação com os estudos de Couto (2023) e Nguyen et al. (2021), nos quais também usaram SVMs (sob seus próprios dados divididos em oito classes e representações geradas) lograram acurácias máximas de 85,37% e de 82,84% respectivamente. E, de outra parte, com o trabalho de López et al. (2022), que usa o ModelSet junto com SVC e o *TermFrequency-InverseDocumentFrequency* (TF-IDF, como mecanismo de *embedding*), alcançou acurácias balanceadas de 89,5735% e 81,5609% para o repositório com e sem duplicados respectivamente. Pode se evidenciar que os resultados atingidos nesta pesquisa, estão muito próximos aos achados por López et al. (2022). O que acredita a eficacia da proposta. Além de reconhecer que são as acurácias balanceadas mais relevantes deste estudo.

O terceiro classificador implementado foi o MLPClassifier, o qual usa de forma predefinida o hiperparâmetro *shuffle*, estabelecido como *True*, que facilitou o tratamento com o desbalanço do ModelSet. Assim, o MLPClassifier com seus hiperparâmetro predefinidos alcançou as acurácias balanceadas de 85,5554% e 73,9017% para o repositório com e sem duplicados respectivamente. Desse modo, o repositório com duplicados atingiu a acurácia balanceada ótima de 87,7533% sob uma arquitetura de uma camada oculta, estabelecida conforme o *hidden_layer_sizes* igual 1300, com o hiperparâmetro *activation* igual a *relu*, com o solver de *sgd* e o *learning_rate_init* igual a 0,05. Paralelamente, o repositório sem duplicados conseguiu

um desempenho ótimo de 78,7852% com *hidden_layer_sizes* igual a 1400, *activation* igual a *logistic*, solver de *adam* e como hiperparâmetro *learning_rate_init* igual a 0,0002. Considerando que, o MLPClassifier pode inferir automaticamente os neurônios das camada de entrada e saída. E, que, em ambos os casos, a estrutura de uma única camada oculta reduziu o *overfitting*, sendo as soluções mais eficazes sob o principio da parcimônia.

No trabalho do Couto (2023) formulou-se a ferramenta chamada MCGML baseada em um MLP, o qual alcançou a acurácia de 91,78% (com seus próprios dados divididos em oito classes e com o Node2Vec). Igualmente, Nguyen et al. (2021) implementou AURORA, uma ferramenta que usa uma rede neural artificial chamada *FeedForward Neural Network* (FFNN), muito similar ao MLP, e logrou uma acurácia igual a 92,98%. De outra parte, López et al. (2022) usou o FFNN, com o ModelSet e TF-IDF, atingindo uma acurácia balanceada de 89,8511% e 82,4972% para o repositório com e sem duplicados. Especialmente, comparando com o trabalho do López et al. (2022) pode se acreditar que os resultados são significativos para a linha de pesquisa.

Deve ter em conta que, em todos os casos, o conjunto dos hiperparâmetros otimizados diferem dos impactos individuais dos hiperparâmetros em cada classificador. Isto devido, principalmente, à sinergia entre os hiperparâmetros ao ser ajustados com o GridSearchCV. Igualmente, pode se acreditar que, a fim de conseguir bom resultados, a presente pesquisa implementou mais hiperparâmetros que as outros trabalhos relacionados.

6 CONCLUSÕES

Focados no atingimento dos objetivos da pesquisa, sobre a classificação de metamodelos de um repositório usando uma abordagem de aprendizado de máquina em grafos, foram usados o repositório ModelSet, um método de *embedding* de grafos inteiros e três algoritmos de aprendizado de máquina. Especificamente, o ModelSet, composto por metamodelos (em formato de grafos) do tipo Ecore, foi usado excluindo os metamodelos das classes *Dummy* e *Unknown*, bem como as classes que possuem menos de dez metamodelos. A fim de ter uma percepção realista, foram feitos experimentos com o ModelSet com seus dois casos (com e sem metamodelos duplicados). Para o caso do repositório com duplicados, usou-se 67 classes e 4155 metamodelos e, de outra parte, para o caso do repositório sem duplicados incluíram-se 48 classes e 2067 metamodelos. Considerando que o ModelSet, em ambos os casos, apresenta certo desbalanço entre suas classes.

