

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
CURSO DE ENGENHARIA MECATRÔNICA

VITOR GABRIEL ALÉSSIO NOGARA

ANÁLISE DE COMPLEXIDADE DE CÓDIGOS COM REDE NEURAL DE GRAFOS

Joinville
2026

VITOR GABRIEL ALÉSSIO NOGARA

ANÁLISE DE COMPLEXIDADE DE CÓDIGOS COM REDE NEURAL DE GRAFOS

Trabalho apresentado como requisito parcial para obtenção do título de bacharel em Engenharia Mecatrônica, no Curso de Engenharia Mecatrônica do Centro Tecnológico de Joinville, da Universidade Federal de Santa Catarina.

Orientador: Ricardo José Pfitscher

Joinville
2026

RESUMO

A complexidade de código é uma característica diretamente relacionada ao tempo de execução de um código e a sua eficiência. Obter uma eficiência maior ou tempo de execução menor, são objetivos comuns em qualquer área na engenharia, mas esses objetivos se tornam preponderantes na análise de sistemas de alta precisão e em aplicações de tempo real. A identificação automatizada da complexidade ou de trechos problemáticos, poderia auxiliar nesses objetivos e otimizar os processos de desenvolvimento de algoritmos, porém, no problema da parada de Alan Turing, provou-se que é impossível a criação de um método ou função genérico capaz de calcular a complexidade exata de qualquer programa. Essa barreira teórica impõe a necessidade de investigar uma diferente abordagem baseada na estimativa de complexidade. Assim, neste trabalho analisa-se a viabilidade de Redes Neurais de Grafo (GNN) para essa tarefa, implementando-se diferentes modelos e testando-os e avaliando-os sobre diferentes métricas. Desenvolveu-se o modelo *ComplexityGNN*, treinado e avaliado sobre 37608 exemplos em oito linguagens, atingindo acurácia de 67,6% na classificação por classe e AUC-ROC de 0,926 na classificação binária de eficiência, demonstrando a viabilidade da abordagem para estimativa de complexidade de código.

Palavra-chave: complexidade de código; tempo de execução; redes neurais de grafo.

ABSTRACT

Code complexity is a characteristic directly related to the execution time and efficiency of a program. Achieving higher efficiency or shorter execution times are common objectives in any engineering field, but these goals become paramount in the analysis of high-precision systems and real-time applications. The automated identification of complexity or problematic code segments could assist in these objectives and optimize algorithm development processes. However, as demonstrated by Alan Turing's halting problem, it is theoretically impossible to create a generic method or function capable of calculating the exact complexity of any arbitrary program. This theoretical barrier imposes the need to investigate an alternative approach based on complexity estimation. Thus, this work analyzes the feasibility of Graph Neural Networks (GNN) for this task by implementing, testing, and evaluating different models across various metrics. The model *ComplexityGNN* was developed, trained and tested under 37608 examples in 8 programming languages, measuring 67,6% accuracy estimating by classes and AUC-ROC of 0,926 in binary classification of efficiency, presenting the viability of the approach of estimation of complexity.

Keywords: Code complexity; execution time; graph neural networks.

LISTA DE FIGURAS

Figura 1 – Isômeros <i>cis</i> e <i>trans</i> do but-2-eno.	23
Figura 2 – Fluxograma da metodologia.	32
Figura 3 – Conversão dos conjuntos de dados para estrutura de diretórios. . .	34
Figura 4 – Fluxo de normalização de rótulos de complexidade.	35
Figura 5 – Extração de átomos a partir de uma função de busca linear em Python.	39
Figura 6 – Arestas estruturais de enlace e sequência sobre os átomos da função de busca linear.	40
Figura 7 – Grafo de Fluxo de Controle da função de busca linear.	41
Figura 8 – Arestas de dependência de dados para a variável <i>i</i> na função de busca linear.	42
Figura 9 – Registro de uma molécula na biblioteca e consulta durante a conversão para tensores.	46
Figura 10 – Segmentação de um arquivo com duas funções em dois exemplos de treinamento independentes.	49
Figura 11 – Conversão de um sub-grafo de função em objeto Data.	52
Figura 12 – Fluxo de processamento na configuração padrão.	54

LISTA DE QUADROS

Quadro 1 – Classes de complexidade de algoritmos e exemplos.	17
Quadro 2 – Trabalhos relacionados e suas contribuições.	30
Quadro 3 – Módulos de conversão dos conjuntos de dados.	33
Quadro 4 – Módulos do processo de análise topológica	37
Quadro 5 – Tipos de aresta do Grafo de Fluxo de Controle.	41
Quadro 6 – Pesos de aresta para detecção de comunidades.	47
Quadro 7 – Blocos do vetor de características de nó (configuração padrão: 29 dimensões).	51
Quadro 8 – Dimensões dos componentes do <i>ComplexityGNN</i> na configuração de referência.	58
Quadro 9 – Hiperparâmetros padrão da <i>GNNConfig</i>	59
Quadro 10 – Configurações possíveis do critério de perda via <i>build_criterion</i> . . .	63
Quadro 11 – Comparação metodológica entre os trabalhos avaliados.	73

LISTA DE TABELAS

Tabela 1 – Classes de complexidade de tempo adotadas e exemplos de rótulos normalizados.	35
Tabela 2 – Divisão do conjunto de dados para treinamento.	58
Tabela 3 – Tempo de processamento por passe de análise sintática (2842 arquivos, média de 0,91 s/arquivo).	65
Tabela 4 – Distribuição de exemplos por classe no conjunto de validação (5641 exemplos).	66
Tabela 5 – Métricas de avaliação do modelo de referência no conjunto de validação.	67
Tabela 6 – Precisão, <i>recall</i> e F1 por classe no conjunto de validação.	67
Tabela 7 – Efeito de cada bloco de características de nó.	68
Tabela 8 – Comparação entre tipos de camada e profundidade.	69
Tabela 9 – Comparação entre estratégias de agrupamento.	70
Tabela 10 – Comparação entre configurações de tipos de aresta.	70
Tabela 11 – Efeito da remoção de componentes de contexto global.	71
Tabela 12 – Evolução das configurações ótimas ao longo das séries de experimentos.	72
Tabela 13 – Comparação de acurácia entre os trabalhos avaliados.	74
Tabela 14 – Acurácia de classe em condições de generalização entre fontes de dados distintas.	75

LISTA DE ABREVIATURAS E SIGLAS

ANN	<i>Artificial Neural Network</i> (Rede Neural Artificial)
AST	<i>Abstract Syntax Tree</i> (Árvore de Sintaxe Abstrata)
AUC-ROC	<i>Area Under the Receiver Operating Characteristic Curve</i>
BN	<i>Batch Normalization</i> (Normalização por Lote)
CFG	<i>Control Flow Graph</i> (Grafo de Fluxo de Controle)
CUDA	<i>Compute Unified Device Architecture</i>
GAT	<i>Graph Attention Network</i> (Rede Neural de Atenção de Grafo)
GCN	<i>Graph Convolutional Network</i> (Rede Convolucional de Grafo)
GFC	Grafo de Fluxo de Controle
GFG	<i>GeeksForGeeks</i> (conjunto de dados)
GIN	<i>Graph Isomorphism Network</i> (Rede de Isomorfismo de Grafo)
GNN	<i>Graph Neural Network</i> (Rede Neural de Grafo)
GPU	<i>Graphics Processing Unit</i> (Unidade de Processamento Gráfico)
IA	Inteligência Artificial
JK	<i>Jumping Knowledge</i>
JSON	<i>JavaScript Object Notation</i>
JSONL	<i>JSON Lines</i>
KNN	K-Vizinhos Mais Próximos (<i>K-Nearest Neighbors</i>)
LSTM	<i>Long Short-Term Memory</i>
MAOE	<i>Mean Absolute Ordinal Error</i> (Erro Absoluto Ordinal Médio)
ML	<i>Machine Learning</i> (Aprendizado de Máquina)
MLP	<i>Multilayer Perceptron</i> (Perceptron Multicamadas)
MPNN	<i>Message Passing Neural Network</i> (Rede neural com passagem de mensagens)

PDG	<i>Program Dependence Graph</i> (Grafo de Dependência do Programa)
PyG	PyTorch Geometric
RAM	<i>Random Access Memory</i> (Memória de Acesso Aleatório)
ReLU	<i>Rectified Linear Unit</i> (Unidade Linear Retificada)
RNA	Rede Neural Artificial
SMOTE	<i>Synthetic Minority Oversampling Technique</i>
SVM	<i>Support Vector Machine</i> (Máquina de Vetores de Suporte)
XGBoost	<i>Extreme Gradient Boosting</i> (Aumento de Gradiente Extremo)

LISTA DE SÍMBOLOS

$G = (V, A)$	Grafo com conjunto de vértices V e conjunto de arestas A
$\mathcal{N}(v)$	Conjunto de vizinhos do nó v
$\mathcal{N}_s(v)$	Subconjunto amostrado de vizinhos de v (GraphSAGE)
$\hat{d}_v$	Grau do nó v com auto-conexão: $\hat{d}_v = 1 + \deg(v)$
$\mathbf{h}_v^{(k)}$	Representação vetorial (<i>embedding</i>) do nó v na camada k
$\mathbf{x}_v$	Vetor de características de entrada do nó v ($\mathbf{x}_v \in R^{29}$)
$\mathbf{e}_{uv}$	Vetor de características da aresta de u para v ($\mathbf{e}_{uv} \in R^{13}$)
$\mathbf{W}^{(k)}$	Matriz de pesos da camada k
$\varepsilon^{(k)}$	Parâmetro escalar aprendido da GIN: controla o peso da auto-representação em relação às mensagens de vizinhos na camada k
$\alpha_{vu}^{(k)}$	Coefficiente de atenção da GAT: importância do nó u para v na camada k
$\phi(\cdot, \cdot)$	Função de combinação mensagem-aresta do GINEConv
H	Dimensão oculta dos vetores de representação ($H = 64$)
L	Número de camadas de passagem de mensagens ($L = 4$)
C	Número de classes de complexidade ($C = 8$)
N	Número de nós em um grafo de função
B	Tamanho do lote de treinamento ($B = 16$)
η	Taxa de aprendizado (inicial: $\eta = 10^{-3}$)
λ	Coefficiente de regularização L2 ($\lambda = 10^{-4}$)
$\sigma(\cdot)$	Função de ativação (ReLU ou <i>softmax</i> conforme contexto)
$\mathbf{p}_G$	Vetor de representação global após agrupamento MEAN+MAX+SUM ($\mathbf{p}_G \in R^{3H}$)
$\mathbf{z}$	Vetor de entrada do classificador ($\mathbf{z} \in R^{226}$)
$\hat{\mathbf{y}}$	Vetor de <i>logits</i> de saída ($\hat{\mathbf{y}} \in R^8$)

$\hat{p}_c$	Probabilidade <i>softmax</i> da classe c
$\mathcal{L}_{\text{CE}}$	Perda de entropia cruzada com pesos de classe
$P_{\text{alvo}[k]}$	Distribuição alvo com suavização ordinal de rótulos
w	Peso da penalidade ordinal na perda combinada
κ	Coeficiente de concordância de Cohen
p_o	Proporção de acertos observados no cálculo de κ
p_e	Proporção de acertos esperados por acaso no cálculo de κ
r_c	Ranque normalizado da classe c : $r_c = c / (C - 1)$, $r_c \in [0, 1]$
$\hat{r}_i$	Ranque esperado segundo a distribuição prevista para o exemplo i
Q	Modularidade da comunidade maximizada pelo algoritmo de Louvain

SUMÁRIO

1	INTRODUÇÃO	13
1.1	OBJETIVOS	14
1.1.1	Objetivo geral	14
1.1.2	Objetivos Específicos	14
2	FUNDAMENTAÇÃO TEÓRICA	16
2.1	COMPLEXIDADE DE TEMPO DE EXECUÇÃO	16
2.2	INTELIGÊNCIA ARTIFICIAL	17
2.2.1	Aprendizado de máquina	17
2.2.2	Métodos de classificação e regressão	18
2.2.3	Redes neurais	19
2.2.4	Perceptron multicamadas	19
2.2.5	Redes neurais de grafo	20
2.2.5.1	Arquiteturas de Redes Neurais de Grafo	20
2.2.5.2	Técnicas de aprendizado de redes	22
2.3	TOPOLOGIAS	23
2.4	BANCO DE DADOS DE CÓDIGOS E COMPLEXIDADE	24
2.5	MÉTRICAS DE AVALIAÇÃO	24
2.6	FERRAMENTAS AUXILIARES	26
2.6.1	Processamento de código-fonte	26
2.6.2	Representação e manipulação de grafos	27
2.6.3	Aprendizado profundo em grafos	27
2.6.4	Otimização	27
2.7	CONCEITOS DA METODOLOGIA	27
2.7.1	Divisão dos dados	28
2.7.2	Função de perda	28
2.7.3	Regularização	28
2.8	TRABALHOS RELACIONADOS	29
3	METODOLOGIA	31
3.1	PREPARAÇÃO DA BASE DE DADOS	33
3.2	ROTULAÇÃO DE COMPLEXIDADE	34
3.3	ANÁLISE SINTÁTICA E LEITURA TOPOLÓGICA	35
3.3.1	Extração de átomos	37
3.3.2	Grafo de Dependência Estrutural	40
3.3.3	Grafo de Fluxo de Controle	40
3.3.4	Grafo de Dependência de Dados	41
3.3.5	Grafo de Dominância	42

3.3.6	Detecção de Recursão	43
3.3.7	Assinatura Molecular e Biblioteca de Topologias	43
3.3.8	Votos de Complexidade por Átomo	45
3.3.9	Detecção de Comunidades	47
3.3.10	Segmentação por Função	48
3.4	CONVERSÃO PARA TENSORES	50
3.5	ARQUITETURA DA REDE NEURAL DE GRAFOS	52
3.6	CONFIGURAÇÃO E TREINAMENTO DA REDE NEURAL	58
3.7	DECLARAÇÃO DE USO DE INTELIGÊNCIA ARTIFICIAL	63
4	RESULTADOS E DISCUSSÕES	64
4.1	CONFIGURAÇÃO EXPERIMENTAL	64
4.2	RESULTADOS DO MODELO	66
4.3	ESTUDOS DE ABLAÇÃO	68
4.3.1	Características de nó	68
4.3.2	Tipo de Camada	69
4.3.3	Estratégia de agrupamento	69
4.3.4	Tipos de aresta	70
4.3.5	Contexto e escalares globais	71
4.3.6	Configuração ótima	71
4.4	COMPARAÇÃO COM TRABALHOS RELACIONADOS	72
4.4.1	Resultados sobre o conjunto GeeksForGeeks	73
4.5	DISCUSSÃO	75
5	CONCLUSÃO	77
5.1	ATENDIMENTO AOS OBJETIVOS ESPECÍFICOS	77
5.2	CONTRIBUIÇÕES	78
5.3	LIMITAÇÕES	78
5.4	TRABALHOS FUTUROS	79
	REFERÊNCIAS	81

1 INTRODUÇÃO

A computação pode ser compreendida como a atividade de processar dados por meio de algoritmos para se obter resultados (CORMEN *et al.*, 2009, p. 17). Desde a sua origem, como qualquer atividade na engenharia, busca-se otimizar esse processo, sendo que um dos parâmetros críticos para aferir a eficiência é o tempo de execução, característica essa em relação direta com o grau de complexidade do algoritmo (KLEINBERG; TARDOS, 2006, p. 33).

Com a evolução de modelos computacionais teóricos, como a Máquina de Turing, e o advento da arquitetura de Von Neumann, novas abordagens surgiram para lidar com essa complexidade, máquinas computacionais popularizaram-se e expandiram-se para outras aplicações, resultando em novos algoritmos complexos. Tal expansão acentuou os desafios impostos por algoritmos com complexidade elevada, tornando intratáveis problemas de grande escala, como o problema do caixeiro viajante, usado na otimização de rotas em logística, e a fatoração de inteiros grandes, usado para criptografia de chaves públicas (KLEINBERG; TARDOS, 2006, p. 474).

A análise da complexidade de algoritmos é uma ferramenta que pode ser usada para compreender e mitigar esses desafios, entretanto, a análise manual é um processo laborioso e suscetível a erros, enquanto a predição exata do tempo de execução para um programa arbitrário é teoricamente impossível, uma consequência direta do Problema da Parada (LUCAS, 2021). Diante deste cenário, a comunidade científica tem explorado abordagens de Inteligência Artificial (IA) para automatizar ou auxiliar a estimativa da complexidade de código.

Alguns dos esforços nessa área aplicaram técnicas de aprendizado de máquina tradicional e análise estática. Pfitscher *et al.* (2023) e Sikka *et al.* (2020) utilizaram modelos como Floresta Aleatória e Redes Neurais sobre um conjunto de características extraídas estaticamente do código para classificar sua complexidade. Paralelamente, ferramentas de análise estática, como a biblioteca Radon para Python, focam em métricas como a complexidade ciclomática, que mede o número de caminhos linearmente independentes através do código, oferecendo uma perspectiva sobre a complexidade estrutural, mas não diretamente sobre a complexidade de tempo assintótica (MCCABE, 1976).

Outra ferramenta que utiliza técnicas de aprendizado e que recentemente tem sido considerada promissora são as Redes Neurais de Grafo (GNN), em especial para processar dados onde as relações entre entidades são fundamentais (XU *et al.*, 2019). Essa característica é aplicável em uma gama de domínios, desde redes sociais até o design de arquitetura (BLOCH *et al.*, 2023). No contexto da engenharia de software, GNNs têm sido aplicadas com sucesso para tarefas como garantia de privacidade em

dados estruturados, detecção de vulnerabilidades e previsão de defeitos em código (DAIGAVANE *et al.*, 2021; HIN *et al.*, 2022; SIKIC *et al.*, 2022).

As GNNs possuem diferentes arquiteturas, como Redes Neurais Convolucionais de Grafo (GCN) ou Redes Neurais de Grafo com mecanismos de Atenção (GAT), cada uma utilizando dados em diferentes representações (BIELIK *et al.*, 2016). Diferentes representações podem ser transformadas em grafos de diferentes maneiras, códigos fonte, por exemplo, podem ser representados através de sua semântica, em uma Árvore de Sintaxe Abstrata (AST) ou pelo caminho de comando executados, em um Grafo de Fluxo de Controle (CFG) (BIELIK *et al.*, 2016).

A depender da representação escolhida, dados em forma de relações podem ser explorados e enfatizados. No caso de códigos, as relações estruturais e semânticas, como a forma em que laços de repetição estão aninhados ou como funções se invocam recursivamente, são fundamentais para determinar a complexidade de um código (CORMEN *et al.*, 2009, p. 159). Uma vez que GNNs são projetadas para aprender características inerentes dessas relações, obtém-se a motivação para a aplicação de GNN na análise de complexidade temporal de códigos (WU, 2021). Espera-se assim desenvolver uma ferramenta para estimativa de complexidade e investigar seu desempenho frente aos métodos já publicados na literatura. Para atingir esse objetivo, a metodologia será dividida em quatro etapas principais.

Primeiramente, será realizada a conversão de códigos-fonte de bancos de dados de códigos públicos para diferentes representações de grafo, como a AST e o CFG. A seguir, será feita a extração e a engenharia de características de nós e arestas. Na terceira etapa, serão implementados e treinados diferentes modelos de GNN, como GCN e GAT. Por fim, os modelos serão sistematicamente avaliados e comparados com meios existentes por meio de métricas como acurácia e F1-score.

1.1 OBJETIVOS

Para a classificação correta da complexidade de códigos, foram definidos os seguintes objetivos.

1.1.1 Objetivo geral

Investigar e desenvolver um método baseado em Redes Neurais de Grafo para a classificação automática da complexidade de tempo de algoritmos a partir de seu código-fonte, avaliando sua eficácia e limitações em comparação com abordagens existentes.

1.1.2 Objetivos Específicos

- Examinar diferentes representações de código-fonte para grafos;

- Tratar um conjunto de dados públicos para o treinamento e a validação do modelo proposto;
- Empregar modelos de aprendizado de máquina baseados em GNN em cima dos dados tratados;
- Avaliar experimentalmente o desempenho dos modelos propostos a partir dos critérios escolhidos;
- Comparar os experimentos com modelos já existentes de classificação de complexidade de códigos.

2 FUNDAMENTAÇÃO TEÓRICA

A fim de desenvolver um modelo capaz de estimar a complexidade de um código, é necessária a compreensão das características de um, bem como o entendimento conceitual dos métodos que serão aplicados na tarefa. Este capítulo tem como objetivo apresentar esses conceitos fundamentais, sendo abordados nesse os princípios de complexidade, inteligência artificial e o aprendizado de máquina. Explora-se também as redes neurais, com foco especial nas redes neurais de grafo. Por fim, conceitua-se as métricas de avaliação e os trabalhos relacionados, aos quais compara-se a performance do modelo estabelecido.

2.1 COMPLEXIDADE DE TEMPO DE EXECUÇÃO

O tempo de execução refere-se ao intervalo de tempo entre o início e o término da execução de um algoritmo por um sistema computacional. De acordo com Cormen *et al.* (2009, p. 22), esse tempo não é uma constante, sendo influenciado por um conjunto de fatores, como especificações do computador ou até o ambiente de programação. No contexto desse trabalho, ele é especialmente influenciado pela estrutura do algoritmo e pelo tamanho e as características dos dados de entrada. A estrutura do código define qual a relação ou proporção que o tempo escala de acordo com a entrada fornecida.