Em relação ao primeiro objetivo, procurando representações vetoriais densas, de baixa dimensionalidade e de longitude fixa dos metamodelos em espaços latentes, usou-se o método de *embedding* de grafos chamado Graph2Vec proposto por Narayanan et al. (2017). Na presente dissertação, o Graph2Vec foi implementado com base na codificado de Rozemberczki et al. (2020) junto com o SVC (com seus hiperparâmetros predefinidos) e a métrica acurácia balanceada para medir o desempenho do processo de *embedding*. Depois dos experimentos, pode se evidenciar que a estrutura do grafo junto com o atributo nome dos nós (imputado para o valor de *None* de Python) geraram os resultados mais significativos. Desta forma, o repositório com duplicados alcançou a acurácia balanceada máxima de 82,2809% e, de outra parte, o repositório sem duplicados atingiu uma pontuação máxima de 73,3260%. Em ambos os casos, o Graph2Vec foi estabelecido a valores otimizados, com o parâmetro número de iterações Weisfeiler-Lehman estabelecido a dois, a dimensão do vetor em 96 e o hiperparâmetro número de *epochs* em 500.

O segundo objetivo, que visou a classificação dos *embeddings* gerados do ModelSet, sob uma abordagem multiclasse e desbalanceada, foi atingido através de três algoritmos de aprendizado de máquina supervisionados: XGBClassifier, SVC e MLPClassifier. O XGBClassifier foi usado, principalmente, com seu hiperparâmetro *booster* igual a *gbtree* para o repositório com duplicados e *gblinear* para o repositório sem duplicados. Por outra parte, implementou-se o SVC com seu hiperparâmetro *class_weight* para ajustar os pesos inversamente proporcionais às frequências das classes e, também, com o *decision_function_shape* para classificar uma classe contra as outras do repositório. E, finalmente, estabeleceu-se o MLPClassifier com uma estrutura de uma sola camada intermediaria de 1300 e de 1400 neurônios para o repositório com e sem duplicados respectivamente, além de taxas de aprendizagem ligeiramente baixas. Sendo o classificador SVC aquele que alcançou resultados ligeiramente mais relevantes dos outros. Especificamente, lograram-se acurácias balanceadas ótimas de 88,7378% para o repositório com duplicados e de 79,6906% para o repositório sem duplicados.

Considerando que o objetivo geral foi decomposto em objetivos específicos, cabe sinalar que o cumprimento dos objetivos específicos permitiu alcançar o objetivo geral com eficacia.

6.1 CONTRIBUIÇÕES

Visando sobre a classificação dos metamodelos do ModelSet usando uma abordagem de aprendizado de máquina em grafos. A presente pesquisa demonstrou, em forma eficiente, o uso adequado do método de *embedding* de grafos chamado Graph2Vec proposto por Narayanan et al. (2017) e codificado por Rozemberczki et al. (2020). Assim, o Graph2Vec gerou representações vetoriais densas, de baixa dimensionalidade e de longitude fixa dos metamodelos (com formato de grafos) em espaços latentes. Representações vetoriais que foram usadas como fonte de entrada para os três algoritmos de aprendizado de máquina supervisionados. A abordagem de grafos inteiros proposto neste estudo denota a contribuição original à linha de pesquisa, uma vez que não foram encontrados outros trabalhos relacionados que usem o Graph2Vec.

A visão superficial e transdutiva do Graph2Vec permitiu processar todos os metamodelos como um conjunto único, sem a necessidade de particionar os dados e criar um modelo de aprendizado com capacidade de generalização para dados não vistos ou novos. Além disso, o carácter não supervisionado e agnóstico em relação à tarefa, possibilitou que o Graph2Vec represente os metamodelos sem considerar os rótulos das classes e sem ter em conta o uso específico dos *embedding*. Isto último, pode se evidenciar com o uso dos mesmos valores otimizados para os hiperparâmetros do Graph2Vec nos dois casos do ModelSet (com e sem duplicados). Igualmente, coerente aos trabalhos do López et al. (2022), pode se reafirmar que a estrutura dos grafos não é suficiente para atingir uma ótima representação vetorial, por conseguinte, é imprescindível incluir o atributo *name* (parte dos nós do metamodelo Ecore) no processo de *embedding*.