A análise de complexidade de tempo é o campo que estuda como o tempo de execução ou o espaço de memória requerido por um algoritmo escala em função do aumento do tamanho da entrada, geralmente denotado por n (SEEDGEWICK; WAYNE, 2011, p. 176). Em vez de medir o tempo em segundos, a análise de complexidade utiliza notações assintóticas para caracterizar o crescimento do número de operações. A notação comum é a notação Big O, que descreve o limite superior ou o pior caso do crescimento do algoritmo (KLEINBERG; TARDOS, 2006, p. 35).

Por exemplo, travessia de uma lista requer a visita a cada um de seus elementos, necessitando um tempo diretamente ou *linearmente* proporcional, assim, é $O(n)$ (CORMEN *et al.*, 2009, p. 189). Se essa lista necessitar ser ordenada, pode-se fazê-lo por flutuações, onde ordena-se dois elementos consecutivos de cada vez, desse jeito, no pior dos casos, precisará-se passar n vezes para cada n elementos da lista, nota-se $O(n^2)$ para a complexidade quadrática (CORMEN *et al.*, 2009, p. 43).

Outra complexidade é a exponencial, que pode ser demonstrada pela sequência de Fibonacci. Até que se alcance o número n , gerará-se duas novas chamadas por iteração, duplicando o total de operações a cada incremento de n e obtendo-se $O(2^n)$ (CORMEN *et al.*, 2009, p. 60). Apresenta-se no Quadro 1 essas e mais classes

de complexidade, com exemplos representativos de cada uma.

Quadro 1 – Classes de complexidade de algoritmos e exemplos.

Notação	Nome	Exemplo
$O(1)$	Constante	Acessar um elemento de um vetor.
$O(\log n)$	Logarítmica	Busca binária em um vetor ordenado.
$O(n)$	Linear	Percorrer todos os elementos de uma lista.
$O(n \log n)$	Linearítmica	Ordenar uma lista por intercalação.
$O(n^2)$	Quadrática	Ordenar uma lista por flutuação.
$O(2^n)$	Exponencial	O cálculo de um número da sequência de Fibonacci.
$O(n!)$	Fatorial	Encontrar todas as permutações de uma lista.

Fonte: Adaptado de Cormen *et al.* (2009).

2.2 INTELIGÊNCIA ARTIFICIAL

A Inteligência Artificial (IA) pode ser definida como um campo da ciência da computação focado na criação de máquinas e sistemas que podem raciocinar e agir de maneiras que exigiriam inteligência humana (RUSSELL; NORVIG, 2010, p. 1). Em 1943, Warren McCulloch e Walter Pitts apresentaram um modelo matemático para os neurônios, considerado um marco fundamental na teoria de redes neurais artificiais (MCCULLOCH; PITTS, 1943).

Pouco depois, Turing (1950) propôs o Teste de Turing como um critério para avaliar a capacidade de uma máquina exibir comportamento inteligente indistinguível de um ser humano. Comportamentos como aprendizado, interpretação ou planejamento deveriam estar presentes para que as máquinas pensantes alcançassem a indistinguibilidade. O termo Inteligência Artificial, foi cunhado em 1956 por John McCarthy, desde então, as aplicações de IA se estenderam e ampliaram, sendo usadas desde corretores ortográficos até aplicações na robótica ou até mesmo na geração de arte (Stanford Institute for Human-Centered AI, 2024, p. 85, 135 e 96).

2.2.1 Aprendizado de máquina

Dentro do campo da IA, o Aprendizado de Máquina (ML) se destaca como uma abordagem fundamental para capacitar sistemas com inteligência. Em 1959, Arthur Samuel, um pioneiro da área, definiu-o como o "campo de estudo que dá aos computadores a habilidade de aprender sem serem explicitamente programados"(SAMUEL, 1959). Pode-se definir IA como:

“Um programa de computador que aprende com a experiência E em relação a alguma classe de tarefas T e medida de performance P, se seu

desempenho em tarefas em T, medido por P, melhora com a experiência E” (MITCHELL, 1997, p. 2).

O ML pode ser categorizado de acordo com o nível de supervisão fornecido durante o treinamento: de acordo com Janikow (1994), no aprendizado supervisionado, o modelo recebe dados de entrada rotulados com suas respectivas saídas e aprende a mapear novas entradas para suas saídas corretas; já de acordo com Chen *et al.* (2022), no aprendizado não supervisionado, o modelo recebe apenas dados de entrada e seu objetivo é encontrar estruturas ou correlações inerentes a eles; o aprendizado semi-supervisionado representa uma abordagem híbrida, utilizando uma mistura de dados rotulados e não rotulados (CHEN *et al.*, 2022).

2.2.2 Métodos de classificação e regressão

Dentro do aprendizado de máquina, diferentes técnicas podem ser utilizadas para o reconhecimento e ordenação de padrões, essas podem ser subdivididas em classificação e regressão. A distinção entre as tarefas de regressão e classificação reside na característica da variável-alvo a ser prevista. Em problemas de regressão, o objetivo é a predição do valor de uma ou mais variáveis-alvo contínuas (BISHOP, 2006, p. 137, 159). Em contrapartida, para problemas de classificação, o objetivo é atribuir um vetor de entrada a uma de um número finito de classes ou categorias discretas.

Bishop (2006, p. 46, 125, 137, 152, 225, 325 e 398), descreve alguns métodos relevantes para o contexto de tarefas de classificação e regressão:

- Regressão Linear e Logística: Modelos lineares que buscam encontrar um hiperplano que melhor se ajuste ou separe os dados;
- Máquina de Vetores de Suporte (SVM): Classificador que busca o hiperplano que maximiza a distância entre as diferentes classes de dados;
- Bayes ingênuo: Classificador probabilístico baseado no Teorema de Bayes, que assume a independência condicional entre as características dos dados;
- Árvores de Decisão: criam um modelo de regras hierárquicas em formato de fluxograma, onde cada nó representa um teste sobre uma característica e as arestas seus resultados;
- K-Vizinhos Mais Próximos (KNN): Método não paramétrico que classifica uma nova amostra com base na classe majoritária de seus 'k' vizinhos mais próximos no espaço de características;
- Redes Neurais Artificiais (RNA): Modelos inspirados na estrutura do cérebro humano, capazes de aprender relações não lineares complexas a partir dos dados;

- Floresta Aleatória: descrita por Leo Breiman (BREIMAN, 2001), é um método que constrói múltiplas árvores de decisão durante o treinamento, resultando na classe que mais votada pelas árvores individuais.

2.2.3 Redes neurais

Uma RNA é composta por múltiplas unidades de processamento interconectadas. Essas unidades são tipicamente organizadas em camadas: uma camada de entrada, que recebe os dados brutos, uma ou mais camadas ocultas, onde ocorre a maior parte do processamento e extração de características, e uma camada de saída, que apresenta o resultado final (BISHOP, 2006, p. 245).

As conexões entre os neurônios possuem pesos associados, que indicam a força ou importância daquela conexão e são ajustados durante o processo de aprendizado. O Aprendizado Profundo é um subcampo do aprendizado de máquina que utiliza redes neurais profundas, ou seja, redes neurais com múltiplas camadas de processamento para aprender representações de dados de forma hierárquica (GOODFELLOW *et al.*, 2016, p. 6).

2.2.4 Perceptron multicamadas

O Perceptron Multicamadas (MLP) é uma arquitetura de rede neural desenvolvida a partir do modelo de perceptron proposto por Rosenblatt (1958), composta por neurônios conectados com funções de ativação não lineares organizados em camadas. Ao contrário do perceptron original, o MLP é capaz de classificar dados que não são linearmente separáveis graças à presença de uma ou mais camadas ocultas entre a camada de entrada e a de saída (ROSENBLATT, 1958). Em cada neurônio, a soma ponderada das entradas é transformada por uma função de ativação não linear, como a unidade linear retificada (ReLU), definida como $\text{ReLU}(x) = \max(0, x)$ (GOODFELLOW *et al.*, 2016, p. 168). Essa não linearidade é o que permite ao MLP aprender funções arbitrariamente complexas (GOODFELLOW *et al.*, 2016, p. 172). Na camada de saída de problemas de classificação multiclasse, utiliza-se a função *softmax*:

$$\hat{p}_c = \frac{\exp(z_c)}{\sum_{k=1}^C \exp(z_k)} \quad (1)$$

O MLP é amplamente utilizado como classificador em tarefas de aprendizado supervisionado (BISHOP, 2006, p. 229). A camada de saída com função *softmax* produz uma distribuição de probabilidade sobre as classes, e a classe prevista é aquela com maior probabilidade atribuída. Neste trabalho, o MLP atua como classificador final da arquitetura GNN, recebendo como entrada a representação vetorial do grafo produzida após o agrupamento global e produzindo a distribuição de probabilidade sobre as oito classes de complexidade.

2.2.5 Redes neurais de grafo

Enquanto as redes neurais tradicionais operam em dados estruturados de forma tabular ou sequencial, GNNs são uma classe de redes neurais projetadas especificamente para processar dados representados na forma de grafos. Formalmente, um grafo G pode ser definido por um par (V, A) , onde V é o conjunto de nós e A é o conjunto de arestas. O grafo pode ser acompanhado por uma matriz de adjacência que denota as características de arestas ou de vértices (ZHOU *et al.*, 2020).

O princípio fundamental de uma GNN é o mecanismo de agregação de vizinhança ou passagem de mensagens, onde a representação vetorial de cada nó ou vértice é atualizada por i iterações com base nas representações de seus nós vizinhos. Isso permite que a GNN aprenda características dos atributos de um nó, de sua topologia e do contexto dos vizinhos desse nó no grafo (WU, 2021). As principais representações utilizadas na literatura são a AST, que captura a estrutura sintática; o CFG, que modela os caminhos de execução; e o PDG que modela as dependências de dados e controle (BIELIK *et al.*, 2016; FERRANTE *et al.*, 1987).

Além da definição dos componentes de um grafo, uma GNN possui uma função de passagem de mensagem, de agregação e de atualização. Existem diferentes funções dessas para diferentes arquiteturas de GNN, cada uma com mecanismos distintos. As principais GNN são descritas por Zhou *et al.* (2020): GCN, generaliza a convolução para dados de grafos, tipicamente agregando características de vizinhos por uma média normalizada; e GAT, que utiliza mecanismos de atenção para ponderar a importância das características dos nós vizinhos durante a agregação, permitindo que o modelo foque em partes relevantes da vizinhança.

2.2.5.1 Arquiteturas de Redes Neurais de Grafo

As principais arquiteturas de GNN diferenciam-se pelo mecanismo pelo qual cada nó agrega as representações dos seus vizinhos para atualizar a própria representação vetorial. Quatro variantes são especialmente relevantes para este trabalho: a Rede Neural Convolucional de Grafo (GCN), a Rede de Atenção de Grafo (GAT), a Rede de Isomorfismo de Grafo (GIN) e o GraphSAGE. Cada uma realiza diferentes compromissos entre expressividade, estabilidade e custo computacional, influenciando diretamente a capacidade do modelo de distinguir estruturas de grafo associadas a diferentes classes de complexidade. A seguir, cada arquitetura é descrita por meio de sua formulação matemática canônica, conforme proposta nos respectivos trabalhos originais, sem as modificações de implementação discutidas na Seção 3.5.

A GCN, proposta por Kipf e Welling (2017), generaliza a convolução espectral para grafos por meio de uma aproximação de primeira ordem do filtro de Chebyshev. A atualização da representação do nó v na camada k é dada por:

$$\mathbf{h}_v^{(k)} = \sigma \left(\sum_{u \in \mathcal{N}(v) \cup \{v\}} \frac{1}{\sqrt{\hat{d}_u \hat{d}_v}} \mathbf{W}^{(k)} \mathbf{h}_u^{(k-1)} \right) \quad (2)$$

onde σ é uma função de ativação não linear, $\hat{d}_v = 1 + \deg(v)$ é o grau do nó v com auto-conexão incluída, e $\mathbf{W}^{(k)} \in \mathbb{R}^{H \times H}$ é a matriz de pesos compartilhada da camada k . De acordo com Kipf e Welling (2017), a divisão pelo produto das raízes dos graus normaliza a contribuição de cada vizinho fazendo com que nós com muitos vizinhos recebam peso menor por vizinho. Essa normalização garante invariância ao tamanho da vizinhança e estabilidade ao custo de tornar a agregação insensível aos aspectos da vizinhança. A formulação padrão da GCN não incorpora características de arestas diretamente.

A GAT, proposta por Veličković *et al.* (2017), substitui a normalização por grau por coeficientes de atenção aprendidos. Para cada par de nós adjacentes (v, u) , um escalar de importância α_{vu} é calculado por uma rede de atenção parametrizada e normalizado via *softmax*:

$$\alpha_{vu}^{(k)} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^{(k)\top} [\mathbf{W}^{(k)} \mathbf{h}_v \parallel \mathbf{W}^{(k)} \mathbf{h}_u]))}{\sum_{w \in \mathcal{N}(v)} \exp(\text{LeakyReLU}(\mathbf{a}^{(k)\top} [\mathbf{W}^{(k)} \mathbf{h}_v \parallel \mathbf{W}^{(k)} \mathbf{h}_w]))} \quad (3)$$

onde $\mathbf{a}^{(k)} \in \mathbb{R}^{2H}$ é o vetor de atenção da camada k e $\parallel$ denota concatenação. A atualização de cada nó é então:

$$\mathbf{h}_v^{(k)} = \sigma \left(\sum_{u \in \mathcal{N}(v) \cup \{v\}} \alpha_{vu}^{(k)} \mathbf{W}^{(k)} \mathbf{h}_u^{(k-1)} \right) \quad (4)$$

O mecanismo de atenção permite que o modelo pondere diferentes vizinhos de forma adaptativa: nós com representações mais relevantes para o nó central recebem coeficientes maiores. Para capturar diferentes aspectos da vizinhança simultaneamente, a GAT opera com múltiplas fontes de atenção independentes, cujas saídas são concatenadas ou calculadas em média (VELIČKOVIĆ *et al.*, 2017). Essa flexibilidade é uma vantagem em relação à GCN, porém a GAT tende a ser instável em grafos com vizinhança muito pequena, onde o número reduzido de exemplos para o cálculo do *softmax* prejudica a estimativa confiável dos coeficientes.

A GIN, proposta por Xu *et al.* (2019), é fundamentada em uma análise do poder discriminativo das GNNs baseadas em passagem de mensagens. A função de atualização é:

$$\mathbf{h}_v^{(k)} = \text{MLP}^{(k)} \left((1 + \varepsilon^{(k)}) \cdot \mathbf{h}_v^{(k-1)} + \sum_{u \in \mathcal{N}(v)} \mathbf{h}_u^{(k-1)} \right) \quad (5)$$

onde $\varepsilon^{(k)}$ é um parâmetro escalar que controla o peso atribuído à representação do próprio nó em relação às mensagens dos vizinhos. O resultado central que distingue a GIN das demais arquiteturas é a escolha da soma como operador de agregação, pois enquanto a média e o máximo descartam informação sobre a vizinhança, impossibilitando distinguir, por exemplo, $\{A, A, B\}$ de $\{A, B\}$, a soma preserva essa informação. Para a tarefa de classificação de complexidade, isso é relevante porque dois grafos de código com o mesmo aninhamento de laços, mas com número diferente de estruturas de controle, devem ser distinguíveis pelo modelo.

O GraphSAGE, proposto por Hamilton *et al.* (2017), difere das arquiteturas anteriores pois ao invés de aprender representações estáticas para nós fixos, aprende uma função de agregação capaz de generalizar para nós não vistos durante o treinamento. A atualização concatena a representação do próprio nó com a agregação de um subconjunto amostrado de vizinhos $\mathcal{N}_s(v)$:

$$\mathbf{h}_v^{(k)} = \sigma(\mathbf{W}^{(k)} [\mathbf{h}_v^{(k-1)} \parallel \text{AGG}(\{\mathbf{h}_u^{(k-1)}, u \in \mathcal{N}_s(v)\})]) \quad (6)$$

onde AGG pode ser qualquer função de agregação diferenciável, como média, máximo ou LSTM (HAMILTON *et al.*, 2017). A amostragem de um número fixo de vizinhos por nó em cada camada permite escalar o método para grafos de grande porte e controlar o custo computacional por passo de treinamento.

2.2.5.2 Técnicas de aprendizado de redes

Proposto por Xu *et al.* (2018), as redes com Jumping Knowledge (JK) busca oferecer uma opção de comunicação que favorece nós centrais e periféricos, evitando sobresuavização dos nós centrais em representações muito profundas, mas ainda capturando dados de nós periféricos, problemas presentes em outras técnicas de agregação (YU *et al.*, 2023). O modelo agrega as representações de todas as camadas intermediárias para cada nó:

$$\mathbf{h}_v^{\text{final}} = \text{AGG}(\mathbf{h}_v^{(0)}, \mathbf{h}_v^{(1)}, \dots, \mathbf{h}_v^{(L)}) \quad (7)$$

onde AGG pode ser uma de três estratégias de agregação: concatenação, máximo ou memória longa de curto prazo (LSTM) (XU *et al.*, 2018; HOCHREITER; SCHMIDHUBER, 1997). A concatenação busca preservar todas as escalas de representação, produzindo um vetor que combina características de camadas rasas e profundas. Máximo seleciona para cada dimensão do vetor em representação o valor mais alto entre todas as camadas. Atenção por LSTM toma as representações de cada camada como sequência e computa coeficientes de atenção (HOCHREITER; SCHMIDHUBER, 1997).

Para tarefas de classificação de grafos, as representações de nós obtidas após as camadas de mensagens precisam ser combinadas em uma única representação global, denominada *READOUT* (BUTEREZ *et al.*, 2022). A formulação geral é:

$$\mathbf{h}_G = \text{READOUT}(\{\mathbf{h}_v^{(L)} \mid v \in V\}) \quad (8)$$

onde a função *READOUT* é invariante à permutação dos nós do grafo.

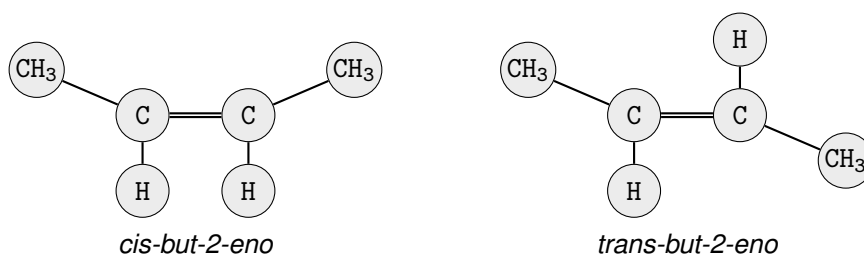
2.3 TOPOLOGIAS

A aplicação de GNNs a representações estruturais como derivação direta da topologia de domínio foi demonstrada em outros contextos: Bloch *et al.* (2023) emprega-as sobre grafos de plantas arquitetônicas. Outro tipo de topologia que pode-se explorar é o da química molecular. Esse tipo de representação estuda como a conectividade e o arranjo espacial dos átomos em uma molécula induz propriedades físicas e químicas (TRINAJSTIĆ, 1992).

Uma consequência fundamental dessas relações é que moléculas compostas pelos mesmos tipos e quantidades de átomos podem apresentar propriedades completamente distintas quando esses átomos estão ligados de formas ou orientações diferentes (WADE, 2012). Na química, essas moléculas são denominadas isômeros *cis-trans* (International Union of Pure and Applied Chemistry, 2025). Um exemplo disso é o par *cis-trans* but-2-eno, que apresenta mesma quantidade de átomos e mesmas ligações, mas difere na disposição dos grupos em torno da ligação dupla central. Essa diferença ocasiona em diferenças em propriedades como pontos de ebulição, momentos dipolares e reatividades distintos entre os dois composto (WADE, 2012).

A Figura 1 ilustra os dois isômeros.

Figura 1 – Isômeros *cis* e *trans* do but-2-eno.



Fonte: Adaptado de International Union of Pure and Applied Chemistry (2025).

Esse princípio serve como base conceitual para a representação de código-fonte adotada neste trabalho. Um arquivo de código-fonte pode ser interpretado como uma molécula, sendo composto por unidades elementares denominadas átomos, que

correspondem a declarações, estruturas de controle e chamadas de função, conectados por diferentes tipos de relações que representam a hierarquia sintática, o fluxo de controle e as dependências de dados. Da mesma forma que isômeros compartilham a mesma composição atômica mas diferem em propriedades em razão do arranjo, dois algoritmos podem utilizar os mesmos tipos de construções e pertencer a classes de complexidade distintas (CORMEN *et al.*, 2009, p. 159).