Com base nos resultados significativos, achados empiricamente, tanto no processo de *embedding* quanto na classificação, pode se acreditar que a proposta desta pesquisa apresenta uma nova abordagem pragmática para a classificação automática de metamodelos. A flexibilidade da abordagem, sob os princípios de equifinalidade e de melhora continua, permite a incorporação de outras técnicas, uso de atributos dos nós ou dos enlaces.

6.2 TRABALHOS FUTUROS

Mantendo a linha de pesquisa, que visa a classificação automática de modelos e metamodelos focados na engenharia de software dirigida por modelos (MDSE) e reconhecendo que a abordagem mais efetiva para representar um modelo ou metamodelo é basada em grafos dirigidos e heterogêneos, é importante que as novas propostas de pesquisas considerem:

- No contexto do processo de *embedding* de grafos:
  - Analisar o desempenho do Graph2Vec implementando uma função de similaridade personalizada entre o grafo original e sua representação vetorial. Para tal fim, pode se usar a abordagem de codificação-decodificação ou na língua inglesa *encoder-decoder*, proposto por Hamilton (2020), para decodificar as representações vetoriais, isto é, reconstruir os grafos originais usando os *embeddings*, desta forma, quantificar a similaridade entre o grafo decodificado e o grafo original.
  - Igualmente, a fim de melhorar as representações vetoriais, sugere-se a inclusão de mais um atributo dos nós e/ou usar algumas propriedades das arestas dos grafos, usando, por exemplo, a implementação do Graph2Vec do BM2[1] (*Biological and Medical Big Data Mining Group*) da Universidade *Tongji* da China.

[1]https://github.com/bm2-lab/graph2vec

  - Comparar o desempenho do Graph2Vec com outras abordagens de *embeddings*, conforme são apresentadas na seção de Taxonomia dos Métodos de *Embeddings* de Grafos no Capítulo 2, segundo Chami et al. (2022); Stamile et al. (2021).

- Na classificação usando o aprendizado de máquina supervisionado:

  - No caso de usar o ModelSet, deve-se mitigar o impacto do desbalanço das classes do repositório conforme às técnicas de sobreamostragem, subamostragem, aumento de dados ou de aprendizado sensível ao custo ou na língua inglesa *oversampling*, *undersampling*, *data augmentation* ou *cost-sensitive learning* (CSL) respetivamente, como são especificados no Abhishek e Abdelaziz (2023).
  - Especialmente, para os algoritmos XGBClassifier e SVC, pode se implementar uma função de custo personalizada para controlar a classificação sob uma abordagem de grafos, como por exemplo usando o *Graph Kernel*, como é explicado por López et al. (2022) e Clarisó e Cabot (2018).
  - Aprofundar o uso das redes neurais em grafos ou na língua inglesa *Graph Neural Network* (GNN), com ênfase em *Graph Transformers* e *Graph Attention Networks*, como são detalhados por Khandelwal e Das (2024), Kamath et al. (2022), Wu et al. (2022) e Hamilton (2020).

Embora testar o impacto da redundância dos metamodelos do ModelSet não fosse o objetivo da pesquisa, pode se advertir que os resultados dos experimentos têm certo tipo de vieses obtendo classificações otimistas, como foi reconhecido em López et al. (2022) e López et al. (2023). Por conseguinte, é altamente recomendável usar o repositório ModelSet sem duplicados.

REFERÊNCIAS

Abhishek, K. e Abdelaziz, D. M. (2023). *Machine Learning for Imbalanced Data: Tackle Imbalanced Datasets Using Machine Learning and Deep Learning Techniques*. Packt Publishing.

Alpaydın, E. (2021). *Machine Learning: Revised and Updated Edition*. The MIT Press Essential Knowledge Series. The Massachusetts Institute of Technology Press.

Barangi, H., Kolahdouz Rahimi, S., Zamani, B. e Khasseh, A. A. (2021). Model-driven software engineering: A bibliometric analysis. *Journal of Computing and Security*, 8(1):93–108.