2.4 BANCO DE DADOS DE CÓDIGOS E COMPLEXIDADE

O sucesso de abordagens baseadas em aprendizado de máquina depende da disponibilidade de conjuntos de dados de para treinamento e avaliação. Para a tarefa de previsão de complexidade de código, destacam-se três principais datasets públicos:

- *Code Runtime Complexity Dataset (CoRCoD)*: Proposto por Sikka *et al.* (2020), este dataset foi um dos primeiros a ser criado especificamente para a tarefa. Contém 932 códigos-fonte em Java, extraídos da plataforma de programação competitiva Codeforces. As amostras são rotuladas manualmente por especialistas em cinco classes de complexidade de tempo: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$ e $O(n^2)$.
- *CodeComplex*: Introduzido por Baik *et al.* (2024) e utilizado por Park *et al.* (2025), é atualmente o maior dataset público para esta finalidade. Ele consiste em aproximadamente 4.600 soluções em Python e 4.800 em Java, também coletadas do Codeforces. Este dataset adiciona às classes de complexidade as categorias $O(n^3)$, e $O(2^n)$.
- *Crawled Dataset/GeeksForGeeks*: apresentado por (RODENBUSCH, 2023), este trabalho desenvolveu um dataset complementar ao CoRCoD, extraindo 394 códigos em Java através da internet. O principal objetivo foi preencher a lacuna de classes de complexidade ineficientes, como exponencial e polinomial, ainda adicionando outra classe, a sublinear $O(\log n)$.

2.5 MÉTRICAS DE AVALIAÇÃO

Os métodos de aprendizado de máquina possuem performance diferente para cada aplicação, necessitando ser investigados e avaliados em cada caso. A avaliação de desempenho do modelo utiliza três métricas, derivadas da matriz de confusão. Esta matriz compara os valores preditos com os valores reais, organizando-os em quatro contagens: Verdadeiros Positivos (TP), Verdadeiros Negativos (TN), Falsos Positivos (FP) e Falsos Negativos (FN) (TAN *et al.*, 2005, p. 149). A partir desses componentes, Tan *et al.* (2005, p. 41 e 548) descrevem as métricas utilizadas no modelo:

- Acurácia: Representa a fração de predições corretas entre o número total de amostras. A fórmula é:

$$\text{Acurácia} = \frac{TP + TN}{TP + TN + FP + FN}$$

- Acurácia Adjacente: Estratégia adotada que calcula a acurácia, mas também contabiliza estimativas adjacentes a classe verdadeira na escala ordinal de classes.
- Precisão: Para uma classe, calcula a proporção de predições positivas que são corretas. A fórmula é:

$$\text{Precisão} = \frac{TP}{TP + FP}$$

- Revocação: Para uma classe, mede a proporção de amostras positivas reais que foram identificadas corretamente pelo modelo. A fórmula é:

$$\text{Revocação} = \frac{TP}{TP + FN}$$

- F1-Score: Corresponde à média harmônica entre precisão e revocação, combinando as duas métricas em um único valor. Esta métrica é utilizada em situações com desbalanceamento de classes. A fórmula é:

$$F_1 = 2 \times \frac{\text{Precisao} \times \text{Revocação}}{\text{Precisao} + \text{Revocação}}$$

Além das métricas derivadas da matriz de confusão, o problema de classificação de complexidade apresenta estrutura ordinal de modo que erros de maior distância na escala são mais graves do que erros adjacentes. Essa propriedade motiva o uso de métricas adicionais descritas a seguir:

- Erro Absoluto Ordinal Médio (MAOE). O MAOE é uma adaptação do Erro Absoluto Médio (MAE) (TAN *et al.*, 2005, p. 149) ao contexto de classificação ordinal. Dado que as classes são indexadas por inteiros de 0 a $C - 1$, o MAOE é definido como a diferença absoluta média entre os índices da classe prevista e da classe verdadeira sobre o conjunto de avaliação:

$$\text{MAOE} = \frac{1}{N} \sum_{i=1}^N |\hat{y}_i - y_i| \quad (9)$$

onde $\hat{y}_i$ é o índice da classe prevista e y_i é o índice da classe verdadeira para o exemplo i . A versão normalizada divide pelo número máximo de posições possíveis ($C - 1$), produzindo um valor em $[0, 1]$ independente do número de classes:

$$\text{MAOE}_{\text{norm}} = \frac{\text{MAOE}}{C - 1} \quad (10)$$

Em conjunto com a acurácia adjacente, obtém-se uma percepção do quão longe o modelo está errando nas estimativas.

- Concordância de Cohen (κ). O coeficiente kappa de Cohen mede a concordância entre dois sistemas de classificação descontando os acertos esperados por acaso (LANDIS; KOCH, 1977). Para um classificador e um conjunto de rótulos verdadeiros, ele é definido como:

$$\kappa = \frac{p_o - p_e}{1 - p_e} \quad (11)$$

onde p_o é a proporção de acertos observados (acurácia) e p_e é a proporção de acertos esperados por acaso, calculada como a soma dos produtos das proporções de cada classe na distribuição prevista e na distribuição verdadeira. O coeficiente varia entre -1 e 1 . Valores negativos indicam concordância abaixo do acaso, zero indica concordância aleatória e 1 indica concordância perfeita. Landis e Koch (1977) propõem uma escala qualitativa amplamente adotada: valores abaixo de 0 indicam concordância pobre; entre 0 e $0,20$, leve; entre $0,21$ e $0,40$, razoável; entre $0,41$ e $0,60$, moderada; entre $0,61$ e $0,80$, substancial; e acima de $0,80$, quase perfeita.

- Área sob a curva ROC (AUC-ROC). A curva ROC (*Receiver Operating Characteristic*) é construída variando o limiar de decisão de um classificador binário e plotando a taxa de verdadeiros positivos (TPR) em função da taxa de falsos positivos (FPR) para cada limiar (FAWCETT, 2006). A área sob essa curva (AUC) fornece uma medida escalar do poder discriminativo do classificador. Um valor de $0,5$ corresponde a um classificador aleatório e um valor de $1,0$ corresponde a discriminação perfeita (FAWCETT, 2006). Neste trabalho, a AUC-ROC é calculada para a classificação binária de eficientes, agrupando as classes em eficientes ($O(1)$ a $O(n \log n)$) e ineficientes ($O(n^2)$ a $O(n!)$) — utilizando as probabilidades previstas pelo modelo após a função *softmax*.

2.6 FERRAMENTAS AUXILIARES

A implementação do modelo proposto nesse trabalho apoia-se um conjunto de bibliotecas e estruturas que auxiliam no processamento de código-fonte, na construção de grafos e no treinamento de redes neurais.

2.6.1 Processamento de código-fonte

A análise sintática de código-fonte requer leitores específicos para cada linguagem de programação. Para Python, utiliza-se o módulo *ast* da biblioteca padrão, que transforma o código-fonte em uma Árvore de Sintaxe Abstrata (Python Software Foundation, 2025). Para Java, utiliza-se a biblioteca *javalang*, que implementa um leitor descendente recursivo para a gramática Java (THIEMEL *et al.*, 2025). Para as

linguagens C, C++, JavaScript e C# utiliza-se o Tree-sitter, um módulo de análise sintática baseado em gramáticas definidas externamente que produz árvores de sintaxe concretas (BRUNSFELD *et al.*, 2025).

2.6.2 Representação e manipulação de grafos

A biblioteca NetworkX é utilizada para a construção e manipulação dos grafos de dependência antes da conversão para tensores (HAGBERG *et al.*, 2008). Em particular, é empregada na detecção de comunidades por meio do algoritmo de Louvain, que maximiza a modularidade do grafo (BLONDEL *et al.*, 2008), e no cálculo de propriedades topológicas auxiliares dos grafos de código.

2.6.3 Aprendizado profundo em grafos

O framework PyTorch fornece a base de álgebra linear e diferenciação sobre as quais o modelo é construído (PASZKE *et al.*, 2019). A extensão PyTorch Geometric (PyG) implementa as operações de passagem de mensagens, os operadores de agrupamento global e os carregadores de dados para grafos utilizados neste trabalho (FEY; LENSSEN, 2019). O PyG representa cada grafo como um objeto *torch_geometric.data.Data* com tensores para as características de nó (*x*), os pares de arestas (*edge_index*) e as características de aresta (*edge_attr*), além de atributos adicionais definidos pelo usuário (FEY; LENSSEN, 2019).

2.6.4 Otimização

O otimizador AdamW (LOSHCHILOV; HUTTER, 2019) é responsável pela atualização dos pesos do modelo durante o treinamento. O AdamW é uma variante do Adam que separa a regularização L2 do cálculo do gradiente adaptativo. O agendamento da taxa de aprendizado é realizado por um *ReduceLROnPlateau*, que reduz a taxa quando a acurácia de validação deixa de melhorar por um número configurável de épocas consecutivas (PyTorch Contributors, 2025).

2.7 CONCEITOS DA METODOLOGIA

O treinamento de uma rede neural é o processo iterativo de ajuste dos parâmetros do modelo, pesos e vieses de suas camadas, de modo a minimizar uma medida de erro sobre um conjunto de exemplos rotulados (GOODFELLOW *et al.*, 2016, p. 164). Esta seção descreve os conceitos fundamentais que constituem esse processo: a divisão dos dados, o cálculo da perda e a regularização.

2.7.1 Divisão dos dados

O conjunto de dados disponível é dividido em três partições baseado em (GOODFELLOW *et al.*, 2016, p. 118). O conjunto de treino é utilizado para atualizar os parâmetros do modelo a cada iteração. O conjunto de validação é usado ao longo do treinamento para monitorar o desempenho em dados não utilizados, como o ajuste da taxa de aprendizado e a parada antecipada. O conjunto de teste é usado para a avaliação final após o término do treinamento, para produzir uma estimativa não enviesada do desempenho do modelo em dados reais (GOODFELLOW *et al.*, 2016, p. 118).

2.7.2 Função de perda

A função de perda quantifica a diferença entre a distribuição de probabilidade prevista pelo modelo e o rótulo verdadeiro de cada exemplo (GOODFELLOW *et al.*, 2016, p. 178). Para tarefas de classificação multiclasse, a função de perda padrão é a entropia cruzada entre a distribuição alvo e a distribuição prevista pelo modelo após a aplicação da função *softmax* (GOODFELLOW *et al.*, 2016, p. 180). Para um único exemplo com rótulo verdadeiro y e distribuição prevista $\hat{p}$, a entropia cruzada é:

$$\mathcal{L}_{CE} = -\log \hat{p}_y \quad (12)$$

Onde busca-se maximizar o log da probabilidade atribuída à classe correta.

A média da perda sobre todos os exemplos de uma amostra de tamanho B compõe o sinal utilizado em cada passo de otimização:

$$\mathcal{L} = \frac{1}{B} \sum_{i=1}^B \mathcal{L}_{CE}^{(i)} \quad (13)$$

2.7.3 Regularização

Regularização é o conjunto de técnicas que reduzem o erro de generalização de um modelo sem reduzir o erro de treinamento (GOODFELLOW *et al.*, 2016, p. 120). As duas formas de regularização mais comuns em redes neurais profundas são o decaimento de pesos e o *dropout*. O decaimento de pesos, também denominado regularização L2, adiciona à função de perda um termo proporcional à norma ao quadrado dos parâmetros, penalizando pesos de grande magnitude (GOODFELLOW *et al.*, 2016, p. 229):

$$\mathcal{L}_{reg} = \mathcal{L} + \frac{\lambda}{2} \|\theta\|_2^2 \quad (14)$$

onde λ é o coeficiente de regularização. No AdamW, esse termo é injetado diretamente sobre os parâmetros a cada passo, sem interferir nas estimativas adaptativas de momento (LOSHCHILOV; HUTTER, 2019).

O *dropout* é uma técnica que, durante o treinamento, zera aleatoriamente uma fração p das ativações de cada camada a cada passo de otimização (GOODFELLOW *et al.*, 2016, p. 251). Isso impede que neurônios individuais se especializem excessivamente em padrões específicos do conjunto de treino.

2.8 TRABALHOS RELACIONADOS

A busca pela automação da análise de complexidade não é recente (ROSENDAHL, 1989), contudo, com os avanços em IA, novas abordagens surgiram. A literatura recente pode ser categorizada em áreas principais: abordagens de aprendizado de máquina para previsão de complexidade, aplicações de GNNs em engenharia de software e os fundamentos teóricos que guiam o desenvolvimento de arquiteturas GNN.

Na parte de previsão de complexidade, trabalhos como os de Pfitscher *et al.* (2023) e Sikka *et al.* (2020) estabeleceram a viabilidade do uso de aprendizado de máquina tradicional. Ambos extraíram características de ASTs e obtiveram resultados significativos com o modelo Floresta Aleatória. Sikka *et al.* (2020) também introduziram o uso de embeddings de grafos com graph2vec, uma aplicação de um método similar a GNN para esta tarefa. Recentemente, a pesquisa avançou para o uso de grandes modelos de linguagem pré-treinados, onde Park *et al.* (2025) propuseram a estratégia de aprendizado contrastivo @CAL para melhorar o desempenho desses modelos na previsão de complexidade de tempo.

Explorando a previsão de tempo de execução como um problema de regressão, Gao *et al.* (2023) apresenta o DNNPerf, uma ferramenta que utiliza GNNs sobre grafos de computação para prever o desempenho de modelos de Deep Learning. Na aplicação de engenharia de software, Hin *et al.* (2022), no trabalho sobre o LineVD, aplicaram GNNs sobre Grafos de Dependência do Programa (PDGs) para a tarefa de detecção de vulnerabilidades em nível de instrução. Sikic *et al.* (2022) utilizaram uma GNN de base espectral sobre ASTs para a predição de defeitos em código.

Outros trabalhos exploraram desafios práticos, como a garantia de privacidade em nível de nó ao treinar GNNs com dados sensíveis (DAIGAVANE *et al.*, 2021) e a análise crítica das dificuldades e do retorno decrescente de múltiplas camadas em GNNs aplicadas à classificação de código em geral (YU *et al.*, 2023). Fora do contexto de código, como em design de arquitetura, GNNs foram empregadas com sucesso para análise de conformidade de projetos (BLOCH *et al.*, 2023). Apresenta-se no Quadro 2 um resumo desses trabalhos.

Quadro 2 – Trabalhos relacionados e suas contribuições.

Artigo	Modelo / Método Utilizado	Objetivo Resumido
Pfitscher <i>et al.</i> (2023)	Floresta aleatória, Extração de Features de AST	Estimar a complexidade de tempo de execução de código Java.
Rodenbusch (2023)	API de AutoGluon Tabular	Uso de múltiplos métodos de ML para estimativa de complexidade aplicados à uma aplicação virtual na rede.
Sikka <i>et al.</i> (2020)	Floresta aleatória, graph2vec	Classificar a complexidade de tempo e introduzir embeddings de grafo.
Park <i>et al.</i> (2025)	Modelos grandes de linguagem (LLMs), @CAL	Aprimorar a predição de complexidade com aprendizado contrastivo.
Gao <i>et al.</i> (2023)	Redes neurais de grafo (GNN)	Prever o desempenho (tempo e memória) de modelos de Aprendizado Profundo.
Hin <i>et al.</i> (2022)	Redes neurais de grafo (GNN) sobre PDG	Detectar vulnerabilidades em nível de instrução no código-fonte.
Sikic <i>et al.</i> (2022)	Redes neurais convolucionais de grafo (GCNN)	Realizar a predição de defeitos em código utilizando ASTs.
Yu <i>et al.</i> (2023)	Redes neurais de grafo (GNNs)	Analisar criticamente as dificuldades e limites de GNNs para classificação de código.

Fonte: Elaborado pelo autor (2026).

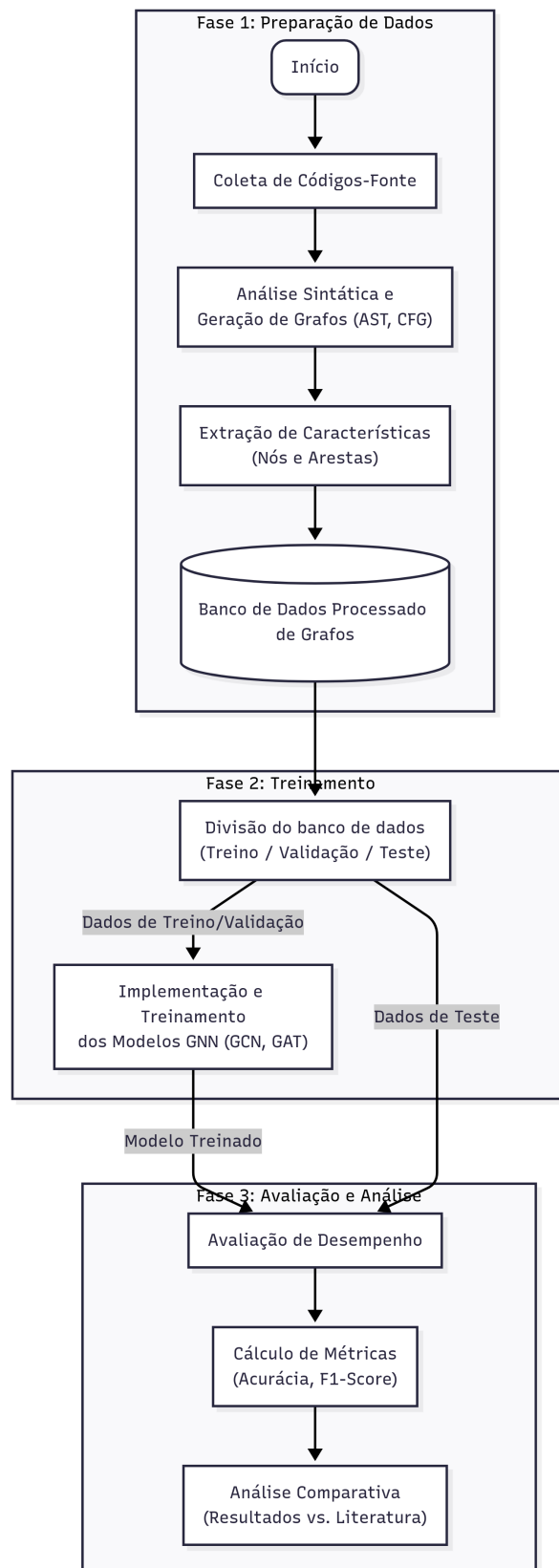
3 METODOLOGIA

A metodologia se inicia com a coleta de códigos-fonte de bases de dados públicas, como *CodeComplex* e *Crawled*. Os códigos serão submetidos a um processo de análise sintática para a conversão em representações de grafo, como AST e CFG. Desses grafos, é extraída características de nós e arestas, que quantificam elementos estruturais e semânticos do código, como tipos de declarações, laços de repetição e chamadas de função.

A etapa subsequente consiste na implementação de arquiteturas de GNN, com foco em modelos como GIN e GCN. As redes são projetadas para operar sobre os grafos gerados, utilizando o mecanismo de passagem de mensagens para agregar informações da vizinhança de cada nó. O objetivo é a aprendizagem de uma representação vetorial do grafo inteiro que capture as propriedades computacionais relevantes para a classificação de complexidade.

Para o treinamento e avaliação, o conjunto de dados é segmentado em partições de treino, validação e teste. A otimização dos modelos utiliza uma função de perda adequada para classificação multiclasse. O desempenho das arquiteturas de GNN será aferido por meio de métricas como acurácia e F1-score. Por fim, os resultados são comparados com os de modelos de classificação de complexidade documentados na literatura, a fim de validar a eficácia da abordagem baseada em GNN. A Figura 2 demonstra o fluxo entre as etapas. A implementação completa está disponível publicamente em repositório (NOGARA, 2026).

Figura 2 – Fluxograma da metodologia.



Fonte: Elaborado pelo autor (2026).

3.1 PREPARAÇÃO DA BASE DE DADOS

A base de dados utilizada nesse trabalho é constituída por dois conjuntos públicos de códigos-fonte possuintes de rótulos de classes de complexidade de tempo: o conjunto *GeeksForGeeks*, disponibilizado por Rodenbusch (2023), e o conjunto *CodeComplex*, disponibilizado por Baik *et al.* (2024). Ambos conjuntos possuem problemas algorítmicos em uma das 8 linguagens de programação aceitas, com todos esses sendo acompanhados de metadados que incluem a complexidade de tempo. A base de dados CoRCoD não foi incluída por apresentar poucos exemplos e poucos tipos de classes, além de não ser estudada isoladamente nos outros trabalhos, não contribuindo para comparações ou conclusões.

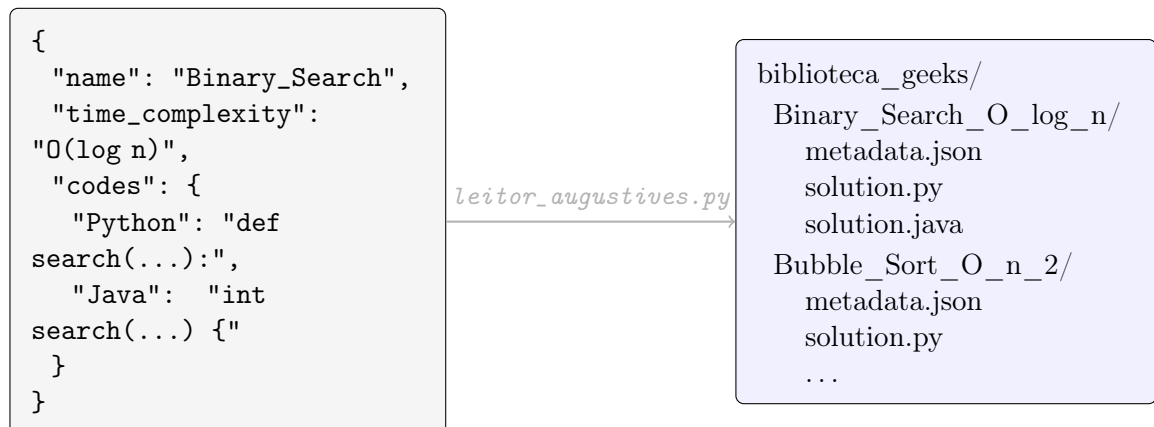
Quadro 3 – Módulos de conversão dos conjuntos de dados.