Barriga, A. (2021). *PARMOREL: Personalized and Automatic Repair of Models using Reinforcement Learning*. Tese de doutorado, Western Norway University of Applied Sciences, Faculty of Engineering and Science, Bergen, Norway.

Berzal, F. (2018). *Redes Neuronales & Deep Learning*. Fernando Berzal.

Brambilla, M., Cabot, J. e Wimmer, M. (2017). *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2 edition.

Bucchiarone, A., Cabot, J., Paige, R. F. e Pierantonio, A. (2020). Grand challenges in model-driven engineering: An analysis of the state of the research. *Software and Systems Modeling*, 19:5–13.

Cabot, J., Clarisó, R., Brambilla, M. e Gérard, S. (2018). Cognifying model-driven software engineering. Em Seidl, M. e Zschaler, S., editores, *Software Technologies: Applications and Foundations*, volume 10748, páginas 154–160. Springer International Publishing.

Cabot, J., Gómez, C., Pastor, O., Sancho, M.-R. e Teniente, E., editores (2017). *Conceptual Modeling Perspectives*. Springer International Publishing.

Cai, H., Zheng, V. W. e Chang, K. C.-C. (2018). A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1616–1637.

Chami, I., Abu-El-Haija, S., Perozzi, B., Ré, C. e Murphy, K. (2022). Machine learning on graphs: A model and comprehensive taxonomy. *Journal of Machine Learning Research*, 23(89):1–64.

Chen, T. e Guestrin, C. (2016). Xgboost: A scalable tree boosting system. Em *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, página 785–794. Association for Computing Machinery.

Chollet, F. (2021). *Deep Learning with Python*. Manning Publications Co, 2 edition.

Clarisó, R. e Cabot, J. (2018). Applying graph kernels to model-driven engineering problems. Em *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, MASES 2018, páginas 1–5. Association for Computing Machinery.

Couto, W. O. (2023). *Arquitetura MCGML: Classificando Modelos Desestruturados usando Aprendizado de Máquina em Grafos*. Tese de doutorado, Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, Curitiba, Brasil.

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

Da Silva, A. R. (2015). Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155.

Di Rocco, J., Di Ruscio, D., Iovino, L. e Pierantonio, A. (2015). Collaborative repositories in model-driven engineering [software technology]. *IEEE Software*, 32(3):28–34.

Domingo, Á. (2022). *Experimentation to Evaluate the Benefits of Model Driven Development*. Tese de doutorado, Universitat Politècnica de València, Departamento de Sistemas Informáticos y Computación, Valencia, España.

Goyal, P. e Ferrara, E. (2018). Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94.

Grigoriev, A. (2021). *Machine Learning Bookcamp: Build a Portfolio of Real-life Projects*. Manning Publications Co.

Grover, A. e Leskovec, J. (2016). Node2vec: Scalable feature learning for networks. Em *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, página 855–864. Association for Computing Machinery.

Géron, A. (2023). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 3 edition.

Hamilton, W. L. (2020). *Graph Representation Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Ishfaq, U., Shabbir, D., Khan, J., Khan, H. U., Naseer, S., Irshad, A., Shafiq, M. e Hamam, H. (2022). Empirical analysis of machine learning algorithms for multiclass prediction. *Wireless Communications and Mobile Computing*, 2022(7451152).

Izbicki, R. e dos Santos, T. M. (2020). *Aprendizado de Máquina: Uma Abordagem Estatística*. Rafael Izbicki.

Jiang, H. (2021). *Machine Learning Fundamentals: A Concise Introduction*. Cambridge University Press.

Jo, T. (2021). *Machine Learning Foundations: Supervised, Unsupervised, and Advanced Learning*. Springer Nature Switzerland AG.

Kamath, U., Graham, K. e Emara, W. (2022). *Transformers for Machine Learning: A Deep Dive*. Chapman & Hall/CRC Machine Learning & Pattern Recognition. CRC Press.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q. e Liu, T.-Y. (2017). LightGBM: A highly efficient gradient boosting decision tree. Em *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*, página 3149–3157. Curran Associates Inc.