Módulo	Conjunto	Descrição
<i>leitor_augustives.py</i>	<i>GeeksForGeeks</i>	Lê o arquivo JSON de array e cria uma pasta por problema, com os arquivos de código-fonte em suas extensões nativas e um arquivo <i>metadata.json</i> contendo os metadados do problema.
<i>leitor_code_complex.py</i>	<i>CodeComplex</i>	Lê os arquivos JSONL linha a linha e gera a mesma estrutura de diretórios, nomeando cada pasta com o identificador de fonte, o identificador de problema e o índice de linha, no formato <i>FONTE_PROBLEMA_sol_N</i> .

Fonte: Elaborado pelo autor (2026).

Os arquivos iniciais do conjunto *GeeksForGeeks* estavam no formato JSON de *array* e os do conjunto *CodeComplex* no formato JSONL, com um registro por linha. Nessa etapa, desenvolveu-se dois arquivos, um para cada conjunto, para separar os códigos em pastas estruturadas compatíveis com os leitores da etapa subsequente. Para cada problema algorítmico, os seus respectivos módulos criam uma pasta com o nome do problema extraído dos metadados originais da base de dados. Dentro dessa pasta, são gravados os arquivos de código-fonte em suas respectivas extensões (*.py*, *.java*, *.cpp*, *.c*, *.js*, *.cs*) e um arquivo *metadata.json* com os metadados relevantes. Caso no código original houvesse duas soluções em uma mesma linguagem, mas em versões diferentes, havia adição de um sufixo aos outros registros, como *Python* e *Python3*, presentes no conjunto *GeeksForGeeks*. A Figura 3 ilustra o processo de leitura.

Figura 3 – Conversão dos conjuntos de dados para estrutura de diretórios.



Fonte: Elaborado pelo autor (2026).

Estabelecida a estrutura padrão, cada código-fonte então passa por uma etapa de limpeza: caracteres de espaço não-separáveis (*U+00A0*), frequentes em dados provenientes de *web scraping*, são substituídos por espaços ASCII. Na sequência, comentários de linha e de bloco são removidos por expressões regulares específicas de cada linguagem. Nenhuma dessas modificações altera a estrutura sintática ou semântica dos programas. Ao fim desse processo, a base estruturada conjunta contém 9635 pastas de problemas, cada uma contendo ao menos um problema.

3.2 ROTULAÇÃO DE COMPLEXIDADE

Os rótulos inicialmente apresentam-se em cadeias de caracteres livres e sem padronização entre os dois conjuntos. O campo *time_complexity* do conjunto *Geeks-ForGeeks* contém notações como $O(n \cdot \log(n))$, $O(N)$ ou *linear*. Já o campo *complexity* do conjunto *CodeComplex* utiliza convenções distintas, incluindo identificadores abreviados ou escrito por extenso. Em ambos conjuntos é comum também classes de complexidade fora do escopo, como $O(N \cdot M)$ ou $O(N \cdot \log(K))$. Com essas questões em mente, elaborou-se um processo de normalização para mapear esses distintos rótulos em um conjunto de 8 classes ordinais, enumeradas na Tabela 1.

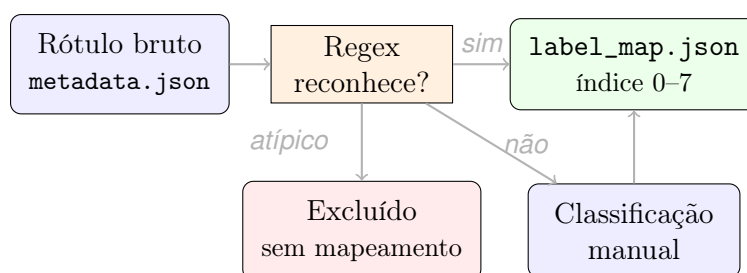
Tabela 1 – Classes de complexidade de tempo adotadas e exemplos de rótulos normalizados.

Índice	Classe	Exemplos de rótulo aceitos
0	$O(1)$	$O(1)$, <i>constant</i>
1	$O(\log n)$	$O(\log n)$, $O(\log N)$, <i>logn</i>
2	$O(n)$	$O(n)$, $O(N)$, <i>linear</i>
3	$O(n \log n)$	$O(n \log n)$, $O(N \log N)$, <i>nlogn</i>
4	$O(n^2)$	$O(n^2)$, $O(n * n)$, <i>quadratic</i>
5	$O(n^3)$	$O(n^3)$, $O(n^3)$, <i>cubic</i>
6	$O(2^n)$	$O(2^n)$, <i>exponential</i>
7	$O(n!)$	$O(n!)$, <i>factorial</i>

Fonte: Elaborado pelo autor (2026).

A normalização opera em duas etapas: na primeira, expressões regulares identificam padrões canônicos e os convertem ao índice inteiro correspondente. Aqui são tratadas variações ortográficas, capitalizações e notações irregulares. Rótulos não reconhecidos automaticamente, como os expressos em múltiplas variáveis, são registrados para classificação manual na etapa seguinte: o mapeamento é então escrito em um arquivo *label_map.json* que funciona como registro persistente. Nesse registro, classificações confirmada manualmente se mantêm em mapeamentos subsequentes, garantindo reprodutibilidade. Rótulos que permaneceram sem mapeamento válido após as duas etapas foram excluídos do conjunto de treinamento, excluindo assim 1806 exemplos da base de dados. Os rótulos validados serão então injetados nos arquivos *mol.json* gerados na etapa de extração de representações estruturais. A Figura 4 sintetiza o fluxo de normalização.

Figura 4 – Fluxo de normalização de rótulos de complexidade.



Fonte: Elaborado pelo autor (2026).

3.3 ANÁLISE SINTÁTICA E LEITURA TOPOLÓGICA

A leitura topológica constitui a etapa central desse trabalho: com inspiração em topologias reais, em especial na área de química, busca-se analisar os códigos como se fossem moléculas. Para alcançar esse objetivo, cada arquivo da base de

dados é processado por uma sequência de dez passes de análise, coordenados pelo código *leitor_molecular.py*, que executa os leitores em ordem topológica de dependência e acumula os resultados em uma estrutura de grafo denominada molécula. Uma molécula é um trecho de código que pode ser repetido, podendo ser apenas uma declaração ou operação, referenciado como átomo, ou uma função inteira. Ao término do processamento, cada molécula é serializada no formato JSON (*.mol.json*) e registrada na biblioteca de topologias, descrita adiante na Seção 3.3.7. O conjunto completo de 12 376 arquivos de código-fonte gerou um arquivos *.mol.json* por código, contendo 53 199 grafos de funções individuais.

A implementação divide-se em quatorze códigos, divididos majoritariamente em duas classes: leitores, responsáveis por extrair informações do código-fonte ou do grafo em construção, e bibliotecas, responsáveis por catalogar e registrar os dados extraídos pelos leitores. O Quadro 4 descreve cada código e seu funcionamento.

Quadro 4 – Módulos do processo de análise topológica

Módulo	Tipo	Responsabilidade
<code>leitor_base.py</code>	Leitor base	Abstrai os leitores nativos de cada linguagem em uma representação de nó universal.
<code>leitor_atomico.py</code>	Leitor	Extraí os nós atômicos do grafo a partir da árvore sintática abstrata e atribui categorias semânticas a cada nó.
<code>leitor_topologico.py</code>	Leitor	Adiciona arestas de contenção sintática e de sequência entre nós adjacentes.
<code>leitor_fluxo.py</code>	Leitor	Constrói o GFC, inserindo arestas de fluxo, ramificação e laço.
<code>leitor_pdg.py</code>	Leitor	Deriva arestas de dependência de dados entre definições e usos de variáveis; calcula a profundidade de aninhamento de laços por nó.
<code>leitor_dominancia.py</code>	Leitor (opt.)	Adiciona arestas de dominância e pós-dominância entre nós de controle.
<code>leitor_recurcao.py</code>	Leitor (opt.)	Detecta chamadas recursivas na lista de átomos da molécula.
<code>leitor_funcional.py</code>	Leitor	Delimita as funções e segmenta o grafo em subgrafos independentes por função.
<code>leitor_molecular.py</code>	Leitor	Orquestra a execução de todos os passes em ordem topológica e serializa o resultado final em formato <code>.mol.json</code> .
<code>biblioteca_atomica.py</code>	Biblioteca	Acumula, por tipo de átomo e linguagem, a distribuição empírica de classes de complexidade observadas no treinamento.
<code>biblioteca_topologica.py</code>	Biblioteca	Acumula padrões de conexão estrutural entre tipos de nó por linguagem.
<code>biblioteca_molecular.py</code>	Biblioteca	Armazena e gerencia a biblioteca de topologias moleculares e suas sub-assinaturas para o contexto de sub-molécula.

Fonte: Elaborado pelo autor (2026).

3.3.1 Extração de átomos

O primeiro leitor de análise converte o código fonte em um conjunto de nós elementares denominados átomos. Similares ao conceito químico, são definidos como a menor unidade de construção com relevância semântica para a análise de com-

plexidade, podendo ser declarações isoladas, chamadas de funções, estruturas de repetição, entre outras.

O código *leitor_base.py* opera como gerenciador, recebendo o código-fonte e a linguagem identificada pela extensão do arquivo e chama o leitor nativo correspondente. Quando o código-fonte está em *Python*, é selecionado o módulo Abstract Syntax Tree (AST), quando em Java é selecionado o módulo *Javalang* e para as linguagens restantes é chamado o módulo Tree-Sitter. Todos esses leitores ao serem chamados sobre o arquivo retornam uma estrutura de árvore específica de cada biblioteca.

Para produzir uma representação uniforme independente de linguagem, o módulo percorre recursivamente essa árvore e constrói um objeto *ParseNode* para cada nó relevante. Cada *ParseNode* expõe três campos: o tipo sintático do nó (*node_type*), a lista de nós filhos (*children*) e os atributos adicionais reportados pelo parser original. O percurso é realizado em profundidade, de modo que os filhos de um nó são processados antes de seus irmãos seguintes.

O módulo *leitor_base.py* também é responsável por normalizar os tipos dos átomos extraídos. Por exemplo, um laço *for* é representado pela cadeia "*for_statement*" em C e C++, porém como "*enhanced_for_statement*" em Java. A normalização é realizada via uma tabela de mapeamento explícita, na qual cada tipo de nó específico de linguagem é associado a um nome compartilhado. Nos exemplos prévios, todas as variantes de laço *for* são convertidas para um único tipo "*For*".

O registro oficial de todos os tipos de átomo e de aresta encontrados durante o processamento é mantido pelo módulo *edge_registry.py* e persistido no arquivo *edge_registry.json*. Esse módulo implementa uma estrutura acumulativa: ao encontrar um tipo ainda não registrado durante o processamento de um arquivo, atribui-se a ele um identificador numérico inteiro sequencial e o novo par (nome, identificador) é adicionado ao registro.

Ao final de cada execução, o estado do registro é serializado. Em execuções subsequentes, o arquivo é carregado de volta e os identificadores existentes são preservados, garantindo que o mesmo tipo de átomo ou aresta receba sempre o mesmo identificador numérico independente da ordem de processamento dos arquivos. Esse identificador numérico também é utilizado como índice no vetor de características de arestas fornecidos à rede neural.

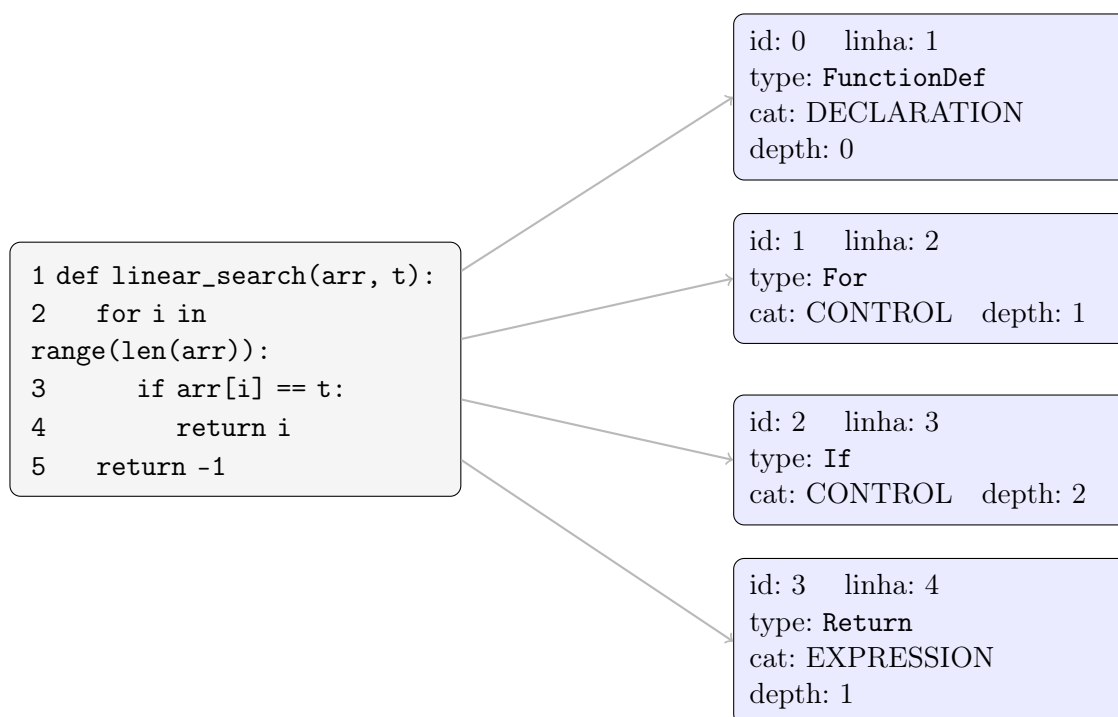
Após a conversão para *ParseNode*, o código *leitor_atómico.py* percorre a árvore gerada e extrai os nós com relevância estrutural para a análise de complexidade, descartando elementos sintáticos auxiliares como operadores, delimitadores, nós de pontuação e outros elementos sintáticos sem conteúdo semântico próprio. Para cada nó selecionado, o módulo instancia um objeto átomo que recebe os seguintes atributos:

- *id*: identificador numérico único dentro da molécula;
- *type*: tipo sintático original, podendo ser declarações, chamadas de funções ou qualquer estrutura elementar. Exemplos são: *For*, *While*, *If*, *FunctionDef*, *Call*, *Assign* e *Return*;
- *category*: categoria semântica que agrupa tipos em quatorze classes, podendo ser itens como *CONTROL* (estruturas de repetição e decisão) ou *COMPARATOR* (maior, menor ou igual).
- *lineno*: número de linha no código-fonte, utilizado como *feature* relativa após normalização.

Um quinto atributo, *loop_depth* é calculado pelo passe de dependência de dados descrito na Seção 3.3.4 e armazenado no mesmo objeto. Esse atributo representa o número de estruturas de laço aninhadas que contém o átomo no momento de sua execução.

A Figura 5 ilustra a extração de átomos sobre uma função de busca linear em Python. O átomo *For* recebe categoria *CONTROL* e o átomo *If* recebe a mesma categoria mas com *loop_depth* igual a 1, indicando um nível de aninhamento.

Figura 5 – Extração de átomos a partir de uma função de busca linear em Python.



Fonte: Elaborado pelo autor (2026).

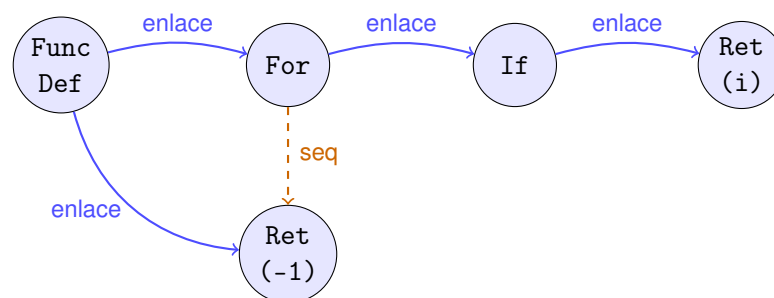
O tipo *Call* não aparece na figura por ter sido descartado neste exemplo simplificado; em funções reais, chamadas como *range()* e *len()* geram átomos *Call* com categoria *CALL*. O *Return(-1)* da linha 5 é mapeado ao mesmo átomo do *Return(i)* por

compartilharem o mesmo tipo sintático; a distinção do valor retornado ocorre no nível do GFC (Seção 3.3.3), não no nível atômico.

3.3.2 Grafo de Dependência Estrutural

Após a extração dos átomos, é necessário estabelecer a conexão entre eles. O módulo *leitor_topologico.py* é responsável por isso, adicionando ao grafo os tipos de arestas que reconstroem a hierarquia sintática da AST. Existem dois tipos de arestas: as arestas de enlace e as arestas de sequência. O primeiro tipo é inserido de um nó pai para cada um de seus nó filhos na AST, já o segundo é inserido entre nós consecutivos na ordem de declaração, representando a ordem de execução entre instruções no mesmo bloco. No exemplo da Figura 5, *def* possui uma aresta de enlace com *for* enquanto *arr[i]* possui aresta de sequência com *assign (=)*. A Figura 6 ilustra essas e as outras arestas resultantes.

Figura 6 – Arestas estruturais de enlace e sequência sobre os átomos da função de busca linear.



Fonte: Elaborado pelo autor (2026).

3.3.3 Grafo de Fluxo de Controle

Enquanto o Grafo de Dependência Estrutural representa a ordem em que as declarações de um módulo estão estabelecidas nele, o Grafo de Fluxo de Controle modela os caminhos de execução possíveis em uma função. Esses caminhos são capturados pelo *leitor_fluxo.py*, assim descrevendo a transferência de controle entre instruções para cada estrutura: laços *for* e *while*, condicionais *if/elif/else*, blocos *try/except* e instruções *switch*. Seis tipos de arestas são utilizados, descritos no Quadro 5.

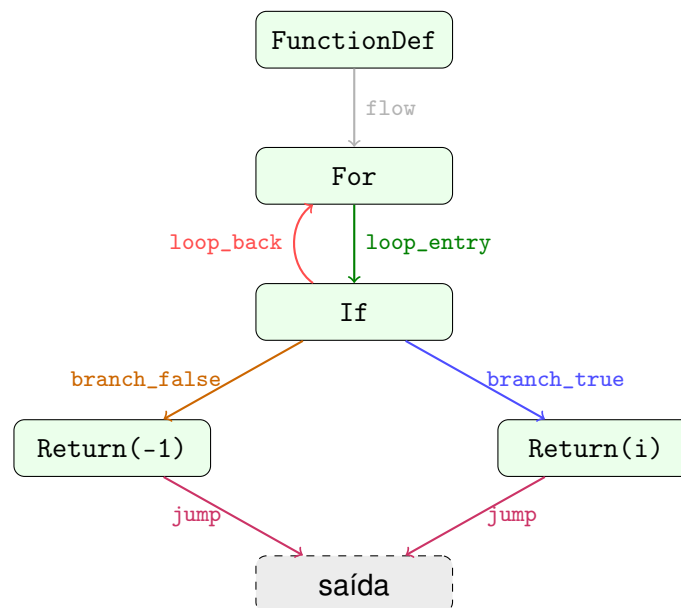
Quadro 5 – Tipos de aresta do Grafo de Fluxo de Controle.

Tipo de aresta	Semântica
<i>flow</i>	Execução sequencial direta entre dois nós no mesmo caminho de controle.
<i>branch_true</i>	Transferência de controle pelo ramo verdadeiro de um condicional.
<i>branch_false</i>	Transferência de controle pelo ramo falso de um condicional.
<i>loop_entry</i>	Transferência do nó de cabeçalho do laço para o primeiro nó do corpo.
<i>loop_back</i>	Aresta de retorno do último nó do corpo do laço ao nó de cabeçalho, representando a iteração.
<i>jump</i>	Transferência de controle por <i>break</i> , <i>continue</i> ou <i>return</i> .

Fonte: Elaborado pelo autor (2026).

A Figura 7 apresenta o GFC construído para a função de busca linear. A aresta de *loop_back* do nó *If* para o nó *For* representa a iteração do laço nos casos em que o elemento não é encontrado, enquanto a aresta de *jump* do nó *Return(i)* para fora do grafo representa a saída antecipada.

Figura 7 – Grafo de Fluxo de Controle da função de busca linear.



Fonte: Elaborado pelo autor (2026).