Kelly, K. (2016). *The Inevitable: Understanding the 12 Technological Forces that will Shape Our Future*. Viking.

Khalilipour, A., Bozyigit, F., Utku, C. e Challenger, M. (2022). Machine learning-based model categorization using textual and structural features. Em Chiusano, S., Cerquitelli, T., Wrembel, R., Nørvåg, K., Catania, B., Vargas-Solar, G. e Zumpano, E., editores, *New Trends in Database and Information Systems. ADBIS 2022. Communications in Computer and Information Science*, volume 1652, páginas 425–436. Springer International Publishing.

Khandelwal, L. e Das, S. (2024). *Applied Deep Learning on Graphs*. Packt Publishing Ltd.

Kumar, A. e Jain, M. (2020). *Ensemble Learning for AI Developers: Learn Bagging, Stacking, and Boosting Methods with Use Cases*. Apress Berkeley.

Kunapuli, G. (2023). *Ensemble Methods for Machine Learning*. Manning Publications Co.

Lavrač, N., Podpečan, V. e Robnik-Šikonja, M. (2021). *Representation Learning: Propositionalization and Embeddings*. Springer Nature Switzerland AG.

Le, Q. V. e Mikolov, T. (2014). Distributed representations of sentences and documents. arXiv preprint arXiv:1405.4053.

Lee, W.-M. (2019). *Python Machine Learning*. John Wiley & Sons.

Liu, Z. e Zhou, J. (2020). *Introduction to Graph Neural Networks*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

López, J. A. H., Cánovas Izquierdo, J. L. e Cuadrado, J. S. (2021). ModelSet: A dataset for machine learning in model-driven engineering. *Software and Systems Modeling*, 21(3):967–986.

López, J. A. H., Durá, C. e Cuadrado, J. S. (2023). Word embeddings for model-driven engineering. Em *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, páginas 151–161. IEEE Computer Society.

López, J. A. H., Rubei, R., Cuadrado, J. S. e Ruscio, D. D. (2022). Machine learning methods for model classification: A comparative study. Em *Proceedings of the 25th International Conference on Model Driven Engineering Languages and System (MODELS '22)*, páginas 165–175. Association for Computing Machinery.

Mikolov, T., Chen, K., Corrado, G. e Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

Mohri, M., Rostamizadeh, A. e Talwalkar, A. (2018). *Foundations of Machine Learning*. Adaptive Computation and Machine Learning Series. The MIT Press, 2 edition.

Moin, A., Challenger, M., Badii, A. e Günnemann, S. (2022). A model-driven approach to machine learning and software modeling for the IoT. *Software and Systems Modeling*, 21:987–1014.

Molina, J., Rubio, F., Pelechano, V., Vallecillo, A., Vara, J. e Vicente-Chicote, C. (2012). *Desarrollo de Software Dirigido por Modelos: Conceptos, Métodos y Herramientas*. Ra-Ma Editorial.

Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y. e Jaiswal, S. (2017). Graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005*.

Negro, A. (2021). *Graph-Powered Machine Learning*. Manning Publications Co.

Nguyen, P. T., Di Rocco, J., Di Ruscio, D., Pierantonio, A. e Iovino, L. (2019). Automated classification of metamodel repositories: A machine learning approach. Em *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, páginas 272–282.

Nguyen, P. T., Di Rocco, J., Iovino, L., Di Ruscio, D. e Pierantonio, A. (2021). Evaluation of a machine learning classifier for metamodels. *Software and Systems Modeling*, 20:1797–1821.

Owen, L. (2022). *Hyperparameter Tuning with Python: Boost your machine learning model's performance via hyperparameter tuning*. Packt Publishing.

Pons, C., Giandini, R. e Pérez, G. (2010). *Desarrollo de software dirigido por modelos: Conceptos teóricos y su aplicación práctica*. Universidad Nacional de La Plata.

Pouyanfar, S., Sadiq, S., Yan, Y., Tian, H., Tao, Y., Reyes, M. P., Shyu, M., Chen, S. e Iyengar, S. S. (2018). A survey on deep learning: Algorithms, techniques, and applications. *ACM Computing Surveys*, 51(5):92:1–36.