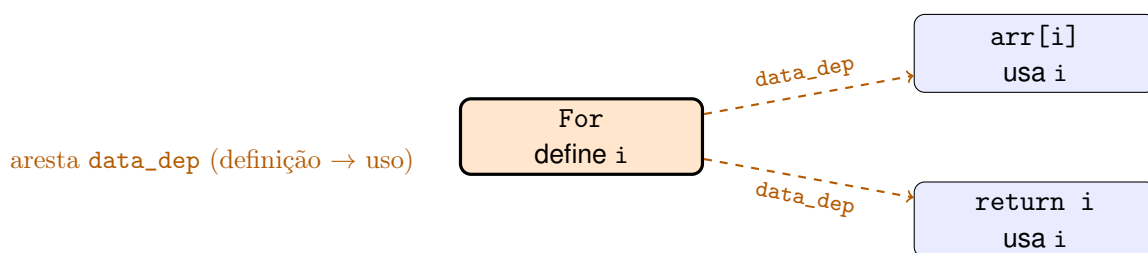
3.3.4 Grafo de Dependência de Dados

As arestas estruturais e as arestas de controle são complementadas pelas arestas de dependência de dados na construção de um Grafo de Dependência do Programa, conforme descrito por Ferrante *et al.* (1987). Essas últimas são adquiridas e analisadas pelo *leitor_pdg.py* e descrevem através de arestas direcionadas onde o código-fonte acessa e modifica a mesma variável ou valor. Essa variável constitui o

padrão *def-use* da teoria de compiladores (AHO *et al.*, 2006). Uma definição ocorre em atribuições, *Assign*, *AugAssign* ou na variável de iteração de laços, recebendo um identificador quando definida, enquanto o uso ocorre em qualquer referência posterior a esse identificador.

O leitor implementa essa análise em duas fases: na primeira, varre a árvore coletando todos os identificadores definidos e os nós de definição correspondentes. Na segunda, varre novamente a árvore e para cada referência a um identificador previamente definido, insere uma aresta de tipo *data_dep* do nó de definição ao nó de uso. Ambas varreduras restringem-se ao escopo de cada função e não rastreia dependências entre funções distintas. Essa característica visa se ater a lógica molecular previamente definida. Ao final, obtém-se um padrão radial onde cada nó de definição aponta para todos os nós que utilizam seu valor, como ilustrado na Figura 8.

Figura 8 – Arestas de dependência de dados para a variável *i* na função de busca linear.



Fonte: Elaborado pelo autor (2026).

Também é nessa seção que ocorre o cálculo do atributo *loop_depth*. Durante as leituras, um contador percorre por profundidade a árvore *ParseNode*, registrando os tipos de nós que iniciam um laço e ao entrar nesses incrementa seu valor em uma unidade. Ao final, armazena nas informações das moléculas o número de estruturas de laço aninhadas que contém o átomo. Por exemplo, um átomo *Assign* no interior de dois laços *for* consecutivos recebe *loop_depth* igual a 2, ao passo que uma instrução no nível raiz da função recebe valor 0. Esse atributo busca atuar como representante do aninhamento de laços, sendo o principal determinante de complexidade polinomial (CORMEN *et al.*, 2009, p. 43).

3.3.5 Grafo de Dominância

Os *leitor_atomico.py*, *leitor_topologico.py*, *leitor_fluxo.py* e *leitor_pdg.py* são módulos que constituem as operações fundamentais para operar a rede neural de grafos, entretanto alguns leitores adicionais foram implementados buscando-se encontrar características que estivessem relacionadas a complexidade temporal.

O código *leitor_dominancia.py* implementa o cálculo de relações de dominância entre os nós de controle da molécula. O algoritmo implementado é o iterativo de

Cooper *et al.* (2001), que computa o conjunto de dominadores imediatos de cada nó por meio de interseções de conjuntos sobre a ordem de pós visita do GFC.

A implementação adota o algoritmo de Cooper *et al.* (2001): primeiramente calcula os dominadores direto, para cada nó v do GFC, determina o nó d mais próximo de v tal que todo caminho do nó de entrada até v passe por d . Após, o algoritmo calcula os pós dominadores aplicando o mesmo algoritmo sobre o GFC invertido, isto é, com todas as arestas de direção revertida. Para cada par (d, v) identificado nos dois cálculos, o módulo insere as arestas correspondentes: arestas de tipo *dominance* do dominador imediato para o nó dominado, e arestas de tipo *post_dominance* do pós-dominador imediato para o nó pós-dominado. Essas arestas são adicionadas ao campo *edges* do *.mol.json*.

O passe de dominância é classificado como opcional no *pipeline* pois não integra a sequência padrão de dez passes executados automaticamente pelo módulo coordenador, mas pode ser ativado a depender da configuração. A decisão de torná-lo opcional deve-se ao fato de que busca-se encontrar a influência que esse aspecto causa nos resultados.

3.3.6 Detecção de Recursão

Além do leitor voltado a característica de dominância, outro leitor opcional é o módulo para recursão *leitor_recursao.py*, que percorre a lista de átomos da molécula em busca de nós do tipo *call* cujo identificador de chamada coincide com o nome da própria função sendo analisada. Quando tal chamada é encontrada, o atributo booleano *has_recursion* da molécula é marcado como verdadeiro e o átomo correspondente recebe o atributo *is_recursive_call = true*.

A motivação para este passe é que a recursão constitui um laço implícito não detectável pelos passes de fluxo (Seção 3.3.3) e PDG (Seção 3.3.4) (AHO *et al.*, 2006). Por exemplo, uma chamada recursiva de Fibonacci apresentaria seu grafo estrutural sem nenhuma aresta de *loop_back*, assim a única forma de detectar o padrão gerador de complexidade seria através do atributo *has_recursion*.

3.3.7 Assinatura Molecular e Biblioteca de Topologias

O conjunto de todos os grafos gerados pelos passes anteriores sobre um arquivo de código-fonte é registrado em bibliotecas de moléculas. A identificação de duas moléculas como a mesma estrutura é o que permite acumular evidências estatísticas entre arquivos e entre execuções. A identidade de uma molécula é determinada por uma cadeia de caracteres que codifica os tipos de átomos de controle presentes na função e seu aninhamento hierárquico denominada assinatura.

A assinatura é construída a partir das arestas de enlace da molécula, percorrendo a sub-árvore de átomos em profundidade e coletando os tipos que pertencem

ao conjunto de construções relevantes para a complexidade gerando cadeia parentética. Por exemplo, *FunctionDef(For(If(For)))* que representa uma função contendo um laço *for* dentro do qual há um condicional *If* contendo outro laço *For*. Átomos não pertencentes ao conjunto de construções relevantes para complexidade não aparecem na assinatura, portanto funções com mesma estrutura de controle mas com operações distintas produzem a mesma assinatura.

A assinatura é então aplicada a uma função de hash determinística que gera um identificador hexadecimal de 16 caracteres *mol_id*. Esse identificador é a chave primária de cada molécula na biblioteca e garante unicidade sem depender de strings potencialmente longos.

Por fim, a biblioteca classifica as moléculas em três categorias estruturalmente distintas, detectadas automaticamente durante o registro:

- Hierárquicas: assinaturas no formato parentético, como *FunctionDef(For(For))* ou *FunctionDef(For(If),While)*. Derivam diretamente da estrutura da AST e capturam o aninhamento de construções de controle. Constituem a maioria dos registros da biblioteca: 17 678 das 19 158 moléculas.
- Comunidades: assinaturas no formato de histograma entre chaves que representam subgrafos detectados pelo algoritmo de Louvain (descrito na Seção 3.3.9). Não capturam aninhamento, mas capturam a composição de tipos de átomos em padrões recorrentes não-hierárquicos. A biblioteca contém 839 moléculas dessa categoria.
- Simples: assinaturas primitivas sem aninhamento, como *For* ou *If*, geradas como sub-moléculas das hierárquicas. São 641 entradas na biblioteca.

Cada molécula é armazenada no arquivo *biblioteca_molecular.json* como um objeto com os seguintes campos:

- *signature*: cadeia canônica da molécula;
- *language*: linguagem de programação de origem;
- *sub_molecule_ids*: lista de *mol_id* de todas as sub-moléculas contidas nesta, resolvidas na biblioteca;
- *seen_count*: número de vezes que esta assinatura foi encontrada no conjunto de dados;
- *complexity_votes*: dicionário *{classe: contagem}* com os votos de complexidade acumulados;
- *atom_histogram*: dicionário *{tipo_átomo: contagem}*, preenchido apenas para moléculas do tipo comunidade;
- *is_community_mol*: booleano que distingue moléculas de comunidade das hierárquicas.

O registro de uma molécula é realizada pelo método *register_molecule* presente no código *biblioteca_molecular.py* em três fases sequenciais: na fase 1 o método chama *extract_sub_signatures* sobre a assinatura da molécula recém chegada. Esse método percorre a cadeia parentética em profundidade, coletando todas as sub-assinaturas contidas na hierarquia. Cada sub-assinatura ainda não presente na biblioteca recebe um registro próprio nessa fase.

Na fase 2 a molécula principal é registrada. Se o *mol_id* ainda não existe na biblioteca, cria-se o registro base. O campo *sub_molecule_ids* é preenchido pelo método interno *_relink* que percorre a biblioteca e resolve os identificadores de todas as sub-moléculas já presentes. Em seguida, o campo *seen_count* é incrementado e o rótulo da complexidade do arquivo de origem é processado, sendo adicionado a uma contagem *complexity_votes*.

Na última fase o método percorre as demais moléculas já registradas e executa *_relink* em todas as que contém a nova assinatura como sub molécula, atualizando seus campos *sub_molecule_ids*, ao final de cada chamada de *register_molecule*.

Durante a conversão para tensores (Seção 3.4), a biblioteca é consultada para enriquecer os atributos de entrada do modelo. O método *get_complexity_distribution* retorna a distribuição normalizada de votos de uma molécula: um vetor de oito probabilidades, uma por classe, calculada como a fração de votos em cada classe sobre o total de votos. Esse vetor é denominado *soft complexity* e é injetado como componente do vetor de característica de cada nó que pertence a molécula correspondente. O método *get_complexity_hint* retorna a classe com maior número de votos, utilizado como sinal de diagnóstico e como entrada opcional para o codificador de contexto de sub moléculas no modelo.

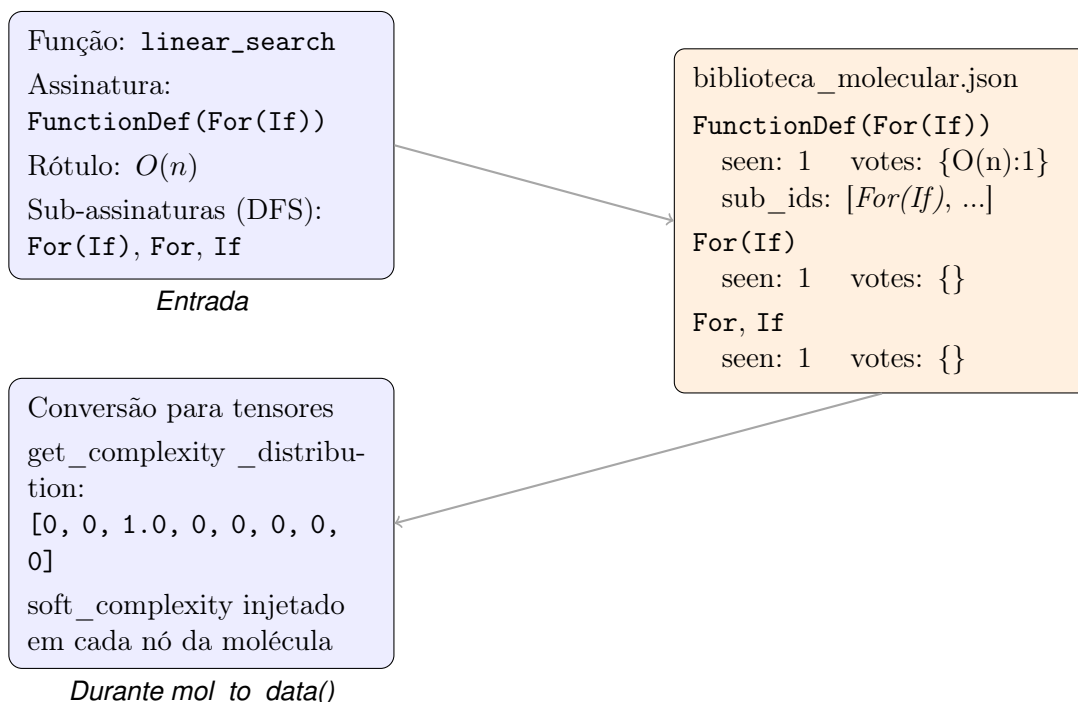
A Figura 9 ilustra o processo completo de registro e consulta para a função de busca linear.

3.3.8 Votos de Complexidade por Átomo

Além da biblioteca de moléculas descrita na Seção 3.3.7, o processo mantém uma biblioteca de nível atômico gerenciada pelo módulo *biblioteca_atomica.py*. Enquanto a biblioteca molecular acumula evidências sobre estruturas completas de função, a biblioteca atômica acumula, para cada tipo de átomo em cada linguagem, a distribuição empírica de classes de complexidade nos arquivos em que aquele tipo de átomo foi observado.

Cada entrada da biblioteca é indexada por uma chave composta no formato *linguagem::tipo*. Por exemplo, *"Python::For"* ou *"Java::while"*. A biblioteca armazena os registros no arquivo *biblioteca_atomica.json*, com duas chaves de nível superior: *"atoms"*, que armazena o conjunto de propriedades registradas por tipo, e *"complexity_votes"*, que armazena o dicionário de votos por chave.

Figura 9 – Registro de uma molécula na biblioteca e consulta durante a conversão para tensores.



Fonte: Elaborado pelo autor (2026).

O registro ocorre em dois contextos distintos: durante o passe atômico, cada novo tipo de nó encontrado em um arquivo é registrado via `register_atom`, que atualiza o conjunto de propriedades conhecidas para aquele tipo e, se um rótulo de complexidade for fornecido, incrementa o contador correspondente no dicionário de votos. Ao fim do processamento de cada arquivo completo, o método `register_code_atoms` registra todos os tipos de átomo do arquivo de uma vez, associando cada um ao rótulo de complexidade do arquivo. Rótulos compostos como $O(N \cdot W)$ são descartados nesta etapa pela mesma política adotada na biblioteca molecular: a função de normalização retorna `None` para esses casos e nenhum voto é registrado.

O método `get_soft_complexity_vector(language, atom_type)` retorna um vetor de oito posições, uma para cada classe de complexidade, com a distribuição empírica normalizada dos votos acumulados para aquele tipo de átomo naquela linguagem. Para calcular o vetor o método recupera o dicionário de votos da chave correspondente, soma o total de votos e divide a contagem de cada classe pelo total, produzindo um vetor de probabilidades que soma 1.

Dois casos especiais incorrem: se não há votos para o par (linguagem, tipo), o método realiza uma busca agregando os votos de todas as chaves que terminam em `::tipo`, independente da linguagem e calcula o vetor sobre esse conjunto agregado. Se mesmo assim não há votos disponíveis, situação encontrada no começo do processo, o método retorna o vetor com o valor 1/8 em todas as posições.

3.3.9 Detecção de Comunidades

O nono passe é feito novamente pelo *leitor_molecular.py*, utilizando a função *build_community_molecules* da *biblioteca_molecular.py*. Esse passe busca detectar subgrafos com grande números de conexões e com conexões fortes dentro de cada molécula e os registra como moléculas independentes na biblioteca.

O passe coleta todos os átomos da molécula e todas as arestas estruturais e de fluxo produzidas pelos passes anteriores. A partir desses dados, constrói um grafo não dirigido ponderado utilizando a biblioteca NetworkX, onde cada átomo é um nó e cada aresta recebe um peso numérico de acordo com o seu tipo semântico.

O Quadro 6 lista os pesos adotados.

Quadro 6 – Pesos de aresta para detecção de comunidades.

Tipo de aresta	Peso	Justificativa
<i>enlace, loop_entry, loop_back</i>	1,0	Acoplamento forte: contenção sintática e iteração.
<i>call</i>	0,9	Acoplamento direto entre nós de chamada.
<i>enlace_transitive, func_entry</i>	0,6–0,8	Acoplamento indireto de contenção transitiva.
<i>branch_true, branch_false</i>	0,7	Bifurcações condicionais.
<i>exception, loop_exit</i>	0,4–0,5	Caminhos excepcionais.
<i>sequencia, seq</i>	0,3	Acoplamento fraco: ordem de aparição.

Fonte: Elaborado pelo autor (2026).

Quando dois átomos são conectados por arestas de tipos diferentes, o grafo preserva apenas a aresta com o maior peso, evitando contagem dupla. O passe executa o algoritmo de Louvain por meio da função *louvain_communities* do NetworkX, com semente fixa *seed=42* para reprodutibilidade. O algoritmo de Louvain maximiza iterativamente a modularidade Q do grafo, definida como a proporção de arestas que caem dentro das comunidades subtraída do valor esperado sob uma distribuição aleatório com os mesmos graus de nó (BLONDEL *et al.*, 2008). A cada iteração cada nó é movido para a comunidade vizinha que maximiza o ganho de modularidade, repetindo esse processo até convergir.

Para cada comunidade com mais de três átomos, o passe calcula uma assinatura do formato de histograma *tipo:contagem*, com os tipos de átomos da comunidade ordenados alfabeticamente. Essa representação captura a composição da comunidade sem pressupor hierarquia entre os átomos, distinguindo-se das assinaturas das moléculas hierárquicas.

Cada comunidade recebe ainda um identificador textual sequencial dentro do arquivo e um histograma completo dos tipos de átomos que a compõem. O conjunto

de comunidades detectadas é passado ao método *register_community_molecules* da *BibliotecaMolecular*, que cria e atualiza o registro de cada comunidade na biblioteca e acumula os votos de complexidade do arquivo de origem, seguindo a mesma política de normalização descrita na Seção 3.3.7. Os identificadores das moléculas de comunidade registradas são armazenadas no campo *community_mols_ids* da molécula principal e incluídos no arquivo *.mol.json*, mas não substituem a assinatura e estrutura original da molécula. Ao término do processamento completo da base de dados, a biblioteca contém 839 moléculas do tipo comunidade.

3.3.10 Segmentação por Função

O décimo e último passe do processo de leitura é o *FunctionSplitterPass*, implementado no módulo *leitor_funcional.py*. O passe divide o grafo de um arquivo em subgrafos individuais por função, de modo que cada função de nível superior se torne um exemplo de treinamento independente. A motivação é que arquivos com múltiplas funções de complexidade distintas não podem ser tratados como um único exemplo sem que o rótulo de uma função contamine a representação da outra.

O passe opera sobre o *MoleculeGraph* construído pelos leitores anteriores, sem reler o código fonte. O algoritmo inicia identificando o átomo raiz do arquivo, usualmente o nó do tipo *Module* e na ausência desse o do tipo *FunctionDef*, desde que esse não seja alvo de nenhuma aresta de enlace. A partir desse nó raiz, o leitor percorre os filhos diretos via arestas de enlace e seleciona aqueles cujo pertence ao conjunto de definições de função: *FunctionDef*, *AsyncFunctionDef* e *Lambda*.

Quando nenhum dos nós *Module* ou de função é encontrado na raiz, o passe realiza uma busca por todos os átomos do tipo funcional do grafo, identifica quais deles aparecem como filhos de outros via procura por largura nas arestas de enlace e descarta os aninhados. As funções remanescentes são tratadas como de nível superior.

Para cada função de nível superior identificada, o leitor coleta a sua sub-árvore completa via procura por largura nas arestas de enlace, partindo do átomo da função como raiz. Esse procedimento garante que átomos de função aninhadas, como funções definidas dentro de outra função, sejam incluídos na sub árvore da função que as contém e não como exemplos independentes. Em seguida, o leitor extrai todas as arestas do grafo cujos nós de origem e destino estejam ambos dentro da sub árvore, separando-as por tipo: estruturais, de fluxo, de dependência de dados e de dominância.

Com os átomos e arestas do subgrafo isolados, o passe calcula localmente os atributos escalares *max_loop_depth* e *has_recursion* para essa função específica, sem depender dos valores globais do arquivo. O nome da função é extraído da propriedade *name* do átomo *functionDef* e a assinatura da função é calculada pelo método

`_subtree_sig`, que percorre a sub árvore de enlace e constrói a cadeia parentética a partir dos tipos de átomo de controle presente.

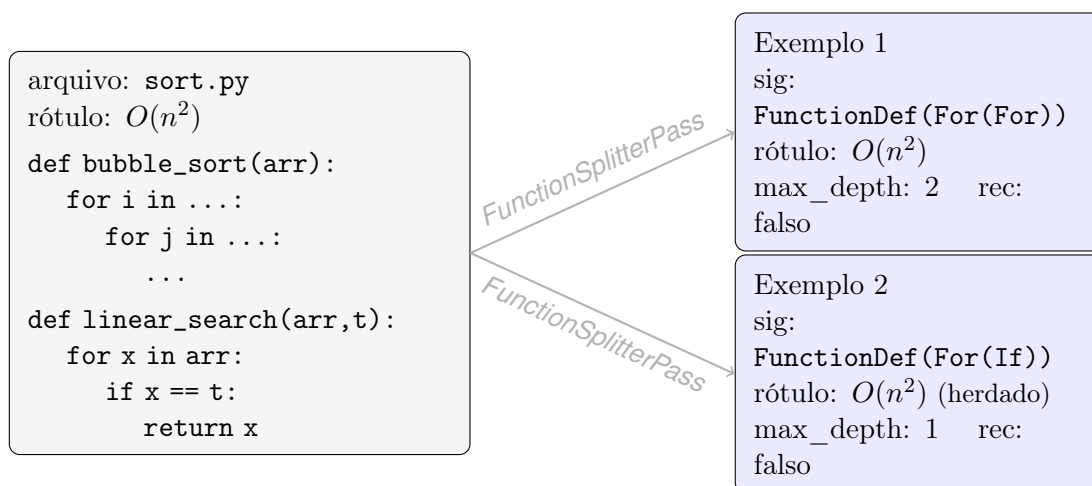
O subgrafo resultante é então registrado na biblioteca molecular como uma molécula independente e armazenado no campo `mol.extra["function_graphs"]` do `MoleculeGraph`. O rótulo de complexidade associado ao subgrafo é herdado do rótulo do arquivo de origem.

Na etapa de carregamento do conjunto de dados para o treinamento, a função `mol_to_data_list` processa cada arquivo `.mol.json` e verifica a presença do campo `function_graphs`. Se presente, itera sobre a lista e converte da subgrafo em um objeto `torch_geometric.data.Data` independente, retornando uma lista com um exemplo por função. Exemplos cujo rótulo de complexidade não foi mapeado são excluídos do treinamento.

Arquivos que não contêm nenhuma definição de função como algoritmos procedurais ou arquivos com apenas código de nível de módulo não produzem subgrafos. Nesses casos o passe utiliza o grafo do arquivo inteiro como único exemplo, sem segmentação.

A Figura 10 ilustra a segmentação de um arquivo com duas funções em dois exemplos de treinamento independentes.

Figura 10 – Segmentação de um arquivo com duas funções em dois exemplos de treinamento independentes.



Fonte: Elaborado pelo autor (2026).

Ao término da leitura, a base de dados estruturada contém 12 376 arquivos `.mol.json`, cada um com o campo `function_graphs` preenchido. O carregamento desses arquivos para o treinamento produz 37 608 exemplos válidos com o rótulo mapeado e 53 199 subgrafos de funções no total, dos quais os exemplos sem rótulo válido são excluídos.

3.4 CONVERSÃO PARA TENSORES

Ao término dos passes de análise, cada função do conjunto de dados está representada como um subgrafo no formato produzido pela *FunctionSplitterPass*. Antes de ser processada pela rede neural, essa representação precisa ser convertida em objetos no formato nativo da biblioteca *PyTorch Geometrics* para grafos. Essa conversão é realizada pela função *mol_to_data*, implementado no módulo *gnn_modular.py* e envolve três operações: a construção do vetor de características de aresta e a montagem dos tensores de conectividade.

Para cada átomo do subgrafo, a função *atom_to_features* constrói um vetor de características de comprimento fixo. Na configuração padrão da GNN, esse vetor tem 29 dimensões, organizadas em seis blocos que podem ser ativados ou desativados independentemente via *FeatureSet*.

O vetor de características é construído na ordem descrita no Quadro 7. Os 14 valores no bloco *CATEGORY* ocupam sempre as primeiras posições, seguidos dos demais blocos na sequência. Essa ordenação é fixa e determinística, garantindo que o mesmo tipo de átomo produza sempre o mesmo vetor de características em qualquer execução. Todos os valores são do tipo flutuante.

Ao final da construção de todos os vetores de nó, a função empilha-os em uma matriz $\mathbf{X} \in \mathbb{R}^{N \times 29}$ (GILMER *et al.*, 2017; FEY; LENSSEN, 2019), onde N é o número de átomos na função, e a converte para o tensor x do objeto *Data*.

Para cada aresta de subgrafo, a função *edge_to_features* consulta o *EdgeTypeRegistry* e constrói um vetor *one-hot* de comprimento igual ao número de tipos de aresta registrados. O índice ativado corresponde à posição do tipo da aresta no registro conforme atribuído durante o processamento do conjunto de dados e registrado no arquivo *edge_registry.json*. O comprimento do vetor cresce automaticamente quando novos tipos de aresta são descobertos, de modo que a dimensão não é fixada em código, mas determinada pelo estado atual do registro no momento da conversão.

Quadro 7 – Blocos do vetor de características de nó (configuração padrão: 29 dimensões).

Índices	Dims	Bloco	Descrição
0–13	14	<i>CATEGORY</i>	Vetor <i>one-hot</i> da categoria semântica do átomo. As 14 categorias são: <i>ACCESS</i> , <i>COMPARATOR</i> , <i>COMPREHENSION</i> , <i>CONTROL</i> , <i>CTX</i> , <i>DATA_FLOW</i> , <i>IMPORT</i> , <i>LITERAL</i> , <i>LOGICAL</i> , <i>OPERATION</i> , <i>OPERATOR</i> , <i>STATEMENT</i> , <i>STRUCTURAL</i> e <i>UNKNOWN</i> . Cada átomo pertence a exatamente uma categoria.
14–17	4	<i>STRUCTURAL_FLAGS</i>	Quatro sinalizadores binários (0,0 ou 1,0): <i>is_loop</i> (<i>For</i> , <i>While</i> , <i>AsyncFor</i>), <i>is_branch</i> (<i>If</i> , <i>Match</i> , <i>IfExp</i>), <i>is_function</i> (<i>FunctionDef</i> , <i>AsyncFunctionDef</i> , <i>Lambda</i>), <i>is_call</i> (<i>Call</i>).
18	1	<i>LINENO</i>	Número de linha do átomo no código-fonte, normalizado pelo número máximo de linhas na função: $lineno / \max(max_lineno, 1)$.
19	1	<i>COMPLEXITY_REL</i>	Sinalizador binário que indica se o tipo do átomo pertence a uma categoria considerada relevante para complexidade computacional (<i>CONTROL</i> , <i>STRUCTURAL</i> , <i>COMPREHENSION</i> ou <i>OPERATION</i>).
20–27	8	<i>SOFT_COMPLEXITY</i>	Distribuição empírica de complexidade fornecida pela <i>BibliotecaAtomica</i> : vetor de 8 probabilidades, uma por classe, calculado sobre os votos acumulados para o tipo do átomo na linguagem de origem. Quando não há votos disponíveis, utiliza distribuição uniforme.
28	1	<i>LOOP_DEPTH</i>	Profundidade de aninhamento de laços do átomo, normalizada por um fator empírico de 6 (maior <i>loop_depth</i> encontrado): $loop_depth / 6, 0$.

Fonte: Elaborado pelo autor (2026).

Nem todos os tipos de arestas são incluídos em todas as configurações. O campo *edge_feature_set* da *GNNConfig* descrito na Seção 3.6 controla quais camada de aresta participam da conversão. Todos os vetores de aresta são empilhados na matriz $\mathbf{E} \in \mathbb{R}^{|E| \times d_e}$, onde d_e é a dimensão de aresta no registro. Após serem empilhados, são convertidos para o tensor *edge_attr* (GILMER *et al.*, 2017; FEY; LENSSEN, 2019). A conectividade do grafo é representada pela matriz *edge_index* $\in \mathbb{Z}^{2 \times |E|}$, cujas colu-

nas contêm os pares (origem, destino) de cada aresta.

Além dos tensores de nó e aresta, dois atributos escalares são calculados por função e armazenados diretamente no objeto *Data*:

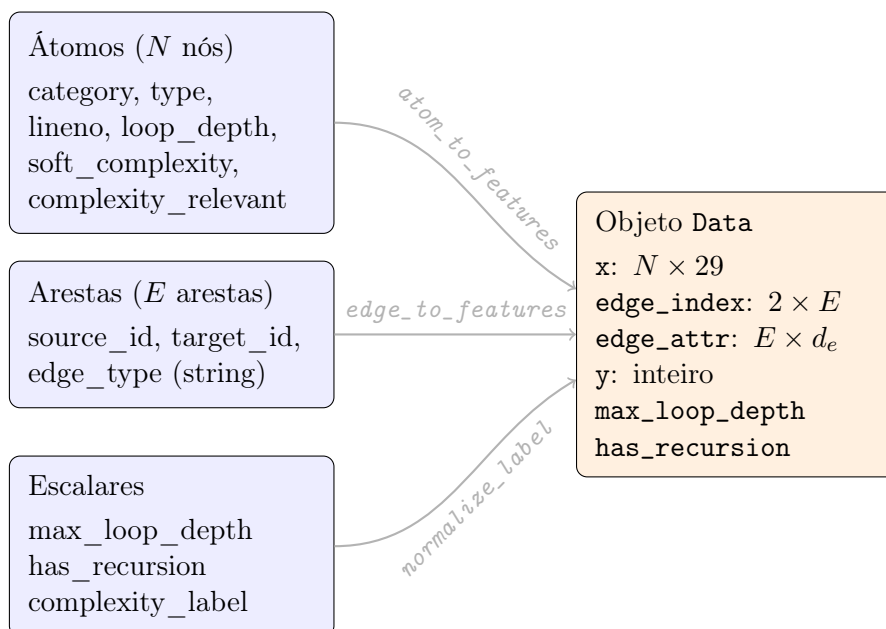
- *max_loop_depth*: valor máximo de *loop_depth* entre todos os átomos da função, normalizado pelo mesmo fator de 6 usado no bloco *LOOP_DEPTH*;
- *has_recursion*: valor binário (0,0 ou 1,0) indicando se a função contém chamada recursiva detectada pelo passe de recursão.

Esses valores não participam da passagem de mensagem das camadas GNN pois são propriedades globais da função, não de nós individuais. Em vez disso, são concatenados ao vetor de grafo produzido pelo agrupamento imediatamente antes do classificador, conforme descrito na Seção 3.6.

O campo *y* do objeto *Data* recebe o índice inteiro da classe de complexidade atribuída a função, obtido por *normalize_label* aplicado ao rótulo do metadado. Funções cujo rótulo não é mapeado a nenhuma das oito classes válidas recebem $y = -1$ e são excluídas do conjunto de dados antes do treinamento.

A Figura 11 demonstra as entradas e saídas da conversão para uma função com N átomos e E arestas.

Figura 11 – Conversão de um sub-grafo de função em objeto *Data*.



Fonte: Elaborado pelo autor (2026).

3.5 ARQUITETURA DA REDE NEURAL DE GRAFOS

Com a base de dados preparada e convertida, essa pode ser enfim treinada. O modelo implementado é denominado *ComplexityGNN* e opera sobre objetos *Data*

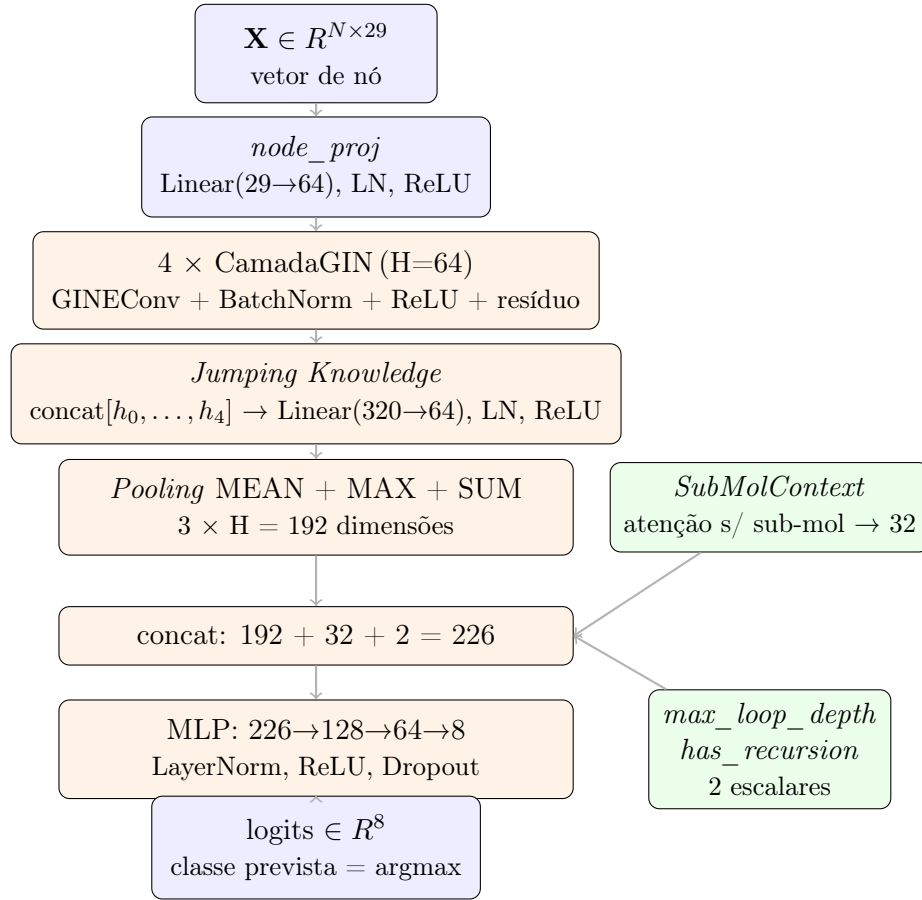
produzidos pela etapa de conversão. A arquitetura é inteiramente configurável via *GNNConfig*, permitindo alterar o tipo de camada, a estratégia de agrupamento e as características ativas.

A arquitetura combina quatro decisões principais, cada uma motivada por propriedades diferentes do problema.

1. Rede de grafos isomórfica: Xu *et al.* (2019) demonstra que as GIN são a variante de GNN com maior capacidade de distinguir estruturas de subgrafo distintas. Para o problema em questão, dois grafos com padrões de aninhamento diferentes devem necessariamente produzir representações distintas, sendo GIN a variante especializada nisso.
2. *Jumping Knowledge*: Xu *et al.* (2018) desenvolve um mecanismo que permite que o modelo combine representações de diferentes profundidades de propagação, capturando simultaneamente o sinal local e o global. Dessa forma, busca-se usar informações referentes a complexidade presentes tanto no átomo quanto no aninhamento ao seu redor.
3. Agrupamento combinado: *MEAN_POOL*, *MAX_POOL* e *SUM_POOL*. Cada um desses agrupamentos reforça um dado presente na molécula, respectivamente a densidade de construções de controle, o pior caso local e o volume total de estruturas presentes (XU *et al.*, 2019).
4. Conexões residuais em todas as camadas: Yu *et al.* (2023) identifica que o empilhamento de muitas camadas de mensagem produz rendimentos decrescentes em vez de crescentes devido ao fenômeno de sobresuavização. As conexões residuais que somam a representação de entrada à saída de cada camada atenuam esse efeito ao preservar a informação original.

A Figura 12 ilustra o fluxo de processamento na configuração padrão.

Figura 12 – Fluxo de processamento na configuração padrão.



Fonte: Elaborado pelo autor (2026).

O vetor de características de nó de 29 dimensões construído na Seção 3.4 é projetado para o espaço oculto de dimensão $H = 64$ pela camada *node_proj*:

$$\mathbf{h}_v^{(0)} = \text{ReLU}(\text{LayerNorm}(\mathbf{W}_{\text{node}} \mathbf{x}_v + \mathbf{b}_{\text{node}})) \quad (15)$$

onde $\mathbf{x}_v \in R^{29}$ é o vetor de características do nó v e $\mathbf{W}_{\text{node}} \in R^{64 \times 29}$ é a matriz de projeção aprendida. A normalização por *LayerNorm* antes da ativação estabiliza os gradientes no início do treinamento. As características de aresta são projetadas para o mesmo espaço R^H por uma camada linear simples *edge_proj* sem normalização.

A camada de Rede de grafo de isomorfismo implementada utiliza a extensão *GINEConv* que incorpora características de aresta durante a agregação. A atualização do nó v na iteração k é

$$\mathbf{h}_v^{(k)} = \text{MLP} \left((1 + \varepsilon^{(k)}) \cdot \mathbf{h}_v^{(k-1)} + \sum_{u \in \mathcal{N}(v)} \phi(\mathbf{h}_u^{(k-1)}, \mathbf{e}_{uv}) \right) \quad (16)$$

onde $\mathcal{N}(v)$ é o conjunto de vizinhos de v , $\mathbf{e}_{uv}$ é o vetor de características da aresta (u, v) após a projeção *edge_proj*, ϕ é uma função de combinação de nó e baseada

no framework MPNN (GILMER *et al.*, 2017) e implementada internamente pelo *GINE-Conv* e $\varepsilon^{(k)}$ é um parâmetro escalar aprendido que controla o peso relativo da própria representação do nó em relação às mensagens recebidas dos vizinhos. O MLP interno possui duas camadas:

$$\text{MLP}(\mathbf{z}) = \mathbf{W}_2 \text{ReLU}(\text{BN}(\mathbf{W}_1 \mathbf{z})) \quad (17)$$

onde $\mathbf{W}_1 \in R^{128 \times H}$ projeta o dobro da dimensão oculta e $\mathbf{W}_2 \in R^{H \times 128}$ projeta de volta para H . A normalização BatchNorm (BN) entre as duas camadas lineares do MLP interno estabiliza a escala das ativações (IOFFE; SZEGEDY, 2015). Após a convolução, aplica-se uma conexão residual:

$$\mathbf{h}_v^{(k)} \leftarrow \text{ReLU}(\text{BN}(\text{GINEConv}(\mathbf{h}_v^{(k-1)}))) + \mathbf{h}_v^{(k-1)} \quad (18)$$

Quando as dimensões de entrada e saída diferem, o que ocorre apenas na primeira camada se $\text{node_dim} \neq H$, uma projeção linear $\mathbf{W}_{\text{res}} \in R^{H \times \text{node_dim}}$ é aplicada ao termo residual antes da soma.

O modelo empilha 4 camadas ($L = 4$) para equilibrar entre alcance de vizinhança e risco de sobreesuavização. Com esse número de camadas cada nó agrega informação até quatro saltos em seu entorno, o que é suficiente para capturar o padrão de aninhamento de até quatro laços consecutivos. Trabalhos sobre classificação de código com GNN (YU *et al.*, 2023) reportam que o desempenho frequentemente atinge platô ou degrada com mais de quatro camadas de propagação, mesmo sem evidência de sobreesuavização.

O mecanismo de *Jumping Knowledge* proposto por Xu *et al.* (2018) foi introduzido para resolver um conflito relacionado ao alcance do nó. O alcance da vizinhança de um nó cresce com a profundidade de rede mas o alcance ótimo depende da posição do nó do grafo. Após as 4 camadas de passagem de mensagens, o modelo coleta as representações de todos os estágios de propagação, incluindo a representação após a projeção inicial:

$$\mathbf{h}_v^{\text{JK}} = \mathbf{h}_v^{(0)} \parallel \mathbf{h}_v^{(1)} \parallel \mathbf{h}_v^{(2)} \parallel \mathbf{h}_v^{(3)} \parallel \mathbf{h}_v^{(4)} \quad (19)$$

onde $\parallel$ denota concatenação, produzindo um vetor de dimensão $H \times (L + 1) = 64 \times 5 = 320$. O modo de agregação utilizado é a concatenação que preserva todas as representações intermediárias. Enquanto isso, *MAX* ou atenção *LSTM* ponderam as escalas (XU *et al.*, 2018). O vetor concatenado é então projetado de volta para R^H por uma camada linear com *LayerNorm* e *ReLU*:

$$\mathbf{h}_v = \text{ReLU}(\text{LayerNorm}(\mathbf{W}_{\text{JK}} \mathbf{h}_v^{\text{JK}})), \quad \mathbf{W}_{\text{JK}} \in R^{H \times 5H} \quad (20)$$

A projeção garante que o vetor de saída por nó mantenha a dimensão H independente do número de camadas, o que é necessário para que as operações de agrupamento subsequentes operem em dimensão fixa. As operações de agrupamento operam sobre os vetores em um único vetor de representação do grafo. A função *readout* $\mathbf{h}_G = \text{READOUT}(\{\mathbf{h}_v \mid v \in G\})$ introduzida na estrutura MPNN (GILMER *et al.*, 2017) devem ser invariante à permutação de nós.

Com essa operação configurada, o modelo implementa as três formas de agrupamento mencionadas:

$$\mathbf{p}_{MEAN} = \frac{1}{|V|} \sum_{v \in V} \mathbf{h}_v \quad (21)$$

$$\mathbf{p}_{MAX} = \max_{v \in V} \mathbf{h}_v \quad (22)$$

$$\mathbf{p}_{SUM} = \sum_{v \in V} \mathbf{h}_v \quad (23)$$

onde o máximo em (22) é aplicado a cada elemento. Os três vetores são então concatenados em:

$$\mathbf{p}_G = \mathbf{p}_{MEAN} \parallel \mathbf{p}_{MAX} \parallel \mathbf{p}_{SUM}, \quad \mathbf{p}_G \in R^{3H} \quad (24)$$

A motivação para a combinação reside nas diferentes propriedades de cada operação. O resultado de (21) é proporcional à fração de cada tipo de representação na função, onde funções com poucos laços e muitas atribuições exibem um vetor médio distinto das funções com muitos laços. O resultado de (22) captura o pior caso local, enquanto o resultado de (23) é sensível ao volume total de construções. Assim, *pool_max* é usado para encontrar estruturas de controle profundamente aninhadas enquanto *pool_sum* encontra cenários de larga iteração.

Xu *et al.* (2019) demonstra que a agregação por soma é estritamente superior que a média para distinguir multiconjuntos de vizinhos. A combinação das três operações aproveita as propriedades complementares de cada uma sem descartar informação.

Além dos três métodos listados de agrupamento, o modelo também suporta agrupamento por *GlobalAttention*. Nesse método, uma rede aprende um escalar de importância por nó: $g_v = \text{Tanh}(\mathbf{W}_2 \text{Linear}(\mathbf{h}_v))$, e o vetor do grafo é a soma ponderada pelas atenções *softmax*.