Pressman, R. S. e Maxim, B. R. (2020). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 9 edition.

Rozemberczki, B., Kiss, O. e Sarkar, R. (2020). Karate club: An api oriented open-source python framework for unsupervised learning on graphs. Em *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*, página 3125–3132.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229.

Sarker, I. H. (2021). Machine learning: Algorithms, real-world applications and research directions. *SN Computer Science*, 2(160).

Sommerville, I. (2016). *Software Engineering*. Always Learning. Pearson Higher Education, 10 edition.

Stamile, C., Marzullo, A. e Deusebio, E. (2021). *Graph Machine Learning: Take graph data to the next level by applying machine learning techniques and algorithms*. Packt Publishing Ltd.

Steinberg, D., Budinsky, F., Paternostro, M. e Merks, E. (2009). *EMF Eclipse Modeling Framework*. Pearson Education, 2 edition.

Wade, C. e Glynn, K. (2020). *Hands-On Gradient Boosting with XGBoost and Scikit-Learn: Perform Accessible Machine Learning and Extreme Gradient Boosting with Python*. Packt Publishing Ltd.

Wang, C., Wang, C., Wang, Z., Ye, X. e Yu, P. S. (2020a). Edge2vec: Edge-based social network embedding. *Association for Computing Machiner*, 14(4).

Wang, Y., Xu, B., Kwak, M. e Zeng, X. (2020b). A simple training strategy for graph autoencoder. Em *Proceedings of the 2020 12th International Conference on Machine Learning and Computing*, página 341–345. Association for Computing Machinery.

Whittle, J., Hutchinson, J. e Rouncefield, M. (2014). The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85.

Wu, L., Cui, P., Pei, J. e Zhao, L., editores (2022). *Graph Neural Networks: Foundations, Frontiers, and Applications*. Springer Nature Singapore Pte Ltd.

Yang, C., Liu, Z., Tu, C., Shi, C. e Sun, M. (2021). *Network Embedding Theories, Methods, and Applications*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Yousaf, N., Akram, M., Bhatti, A. e Zaib, A. (2019). Investigation of tools, techniques and languages for model driven software product lines (SPL): A systematic literature review. *Journal of Software Engineering and Applications*, 12:293–306.

Zheng, A. e Casari, A. (2018). *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. O'Reilly Media, Inc.

Zhou, J., Liu, L., Wei, W. e Fan, J. (2021). Network representation learning: From preprocessing, feature extraction to node embedding. *ACM Computing Surveys*, 55(2):1–35.

ANEXO A - EXEMPLO DE UM METAMODELO REPRESENTADO COMO GRAFO


```
{
   "directed":true,
   "nodes":
   [
      {
         "nsPrefix":"lib",
         "nsURI":"www.library.com",
         "name":"Library",
         "id":0,
         "eClass":"EPackage"
      },
      {
         "instanceTypeName":null,
         "defaultValue":null,
         "instanceClassName":null,
         "name":"Library",
         "instanceClass":null,
         "abstract":false,
         "id":1,
         "interface":false,
         "eClass":"EClass"
      },
      {
         "instanceTypeName":null,
         "defaultValue":null,
         "instanceClassName":null,
         "name":"Book",
         "instanceClass":null,
         "abstract":false,
         "id":2,
         "interface":false,
         "eClass":"EClass"
      },
      {
         "container":false,
         "ordered":true,
         "upperBound":-1,
         "defaultValue":null,
         "volatile":false,
         "many":true,
         "required":false,
         "eClass":"EReference",
         "defaultValueLiteral":null,
         "containment":true,
         "unsettable":false,
         "transient":false,
         "unique":true,
         "name":"books",
         "changeable":true,
         "resolveProxies":true,
         "lowerBound":0,
         "id":3,
         "derived":false
```

```
        },
        {
           "id":4,
           "eClass":"EGenericType"
        },
        {
           "ordered":true,
           "upperBound":1,
           "defaultValue":null,
           "volatile":false,
           "many":false,
           "required":false,
           "eClass":"EAttribute",
           "defaultValueLiteral":null,
           "unsettable":false,
           "transient":false,
           "unique":true,
           "name":"name",
           "changeable":true,
           "lowerBound":0,
           "iD":false,"id":5,
           "derived":false
        },
        {
           "id":6,
           "eClass":"EGenericType"
        }
    ],
    "links":
    [
        {"source":0,"target":1},
        {"source":0,"target":2},
        {"source":1,"target":0},
        {"source":1,"target":3},
        {"source":2,"target":0},
        {"source":2,"target":5},
        {"source":3,"target":2},
        {"source":3,"target":4},
        {"source":3,"target":1},
        {"source":4,"target":2},
        {"source":5,"target":6},
        {"source":5,"target":2}
    ],
    "multigraph":true
}
```

APÊNDICE A – RESULTADOS DOS EXPERIMENTOS INICIAIS COM VARIAÇÃO DE HIPERPARÂMETROS DO GRAPH2VEC

Tabela A.1: Impacto da Dimensão do Vetor e do Número de Iterações Weisfeiler-Lehman do Graph2Vec na Representação Vetorial do ModelSet com Duplicados

Épocas	**Dimensão do Vetor**																**Máximo**	**Média**	**Mínimo**
	2	**4**	**8**	**16**	**32**	**64**	**96**	**128**	**192**	**256**	**384**	**512**	**640**	**768**	**896**	**1024**			
1	6,3297	23,1644	31,9063	44,7272	46,2199	47,4029	46,4325	46,5315	44,6118	43,3056	42,2667	40,8211	39,5536	39,2189	38,8984	39,0769	47,4029	38,7792	6,3297
2	5,6308	18,8468	31,8168	45,8093	48,9405	**49,8510**	**49,1240**	**48,5619**	**47,8966**	47,3719	46,4691	45,0668	44,1724	45,3792	45,0463	43,6835	**49,8510**	**41,4792**	**5,6308**
4	5,9357	16,3849	29,3004	44,4889	48,7538	**49,9091**	**49,0054**	49,5375	**48,3575**	47,9580	46,7062	47,7083	46,5315	46,0896	45,8977	45,6291	**49,9091**	**41,7621**	**5,9357**
8	4,9189	15,1268	27,3088	42,4494	48,8360	**50,2806**	**50,2063**	**50,0465**	**49,4258**	48,3112	48,1181	47,0731	47,4200	46,7355	46,6881	45,7928	**50,2806**	**41,7961**	**4,9189**
16	4,4991	14,3798	24,0761	36,5006	46,3053	49,1752	49,3003	49,2177	48,7896	48,0109	47,3566	47,1727	46,7340	47,2905	46,4699	45,6081	49,3003	40,6804	4,4991
24	4,5994	12,7050	22,1091	33,8176	43,9769	47,9367	47,8472	47,6482	48,0386	47,2475	46,8213	46,1251	44,8652	46,0369	44,3729	44,4055	48,0386	39,2846	4,5994
32	4,4264	12,0489	21,2613	31,8053	42,6872	46,8056	46,8166	46,4682	46,0222	44,6003	44,2826	43,6688	44,0553	42,8650	42,0446	42,4572	46,8166	37,6447	4,4264
48	4,0440	8,7055	18,8948	29,3709	39,4663	42,8024	43,4226	44,5280	43,8803	43,8949	43,3511	42,8250	43,5130	42,8039	42,4585	41,8248	44,5280	35,9866	4,0440
64	4,2644	8,0974	18,5968	27,0987	38,6944	41,9134	42,3427	42,8419	42,4487	42,4196	42,1068	42,0695	40,8484	40,8224	40,4389	40,5513	42,8419	34,7222	4,2644
Máximo	6,3297	23,1644	31,9063	45,8093	48,9405	**50,2806**	**50,2063**	**50,0465**	**49,4258**	48,3112	48,1181	47,7083	47,4200	47,2905	46,6881	45,7928			
Média	4,9609	14,3844	25,0300	37,3409	44,8756	**47,3419**	**47,1664**	**47,2646**	**46,6079**	45,9022	45,2754	44,7256	44,1881	44,1380	43,5906	43,2255			
Mínimo	4,0440	8,0974	18,5968	27,0987	38,6944	**41,9134**	**42,3427**	**42,8419**	**42,4487**	42,4196	42,1068	40,8211	39,5536	39,2189	38,8984	39,0769			

Fonte: Elaboração própria.