O *SubMolContextEncoder* é um módulo adicional que transmite ao classificador a informação acumulada na *BibliotecaMolecular* sobre as submoléculas das funções. Cada molécula tem seu campo *sub_molecule_ids* e esse módulo aponta para as submoléculas registradas na biblioteca, cada uma contendo um vetor de voto de complexidade de 8 dimensões.

O *SubMolContextEncoder* recebe uma matriz de votos $S \in R^{M \times 8}$, onde M é o número de submoléculas, e produz um vetor de contexto de 32 dimensões por função:

$$\mathbf{e}_m = \text{ReLU}(\mathbf{W}_v \mathbf{s}_m), \quad \mathbf{e}_m \in R^{32} \quad (25)$$

$$\alpha_m = \frac{\exp(a_m)}{\sum_j \exp(a_j)}, \quad a_m = \mathbf{w}_a^\top \mathbf{e}_m \quad (26)$$

$$\mathbf{c} = \sum_{m=1}^M \alpha_m \mathbf{e}_m \quad (27)$$

onde $\mathbf{W}_v \in R^{32 \times 8}$ é a matriz de projeção dos votos e $\mathbf{w}_a \in R^{32}$ é o vetor de atenção. Submoléculas sem votos acumulados são mascaradas com $-\infty$ antes da operação *softmax* em (26), evitando que contribuam para o contexto; Quando todas as moléculas estão mascaradas, o módulo retorna $\mathbf{c} = \mathbf{0}$.

A motivação para este módulo é que a biblioteca acumula evidências estatísticas sobre padrões moleculares ao longo de todo o conjunto de dados. Injetar um vetor com essas evidências no classificador permite que o modelo utilize conhecimento coletivo acumulado para além da função sendo analisada.

Antes do classificador, o vetor de acúmulo $\mathbf{p}_G \in R^{192}$, o vetor de contexto de submoléculas $\mathbf{c} \in R^{32}$ e os dois escalares *max_loop_depth* normalizado e *has_recursion* são concatenados:

$$\mathbf{z} = \mathbf{p}_G \parallel \mathbf{c} \parallel \text{max_loop_depth} \parallel \text{has_recursion}, \quad \mathbf{z} \in R^{226} \quad (28)$$

Os dois escalares são injetados nessa etapa para que sinais globais da função sejam disponibilizados diretamente ao classificador sem distorcer as representações nodais durante a propagação. O classificador é um MLP de três camadas com *LayerNorm* e *Dropout*:

$$\hat{\mathbf{y}} = \mathbf{W}_3 \text{ReLU}(\text{LN}(\mathbf{W}_2 \text{ReLU}(\text{LN}(\mathbf{W}_1 \mathbf{z})))) \quad (29)$$

com dimensões $\mathbf{W}_1 \in R^{128 \times 226}$, $\mathbf{W}_2 \in R^{64 \times 128}$, $\mathbf{W}_3 \in R^{8 \times 64}$, *Dropout*(0,3) após a primeira camada oculta e *Dropout* (0,15) após a segunda. O uso de *LayerNorm* em vez de *BatchNorm* nessas camadas deve-se à estabilidade superior durante a estimativa de arquivos individuais *batch_size* = 1. O vetor $\hat{\mathbf{y}} \in R^8$ contém os *logits* não normalizados de cada classe e a classe prevista é $\hat{y} = \arg \max_c \hat{y}_c$.

O Quadro 8 resume as dimensões de cada componente.

Quadro 8 – Dimensões dos componentes do *ComplexityGNN* na configuração de referência.

Componente	Dim. saída	Observações
<i>node_proj</i>	$N \times 64$	Linear($29 \rightarrow 64$), LayerNorm, ReLU. Projeção inicial de nó.
<i>edge_proj</i>	$E \times 64$	Linear($d_e \rightarrow 64$). Ausente quando <i>edge_dim</i> = 0.
CamadaGIN $\times 4$	$N \times 64$	GINEConv + BN + ReLU + resíduo. ε aprendido.
<i>Jumping Knowledge</i>	$N \times 64$	Concat de 5 camadas (320 dim) $\rightarrow$ Linear($320 \rightarrow 64$), LN, ReLU.
Pooling MEAN+MAX+SUM	192	Três operações globais concatenadas: $3 \times H = 192$.
<i>SubMolContextEncoder</i>	32	Atenção sobre vetores de votos das sub-moléculas.
Escalares	2	<i>max_loop_depth</i> e <i>has_recursion</i> .
Vetor <i>z</i>	226	Entrada do classificador: $192 + 32 + 2 = 226$.
MLP classificador	8	Linear($226 \rightarrow 128 \rightarrow 64 \rightarrow 8$), LN, ReLU, Dropout.

Fonte: Elaborado pelo autor (2026).

3.6 CONFIGURAÇÃO E TREINAMENTO DA REDE NEURAL

O treinamento da rede neural é gerenciado pela classe *TreinadorModular*, implementada no código *gnn_modular.py*. A classe recebe um modelo *ComplexityGNN*, uma *GNNConfig* e a lista de objetos *Data* produzido pela etapa de conversão. Após, essa classe executa o ciclo completo de treinamento com divisão de dados, otimização, agendamento da taxa de aprendizado e salvamento de progresso.

O conjunto de 37 608 exemplos é dividido em três partições: 70% para treino, 15% para validação e 15% para teste. Antes de particionar, os conjuntos são embaralhados com semente fixa (*random.seed(42)*), garantindo que execuções distintas produzam exatamente a mesma divisão. A Tabela 2 resume as proporções e tamanhos resultantes.

Tabela 2 – Divisão do conjunto de dados para treinamento.

Partição	Proporção	Exemplos	Papel
Treino	70%	26 326	Atualização de pesos
Validação	15%	5641	Monitoramento
Teste	15%	5641	Avaliação final

Fonte: Elaborado pelo autor (2026).

Os hiperparâmetros do modelo e do treinamento são centralizados na classe *GNNConfig*. O Quadro 9 lista os valores padrão utilizados.

Quadro 9 – Hiperparâmetros padrão da *GNNConfig*.

Parâmetro	Valor	Descrição
<i>num_layers</i>	4	Número de camadas de passagem de mensagens.
<i>hidden_dim</i>	64	Dimensão dos vetores de representação ocultos.
<i>dropout</i>	0,3	Taxa de <i>dropout</i> aplicada após cada camada.
<i>lr</i>	10^{-3}	Taxa de aprendizado inicial do otimizador.
<i>weight_decay</i>	10^{-4}	Fator de regularização L2 do otimizador.
<i>epochs</i>	100	Número máximo de épocas de treinamento.
<i>batch_size</i>	16	Número de grafos por <i>mini-batch</i> .
<i>patience</i>	15	Épocas sem melhoria requeridas para interrupção antecipada.

Fonte: Elaborado pelo autor (2026).

Conforme descrito na Seção 2.2.5.1, as quatro arquiteturas de GNN diferem fundamentalmente em seu mecanismo de agregação de vizinhança. A implementação realizada neste trabalho adapta cada variante ao problema de classificação de complexidade por meio de três modificações:

1. Conexões residuais: a representação de entrada $\mathbf{h}_v^{(k-1)}$ é somada à saída de cada camada, atenuando a sobresuavização que converge as representações nodais a um subespaço indistinguível em redes profundas (YU *et al.*, 2023);
2. Normalização por BatchNorm: aplicada após a convolução e antes da ativação, estabiliza a escala das ativações ao longo do treinamento (IOFFE; SZEGEDY, 2015);
3. Intercambialidade de dimensões: todas as variantes são instanciadas com as mesmas dimensões de entrada e saída ($H = 64$) e podem ser substituídas entre si sem alteração.

As especificações de implementação de cada variante são descritas a seguir. Os resultados empíricos comparativos encontram-se na Seção 4.3.2.

A variante GIN é a camada padrão e é implementada por meio do operador *GINEConv* da biblioteca PyTorch Geometric (FEY; LENSSEN, 2019). O *GINEConv* estende a GIN original (Equação 5) com suporte a características de arestas, incorporando o vetor $\mathbf{e}_{uv}$ de cada aresta diretamente na mensagem enviada pelo nó u ao nó v (GILMER *et al.*, 2017), conforme a Equação 16. A extensão para arestas é essencial para este trabalho: tipos de arestas como *loop_entry*, *loop_back* e *branch_true* carregam semântica distinta de controle de fluxo, e sua diferenciação durante a agregação é um sinal direto de padrões de complexidade.

O MLP interno possui duas camadas lineares com dimensão intermediária $2H = 128$, norma por *BatchNorm1d* entre elas, e ativação ReLU:

$$\text{MLP}(\mathbf{z}) = \mathbf{W}_2 \text{ReLU}(\text{BN}(\mathbf{W}_1 \mathbf{z})) \quad (30)$$

com $\mathbf{W}_1 \in R^{2H \times H}$ e $\mathbf{W}_2 \in R^{H \times 2H}$. A dimensão intermediária $2H$ confere maior capacidade de representação sem elevar o custo quadraticamente. Após a convolução, aplica-se a conexão residual com projeção linear opcional primeira camada, onde $\text{node_dim} = 29 \neq H = 64$:

$$\mathbf{h}_v^{(k)} \leftarrow \text{ReLU}(\text{BN}(\text{GINEConv}(\mathbf{h}_v^{(k-1)}))) + \begin{cases} \mathbf{W}_{\text{res}} \mathbf{h}_v^{(k-1)} & \text{se in_dim} \neq H \\ \mathbf{h}_v^{(k-1)} & \text{caso contrário} \end{cases} \quad (31)$$

A variante GCN é implementada por meio de *GCNConv*, que realiza a convolução espectral descrita pela Equação 2. A implementação acrescenta à formulação original a conexão residual e a normalização por BatchNorm:

$$\mathbf{h}_v^{(k)} \leftarrow \text{ReLU} \left(\text{BN} \left(\sum_{u \in \mathcal{N}(v) \cup \{v\}} \frac{1}{\sqrt{\hat{d}_u \hat{d}_v}} \mathbf{W}^{(k)} \mathbf{h}_u^{(k-1)} \right) \right) + \mathbf{h}_v^{(k-1)} \quad (32)$$

A principal limitação da GCN para grafos de código é que a normalização por grau atenua o sinal de nós com muitas conexões que acumulam arestas de enlace, *loop_entry* e *loop_back* simultaneamente. Nós com alto grau de saída contribuem com mensagens de menor magnitude por vizinho, o que pode reduzir o sinal discriminativo associado a estruturas de maior aninhamento.

A variante GAT é implementada com quatro fontes de atenção paralelas ($L = 4$), cada uma com dimensão $H/L = 64/4 = 16$; as saídas são concatenadas, mantendo a dimensão de saída $H = 64$. Diferentemente da formulação original de Veličković *et al.* (2017), a *GATConv* do PyTorch Geometric permite a inclusão de características de arestas no cálculo dos coeficientes de atenção, estendendo a Equação 3:

$$\alpha_{vu}^{(\ell)} = \frac{\exp(\text{LReLU}(\mathbf{a}^{(\ell)\top} [\mathbf{W}^{(\ell)} \mathbf{h}_v \parallel \mathbf{W}^{(\ell)} \mathbf{h}_u \parallel \mathbf{W}_e \mathbf{e}_{vu}]))}{\sum_{w \in \mathcal{N}(v)} \exp(\text{LReLU}(\mathbf{a}^{(\ell)\top} [\mathbf{W}^{(\ell)} \mathbf{h}_v \parallel \mathbf{W}^{(\ell)} \mathbf{h}_w \parallel \mathbf{W}_e \mathbf{e}_{vw}]))} \quad (33)$$

onde LReLU denota a função LeakyReLU, $\mathbf{a}^{(\ell)} \in R^{3 \cdot (H/L)}$ é o vetor de atenção da fonte ℓ e $\mathbf{W}_e$ projeta o vetor de característica da aresta para o espaço de atenção. Os coeficientes das quatro fontes são calculados com *dropout* de 0,1 aplicado sobre os pesos de atenção antes da ponderação, funcionando como regularização específica do mecanismo de atenção. A inclusão de características de arestas no cálculo permite que a GAT atribua pesos distintos a arestas de *loop_entry* versus *branch_false*, por

exemplo, potencialmente capturando a diferença semântica entre iteração e bifurcação condicional. A conexão residual é adicionada com a mesma estrutura das demais variantes (Equação 31).

A variante GraphSAGE é implementada por *SAGEConv* com função de agregação por média, conforme proposto por Hamilton *et al.* (2017). A formulação de implementação, com conexão residual e BatchNorm, é:

$$\mathbf{h}_v^{(k)} \leftarrow \text{ReLU} \left(\text{BN} \left(\mathbf{W}^{(k)} \left[\mathbf{h}_v^{(k-1)} \parallel \frac{1}{|\mathcal{N}(v)|} \sum_{u \in \mathcal{N}(v)} \mathbf{h}_u^{(k-1)} \right] \right) \right) + \mathbf{h}_v^{(k-1)} \quad (34)$$

A concatenação da auto-representação com a média dos vizinhos distingue o GraphSAGE da GCN: enquanto a GCN mistura os dois termos em uma única soma normalizada, o GraphSAGE os trata como entidades separadas antes da projeção linear, preservando a identidade do nó central.

O conjunto de dados apresenta desbalanceamento entre as oito classes de complexidade. Para mitigar o viés do modelo em direção às classes mais frequente, o treinador calcula pesos de classe inversamente proporcionais à frequência de cada classe no conjunto de treino:

$$w_c = \frac{N_{\text{treino}}}{C \times n_c} \quad (35)$$

onde N_{treino} é o total de exemplos no conjunto de treino, $C = 8$ é o número de classes e n_c é o número de exemplos da classe c . Esses pesos são passados à função de perda de peso, de modo que exemplos de classes menos representadas contribuam mais para o gradiente.

Na configuração padrão, a função de perda é a entropia cruzada ponderada com *class_weights*, instanciada via *build_criterion(config, class_weights)*. O critério é selecionado pelos campos *use_ordinal_loss* e *use_label_smoothing* da *GNNConfig*. Caso não haja chamada com um critério diferente, a padrão *CrossEntropyLoss* com os pesos calculados pela Equação 35.

O otimizador utilizado é o AdamW (LOSHCHILOV; HUTTER, 2019), com taxa de aprendizado inicial $\eta = 10^{-3}$ e fator de regularização $\lambda = 10^{-4}$. Assim o otimizador é aplicado em um *scheduler* do tipo *ReduceLROnPlateau* configurado para monitorar a acurácia no conjunto de validação *mode="max"*. Quando a acurácia de validação não melhora em 5 épocas consecutivas a taxa de aprendizado corrente é multiplicada por um fator de 0,5, respeitando um limite inferior de 10^{-5} .

Em cada passo de otimização, antes da atualização dos parâmetros, aplica-se recorte de gradiente com norma 1,0:

$$\mathbf{g} \leftarrow \mathbf{g} \cdot \min \left(1, \frac{1,0}{\|\mathbf{g}\|_2} \right) \quad (36)$$

onde g é o vetor concatenado de todos os gradientes do modelo. Esse procedimento previne instabilidade numérica em lotes cujos gradientes apresentam norma elevada.

O treinamento pode ser interrompido quando a acurácia de validação não supera o valor máximo histórico por *patience* = 15 épocas consecutivas. Sempre que a acurácia de validação atinge um novo máximo, o treinador salva o estado atual do modelo no arquivo *best.pt*. O *checkpoint* contém o *state_dict* do modelo, a acurácia de validação correspondente, o nome da configuração, o dicionário completo de hiperparâmetros, a dimensão do vetor de características de nó, a dimensão do vetor de características de aresta e o número de classes. A dimensão de aresta é salva no *checkpoint* para suportar a compatibilidade retroativa descrita na Seção 3.3.5, onde os vetores de aresta são truncados à dimensão registrada.

O módulo implementa dois critérios adicionais que exploram a estrutura das oito classes de complexidade configuráveis pelos campos *use_ordinal_loss* e *use_label_smoothing* da *GNNConfig*. Essa suavização quando configurada substitui a distribuição alvo *one-hot* por uma distribuição que transfere uma fração $\varepsilon/2$ da massa de probabilidade para cada uma das classes ordinalmente adjacentes à classe verdadeira. Para classes de fronteira ($c = 0$ ou $c = C - 1$), o deslocamento $\varepsilon/2$ é aplicado apenas ao vizinho existente, e a massa não transferida permanece na classe verdadeira, garantindo que a distribuição some 1 em todos os casos. Para a classe c verdadeira com coeficiente da suavização ε *label_smoothing*, padrão 0,1:

$$P_{\text{alvo}}[k] = \begin{cases} 1 - \varepsilon & \text{se } k = c \\ \varepsilon/2 & \text{se } k = c \pm 1 \\ 0 & \text{caso contrário} \end{cases} \quad (37)$$

Por exemplo, para a classe verdadeira $O(n^2)$ de índice 4 com $\varepsilon = 0,1$ o vetor alvo é $[0, 0, 0, 0,05, 0,90, 0,05, 0, 0]$. A função de perda resultante é a divergência de Kullback-Leibler entre P_{alvo} e a distribuição prevista (KULLBACK; LEIBLER, 1951). Pesos de classe podem ser aplicados sobre essa perda.

A penalidade ordinal (*use_ordinal_loss=True*) combina a entropia cruzada com um termo proporcional ao erro de distância ordinal entre a classe prevista e a classe verdadeira:

$$\mathcal{L} = (1 - w) \cdot \mathcal{L}_{\text{CE}} + w \cdot \frac{1}{B} \sum_{i=1}^B |\hat{r}_i - r_{y_i}| \quad (38)$$

onde w é o peso ordinal padrão 0,3, $r_c = c/(C - 1)$ é o ranque normalizado da classe c na escala ordinal e $\hat{r}_i$ é o ranque esperado segundo a distribuição prevista:

$$\hat{r}_i = \sum_{k=0}^{C-1} p_{ik} \cdot r_k \quad (39)$$

sendo p_{ik} a probabilidade *softmax* da classe k para o exemplo i . O uso da expectativa sobre as probabilidades em vez do *argmax* torna o termo diferenciável e fornece gradiente útil em todas as épocas do treinamento. A expectativa é utilizada em detrimento da função *argmax* porque a operação de maximização estrita não é diferenciável, o que impossibilitaria a retropropagação do gradiente de erro para a rede. Com $w = 0,3$ o modelo recebe penalidade proporcional ao erro ordinal cometido, gerando penalidade maior quando um erro incorre em uma estimativa mais distante. Por exemplo, um código $O(n)$ ser errado como $O(n^2)$ tem menor penalidade que um $O(1)$ com $O(n!)$.

Quando ambos os campos são ativados (*use_ordinal_loss=True* e *use_label_smoothing=True*), a função de perda utiliza *OrdinalLabelSmoothingLoss* como critério interno do *OrdinalLoss*, substituindo a entropia cruzada padrão pelo critério suavizado. O gradiente do termo CE na Equação 38 passa, portanto, pela distribuição alvo suavizada da Equação 37. O Quadro 10 resume as quatro configurações possíveis de critério.

Quadro 10 – Configurações possíveis do critério de perda via *build_criterion*.

<i>use_ordinal_loss</i>	<i>use_label_smoothing</i>	Critério resultante
<i>False</i>	<i>False</i>	<i>CrossEntropyLoss</i> com pesos de classe (padrão).
<i>False</i>	<i>True</i>	<i>OrdinalLabelSmoothingLoss</i> : CE com suavização para as classes vizinhas na escala ordinal.
<i>True</i>	<i>False</i>	<i>OrdinalLoss</i> : combinação de CE e penalidade proporcional ao erro de distância ordinal.
<i>True</i>	<i>True</i>	Loss combinada: <i>OrdinalLoss</i> com <i>OrdinalLabelSmoothingLoss</i> como CE interna.

Fonte: Elaborado pelo autor (2026).

3.7 DECLARAÇÃO DE USO DE INTELIGÊNCIA ARTIFICIAL

Ferramentas de inteligência artificial generativa foram utilizadas como auxílio no desenvolvimento deste trabalho, especificamente para a revisão gramatical de trechos do texto e da documentação, para o aprimoramento da consistência e coesão entre as seções e para a geração de figuras ilustrativas. A ferramenta utilizada foi o Gemini Pro 2.5 e o 3.0.

O uso dessas ferramentas não implica coautoria. O autor é integralmente responsável pelo conteúdo, pelas análises, pelos resultados e pelas conclusões apresentadas. Todas as afirmações técnicas e científicas foram verificadas pelo autor com base nas referências bibliográficas citadas.

4 RESULTADOS E DISCUSSÕES

Este capítulo apresenta os resultados experimentais obtidos com a arquitetura descrita no Capítulo 3. A abordagem proposta nesse trabalho parte da hipótese de que a representação de código-fonte como grafo de dependências, combinada com uma rede neural capaz de operar sobre essa estrutura é suficiente para classificar a complexidade de tempo em oito classes. Os resultados apresentados nesse capítulo avaliam o desempenho absoluto do modelo sobre o conjunto de dados completo, a contribuição de cada componente por meio de um estudo ablativo e como modelo se compara a trabalhos anteriores.