Nota: Pontuações correspondentes à acurácia balanceada em escala percentual. Sublinhado os resultados com hiperparâmetros predefinidos do Graph2Vec e, em negrito, os melhores resultados.

Tabela A.2: Impacto da Dimensão do Vetor e do Número de Iterações Weisfeiler-Lehman do Graph2Vec na Representação Vetorial do ModelSet sem Duplicados

Épocas	**Dimensão do Vetor**																**Máximo**	**Média**	**Mínimo**
	2	**4**	**8**	**16**	**32**	**64**	**96**	**128**	**192**	**256**	**384**	**512**	**640**	**768**	**896**	**1024**			
1	7,6349	11,7811	18,1941	20,4579	21,5516	19,1739	18,1098	17,3624	16,3911	15,6164	14,3800	14,2391	13,3487	13,6020	13,5559	13,2499	21,5516	15,5406	7,6349
2	7,0472	10,0446	19,3959	21,0436	**23,0342**	**22,4828**	**21,3497**	**21,1764**	19,4907	18,2256	17,1089	17,1773	16,5024	15,9187	15,7362	15,7579	textbf23,0342	**17,5932**	**7,0472**
4	5,7734	10,5976	15,9375	20,3158	**22,7260**	**21,9202**	**21,5044**	**21,5939**	20,5910	19,2343	18,7744	17,9099	18,2579	16,8754	16,9475	16,6492	**22,7260**	**17,8505**	**5,7734**
8	5,5479	8,7577	11,4678	15,2520	20,4073	20,4834	20,9591	20,1942	19,6172	18,7028	18,5052	19,0023	18,2869	18,3584	18,2164	18,0200	20,9591	16,9862	5,5479
16	6,0295	8,7719	10,0547	13,2532	18,7804	19,2356	20,1716	19,6292	19,5440	19,6615	17,9772	18,4791	16,7838	17,4369	17,4612	17,6273	20,1716	16,3061	6,0295
24	5,7472	8,2408	9,4137	11,1185	17,3239	18,8673	19,8843	19,5862	19,2921	18,6609	17,9242	18,3709	16,7504	16,9288	16,7359	17,2200	19,8843	15,7541	5,7472
32	5,4379	7,3478	8,8413	10,9745	16,7558	17,0445	18,3237	19,7647	18,2564	18,6518	18,0608	17,9401	17,5017	17,1149	17,6977	16,6242	19,7647	15,3961	5,4379
48	5,6012	6,4461	8,0779	10,3854	13,9503	16,2948	17,2076	17,7341	17,4802	17,9309	16,7917	17,3758	17,2658	16,7194	15,9040	16,3309	17,9309	14,4685	5,6012
64	5,3586	6,2571	7,4696	8,7965	13,1707	15,6535	16,0301	16,2077	15,9435	16,1720	14,9103	15,8735	15,6634	15,8755	15,1484	14,9477	16,2077	13,3424	5,3586
Máximo	7,6349	11,7811	19,3959	21,0436	**23,0342**	**22,4828**	**21,5044**	**21,5939**	20,5910	19,6615	18,7744	19,0023	18,2869	18,3584	18,2164	18,0200			
Média	6,0197	8,6939	12,0947	14,6219	**18,6334**	**19,0173**	**19,2823**	**19,2499**	18,5118	18,0951	17,1592	17,3742	16,7068	16,5367	16,3781	16,2697			
Mínimo	5,3586	6,2571	7,4696	8,7965	**13,1707**	**15,6535**	**16,0301**	**16,2077**	15,9435	15,6164	14,3800	14,2391	13,3487	13,6020	13,5559	13,2499			

Fonte: Elaboração própria.

Nota: Pontuações correspondentes à acurácia balanceada em escala percentual. Sublinhado os resultados com hiperparâmetros predefinidos do Graph2Vec e, em negrito, os melhores resultados.