4.1 CONFIGURAÇÃO EXPERIMENTAL

Todos os experimentos foram conduzidos em uma única máquina equipada com processador Intel Core i7, 16 GB de memória RAM e uma unidade de processamento gráfico NVIDIA RTX 4050 com suporte a CUDA, utilizada para as operações de treinamento e inferência. O ambiente de software utilizou Python 3.11, PyTorch 2.5.1 e PyTorch Geometric, com semente de aleatoriedade fixada em 42 para inicialização dos pesos do modelo, embaralhamento do conjunto de dados e todas as operações estocásticas internas da biblioteca.

O processo de análise sintática descrito na Seção 3.3 concebeu um conjunto de dados completo de 12376 arquivos de código-fonte e arquivos *.mol.json* e 37608 exemplos válidos após a segmentação por função e a filtragem de rótulos inválidos. A Tabela 3 demonstra o tempo acumulado por passe de análise, medido sobre a amostra de 2842 arquivos do conjunto *GeeksForGeeks*.

Tabela 3 – Tempo de processamento por passe de análise sintática (2842 arquivos, média de 0,91 s/arquivo).

Passe	Tempo total (s)	Média (s/arquivo)
<i>function_splitter</i>	1182,8	0,416
<i>signature</i>	700,4	0,246
<i>community</i>	586,6	0,206
<i>soft_votes</i>	41,3	0,015
<i>atomic</i>	38,2	0,013
<i>topological</i>	8,2	0,003
<i>flow</i>	5,7	0,002
<i>pdg</i>	3,7	0,001
<i>transitive</i>	2,6	0,001
<i>dominance</i>	1,4	0,000
<i>recursion</i>	0,2	0,000
Total	2579,5	0,908

Fonte: Elaborado pelo autor (2026).

Os passes *function_splitter*, *signature* e *community* respondem, juntos, por 96% do tempo total de processamento. O tempo dominante do primeiro justifica-se pela execução de buscas em largura para a extração das subárvores funcionais. Em contrapartida, o tempo dos leitores de arestas se encontra consideravelmente baixo, entretanto provavelmente esses não estão sendo ativados pelo leitor coordenador. A investigação das causas desse comportamento constitui direção para trabalhos futuros.

O conjunto combinado de base de dados apresenta desbalanceamento acentuado com metade das classes, $O(n)$, $O(n^2)$, $O(n \log n)$ e $O(n^3)$, sendo 84,6% dos exemplos. Enquanto isso, $O(\log n)$, $O(2^n)$ e $O(n!)$ representa apenas 1,5% dos exemplos. A distribuição de classes no conjunto de validação é apresentada na Tabela 4.

Tabela 4 – Distribuição de exemplos por classe no conjunto de validação (5641 exemplos).

Classe	Exemplos	Proporção
$O(1)$	786	13,9%
$O(\log n)$	50	0,9%
$O(n)$	1459	25,9%
$O(n \log n)$	1152	20,4%
$O(n^2)$	1241	22,0%
$O(n^3)$	921	16,3%
$O(2^n)$	10	0,2%
$O(n!)$	22	0,4%
Total	5641	100,0%

Fonte: Elaborado pelo autor (2026).

A configuração de referência utiliza a arquitetura GIN com quatro camadas *Jumping Knowledge* por concatenação, agrupamento combinado de *MEAN+MAX+SUM*, o conjunto completo de seis blocos de características atômicas descritas no Quadro 7 e quinze tipos de arestas registrados no *edge_registry.json*. As características de *label_smoothing* e *ordinal_loss* não foram utilizadas.

O modelo de referência possui 146637 parâmetros treináveis e foi treinado por 119 épocas com parada antecipada ao não superar a acurácia de validação 0,699 por 15 épocas consecutivas. O tempo total de treinamento foi de 45,8 minutos com média de 23,1 s por época.

4.2 RESULTADOS DO MODELO

A Tabela 5 apresenta o desempenho do modelo no conjunto de validação.

Tabela 5 – Métricas de avaliação do modelo de referência no conjunto de validação.

Métrica	Valor
Acurácia exata	67,6%
Acurácia adjacente ($k=1$)	80,9%
Acurácia adjacente ($k=2$)	92,0%
MAOE	0,634 classes
MAOE normalizado	0,091
Cohen's κ	0,595
F1 macro	0,580
F1 ponderado	0,678
Eficiência binária — Acurácia	83,6%
Eficiência binária — F1	0,797
Eficiência binária — AUC-ROC	0,926

Fonte: Elaborado pelo autor (2026).

A acurácia de 67,6% indica a quantidade relativa de estimativas coincidentes com o rótulo verdadeiro dos exemplos. A métrica de adjacência de 80,9% e a de eficiência binária de 83,6% demonstra que o modelo comete erros próximos na escala ordinal, não estimando previsões consideravelmente diferentes da classe correta.

O MAOE normalizado de 0,091 quantifica o quão longe a estimativa erra na escala ordinal, correspondendo a aproximadamente 0,64 posições de diferença média. O κ de Cohen de 0,595 situa o modelo na faixa de concordância moderada segundo Landis e Koch (1977).

A classificação binária de eficiência, que agrupa as classes em eficientes, $O(1)$ a $O(n \log n)$, e ineficientes, acima de $O(n^2)$, produz AUC-ROC de 0,926. Esse resultado indica que o modelo aprende a distinguir confiavelmente entre os dois grupos.

A Tabela 6 detalha as métricas por classe de complexidade no conjunto de validação.

Tabela 6 – Precisão, *recall* e F1 por classe no conjunto de validação.

Classe	Precisão	<i>Recall</i>	F1	Exemplos
$O(1)$	0,660	0,681	0,670	786
$O(\log n)$	0,306	0,600	0,405	50
$O(n)$	0,654	0,585	0,618	1459
$O(n \log n)$	0,665	0,613	0,638	1152
$O(n^2)$	0,621	0,747	0,678	1241
$O(n^3)$	0,907	0,811	0,856	921
$O(2^n)$	0,217	0,500	0,303	10
$O(n!)$	0,440	0,500	0,468	22

Fonte: Elaborado pelo autor (2026).

A classe $O(n^3)$ obteve o melhor desempenho nas métricas de precisão, *recall* e F1, respectivamente 0,907, 0,811 e 0,856. Esse resultado reflete a presença de características marcantes de complexidade temporal, como 3 loops encadeados, gerando assinaturas moleculares suficientemente distintas. As classes $O(\log n)$, $O(2^n)$ e $O(n!)$ tiveram o pior desempenho nas métricas, possivelmente devido a quantidade insuficiente de exemplos dos repositórios originais.

4.3 ESTUDOS DE ABLAÇÃO

A ablação constitui um procedimento sistemático de remoção de componentes individuais de um modelo, com o objetivo de quantificar empiricamente a contribuição de cada um para o desempenho geral do sistema. Nessa seção conduz-se um estudo de ablação sobre o conjunto completo de dados e com a mesma divisão do modelo completo, utilizando acurácia de validação como critério de comparação. Diferentes combinações de características foram testadas em múltiplos estudos buscando-se encontrar a que possuía melhor desempenho com esse critério.

4.3.1 Características de nó

A Tabela 7 apresenta o efeito da remoção de cada bloco de características de nó ou átomo sobre o desempenho. Os dados são comparados com a GNN padrão com acurácia de validação.

Tabela 7 – Efeito de cada bloco de características de nó.

Configuração	Acc.	Δ	Épocas
GNN referência	67,6%	—	108
Apenas <i>CATEGORY</i>	34,3%	−33,3 pp	50
Sem <i>SOFT_COMPLEXITY</i>	39,4%	−28,2 pp	62
Sem <i>LINENO</i>	65,3%	−2,3 pp	86
Sem <i>LOOP_DEPTH</i>	67,1%	−0,5 pp	108

Fonte: Elaborado pelo autor (2026).

O bloco *SOFT_COMPLEXITY* demonstra-se o componente mais crítico de vetor de características, causando uma redução de 28,2 pp. Sem esse vetor o modelo não parece aprender representações discriminativas. A configuração com apenas o bloco *CATEGORY* também confirma essa conclusão, operando com desempenho reduzido. O bloco *LINENO* têm contribuição ínfima, causando uma redução de apenas 2,3 pp, enquanto *LOOP_DEPTH* apresentou uma redução ainda menor de apenas 0,5 pp.

4.3.2 Tipo de Camada

A Tabela 8 compara os diferentes tipos de camadas de passagem de mensagens e a profundidade da rede. Os dados são comparados com a GNN padrão com acurácia de validação, sendo essa a GIN + JK com 4 camadas de profundidade.

Tabela 8 – Comparação entre tipos de camada e profundidade.

Configuração	Val acc	Δ	Parâmetros
GNN referência	67,6%	—	125 965
GIN $\times 4$ + JK, $H=128$	71,4%	+3,8 pp	563 277
GIN $\times 6$ + JK	68,6%	+1,0 pp	197 071
GraphSAGE $\times 4$	68,2%	+0,6 pp	95 689
GIN $\times 2$ + JK	66,6%	−1,0 pp	96 203
GCN $\times 4$	66,6%	−1,0 pp	79 305
GAT $\times 4$	64,9%	−2,7 pp	96 457
GAT $\times 4$ sem JK	62,9%	−4,7 pp	75 785

Fonte: Elaborado pelo autor (2026).

A GIN com quatro camadas e *Jumping Knowledge* possui o melhor desempenho entre as variações avaliadas. O aumento da dimensão oculta para $H = 128$ produz ganho de 3,8 pp, mas com 4,5 vezes mais parâmetros. Nessa configuração, a acurácia de treinamento atingiu aproximadamente 75,1%, gerando uma diferença de 3,7 pp em relação a de validação. Essa diferença indica que o modelo está memorizando padrões do conjunto que não se generalizam para dados não vistos. Os parâmetros em excesso conferem capacidade de representação além do necessário para o conjunto de dados disponível. Seis camadas adicionam apenas 1,0 pp com convergência mais lenta, exigindo 15 épocas a mais para estabilizar. GCN com duas camadas produz queda simétrica de 1,0 pp, sugerindo que quatro camadas constituem o equilíbrio adequado entre profundidade de propagação e estabilidade para esse conjunto.

4.3.3 Estratégia de agrupamento

A Tabela 9 compara as operações de agrupamento global individualmente e em combinações. Os dados são comparados com a GNN padrão com acurácia de validação, sendo essa a configuração de *MEAN+MAX+SUM*.

Tabela 9 – Comparação entre estratégias de agrupamento.

Configuração	Val acc	Δ
GNN referência	67,6%	—
MEAN (único)	70,0%	+2,4 pp
MAX (único)	68,5%	+0,9 pp
ATT (único)	68,3%	+0,7 pp
SUM (único)	67,4%	-0,2 pp
MEAN+ATT	69,7%	+2,1 pp
MEAN+MAX+ATT	68,6%	+1,0 pp
MEAN+MAX	68,0%	+0,4 pp
MEAN+SUM	67,4%	-0,2 pp
MEAN+MAX+SUM+ATT	73,2%	+5,6 pp

Fonte: Elaborado pelo autor (2026).

A combinação com todas as estratégias de agrupamento produziu o melhor resultado, com 73,2% de acurácia, representando um ganho de 5,6 pp sobre a referência. Todos os testes com agrupamento por atenção (*Attention*) apresentaram ganhos, enquanto o agrupamento *MEAN* obteve o maior ganho de forma isolada. Nota-se que, com a remoção do agrupamento *MAX*, os experimentos contendo *SUM* sofreram perdas, levantando a hipótese de que o mecanismo *SUM* introduz ruídos que prejudicam os demais sinais. Entretanto, como a combinação dos agrupamentos teve consideravelmente o melhor resultado, levanta-se a hipótese de que o agrupamento por atenção opere como um filtro nesses ruídos, aprimorando todos os sinais de agrupamento quando combinado a eles, já que os resultados que não englobam todos, mostram que esse reduz o desempenho dos agrupamentos isolados.

4.3.4 Tipos de aresta

A Tabela 10 compara configurações de tipos de aresta. Os dados são comparados com a GNN padrão com acurácia de validação, sendo essa com todas as características ativas.

Tabela 10 – Comparação entre configurações de tipos de aresta.

Configuração	Acc.	Δ
GNN referência	67,6%	—
Flow only	68,7%	+1,1 pp
Modo FULL	68,8%	+1,2 pp
Sem <i>data_dep</i>	68,6%	+1,0 pp
Apenas estruturais	65,1%	-2,5 pp

Fonte: Elaborado pelo autor (2026).

Arestas de fluxo de controle isoladas superam a configuração completa em 1,1 pp. As arestas *loop_entry*, *loop_back*, *branch_true* e *branch_false* carregam o sinal mais significativo para complexidade. O tipo *enlace_transitive*, que representa 35% do volume total de arestas, é redundante com o atributo *loop_depth* nos nós e pode diluir os gradientes das arestas mais informativas. As arestas de dependência de dados têm impacto marginal +1,0 pp ao removê-las. Esses resultados, combinado com o tempo de leitura dessas características, demonstram que o único leitor funcionalmente ativo é o de fluxo. A reformulação desses leitores, com validação sobre exemplos controlados, constitui trabalho futuro.

4.3.5 Contexto e escalares globais

A ablação dos componentes injetados após o agrupamento foi conduzida a partir de configuração diferente, utilizando apenas a estratégia de agrupamento *MAX* e apenas com aplicação de arestas de fluxo, resultando em 70,4% de acurácia de validação. O modelo de referência não foi utilizado como ponto de partida pois esses experimentos foram conduzidos após a identificação dessas melhorias em experimentos anteriores. A Tabela 11 apresenta o efeito da remoção de cada componente de contexto global a partir dessa configuração.

Tabela 11 – Efeito da remoção de componentes de contexto global.

Configuração	Val acc	Δ
Modelo referência	70,4%	—
Sem <i>has_recursion</i>	69,0%	−1,5 pp
Sem <i>sub_mol_context</i>	68,7%	−1,7 pp
Sem todos os extras (escalares + submol)	67,8%	−2,6 pp
Sem <i>max_loop_depth</i> + <i>has_recursion</i>	67,3%	−3,1 pp

Fonte: Elaborado pelo autor (2026).

Nota-se que todos os escalares contribuem para a estimativa de complexidade, entretanto a remoção simultânea de todos os componentes extras produz menor queda do que a que envolve apenas os escalares, sugerindo que há sobreposição parcial entre sinais dos escalares e do contexto molecular.

4.3.6 Configuração ótima

A partir dos achados das subseções anteriores, as decisões de configuração com melhor desempenho isolado foram progressivamente combinadas, produzindo uma sequência de configurações ótimas descrita na Tabela 12.

Tabela 12 – Evolução das configurações ótimas ao longo das séries de experimentos.

Configuração	Val acc	Teste acc	Épocas	Parâmetros
GNN referência	67,6%	69,9% ^a	119	146 637
<i>optimal</i>	70,4%	71,1%	114	109 389
<i>optimal_v2_no_smooth</i>	70,5%	72,3%	134	125 644
<i>opt_att_max_pool</i>	72,1%	73,6%	119	119 694
<i>optimal_v3</i>	73,1%	75,3%	197	115 277

^a Resultado do treino de referência descrito na Seção 4.2.

Fonte: Elaborado pelo autor (2026).

A configuração *optimal_v3*, que obteve o maior valor de acurácia de validação registrado combina agrupamento de atenção, penalidade ordinal com peso $w = 0,4$ e suavização de rótulos com $\varepsilon = 0,02$ sobre o grafo com arestas de fluxo. Em relação ao modelo base, o ganho é de 5,8 pp com redução de parâmetros de 146.637 para 115.277.

4.4 COMPARAÇÃO COM TRABALHOS RELACIONADOS

A comparação entre o modelo proposto e trabalhos que compartilham o problema de classificação de complexidade temporal diferem em dimensões metodológicas e impedem comparação direta entre as métricas reportadas. O Quadro 11 sintetiza as principais dimensões de divergência.

Quadro 11 – Comparação metodológica entre os trabalhos avaliados.

Aspecto	Este trabalho	Pfitscher <i>et al</i> (2023)	RODENBUSCH (2023)
Representação	Grafo de dependências (GNN)	16 características manuais	16 características manuais (AutoML)
Linguagens	8 linguagens	Java	Java
Número de classes	8	8 (após expansão)	8 (com sublinnear)
Balanceamento	Pesos de classe inversos	SMOTE antes da divisão	Nenhum
Divisão <i>dataset</i>	70/15/15	70/30/0	100/0/0
Conjunto de dados	<i>GFG + Co-deComplex</i>	<i>GFG + Cor-CoD</i>	<i>GFG + Cor-CoD</i>
Exemplos de treino	26 326	$\approx 550^a$	$\approx 1668^b$

^a Estimativa a partir do Merged dataset de 786 amostras com divisão 70/30.

^b Dataset Merged completo; AutoGluon treina sobre 100% dos dados sem partição de teste explícita.

Fonte: Elaborado pelo autor (2026).

Uma diferença considerável entre esse trabalho e o de Pfitscher *et al.* (2023) está na aplicação de SMOTE antes da separação entre treino, que, conforme apontado por Rodenbusch (2023), as amostras sintéticas geradas a partir do conjunto completo podem ter contaminado a partição de teste inflacionado artificialmente as métricas reportadas.

Outra diferença está nas linguagens que os modelos lidam, ambos trabalhos lidam apenas com JAVA, o que limita a capacidade de generalização a outras línguas. Enquanto neste trabalho, o modelo proposto opera sobre oito linguagens distintas de programação, o que aumenta a variância do espaço de entrada.

Por fim, a falta de um conjunto de teste explícito em Rodenbusch (2023) impede a estimativa direta de variância das métricas reportadas, já que o AutoGluonTabular treina sobre 100% dos dados disponíveis.

4.4.1 Resultados sobre o conjunto GeeksForGeeks

O trabalho de Pfitscher *et al.* (2023) é o mais diretamente comparável, pois utiliza-se do conjunto *GeeksForGeeks*, derivada do *Web Scraper* de Rodenbusch (2023). A comparação sobre esse conjunto é relevante, ainda que haja diferenças no processo de extração, na quantidade de amostras e nas linguagens incluídas. Assim, foi conduzido um experimento com as configurações padrões da GNN mas com apenas o repositório mencionado como base de dados.

A Tabela 13 apresenta todas as métricas disponíveis para cada modelo. Métricas não reportadas pelos trabalhos originais são indicadas com traço.

Tabela 13 – Comparação de acurácia entre os trabalhos avaliados.

Modelo	Acurácia de classe	Acurácia de eficiência
RF (PFITSCHER <i>et al.</i> , 2023) ^a	89,8%	90,2%
RF (PFITSCHER <i>et al.</i> , 2023) - GFG ^b	80,0%	—
AutoGluon (RODENBUSCH, 2023) ^c	77,6%	88,4%
GNN - GFG ^d	45,4%	79,3%
GNN referência ^e	67,6%	83,6%

^a SMOTE antes da divisão; GFG + CorCoD.

^b Teste feito apenas com o GFG como treino e validação, não foi feito estudo de eficiência.

^c AutoGluon *best_quality*; GFG + CorCoD; 100% treino, sem partição de teste.

^d GFG isolado; 21 épocas; condição experimental subótima com parada antecipada.

^e GFG + CodeComplex; 119 épocas.

Fonte: Elaborado pelo autor (2026).

A configuração padrão do modelo proposto obtém 67,6% de acurácia de classe, valor inferior em 22,2 pp ao *Random Forest* de Pfitscher *et al.* (2023) e em 10,0 pp ao AutoGluon de Rodenbusch (2023). Para a classificação binária de eficiência, a do modelo proposto é inferior em 6,6 pp e 4,8 pp, respectivamente.

Em comparação, o modelo que rodou apenas sobre a base de dados *Geeks-ForGeeks* é consideravelmente inferior aos outros modelos, resultado esperado dado o volume reduzido de dados disponíveis para treinamento. A acurácia do modelo reduzido obteve 45,4%, 22,2pp inferior ao modelo padrão, e 44,4 pp e 32,2 em comparação com Pfitscher *et al.* (2023) e Rodenbusch (2023). A acurácia binária ainda foi menor que os outros, entretanto com uma diferença consideravelmente menor: 79,3%, inferior respectivamente em 4,3, 10,9 e 9,1 pp's.

Três possíveis pontos explicam essa diferença, sendo o primeiro o fato de que os modelos comparados operam sobre características numéricas extraídas manualmente de código Java, aspectos diretamente relacionados à complexidade de código. O modelo GNN não recebe informações explicitamente, extraindo de dados como a estrutura e topologia do grafo.

Outro ponto é a variedade do problema. Com oito linguagens de programação, o espaço de variação dos grafos de entrada é substancialmente maior do que nas configurações que utilizam apenas Java. Classes como $O(\log n)$ cujo comportamento algorítmico é independente de linguagem, manifestam-se em padrões sintáticos distintos conforme a linguagem, aumentando a dificuldade de classificação.

Por fim, Rodenbusch (2023) demonstra que o *Random Forest* de Pfitscher *et al.* (2023) reduziu sua acurácia em 36 pp ao ser avaliado sobre o *CorCoD dataset*.

O AutoGluon, nas mesmas condições, reduz em apenas 2,7 pp, evidenciando maior robustez à mudança de domínio. O modelo de GNN proposto foi treinado e avaliado sobre duas fontes distintas com partições de treino, teste e validação, o que impõe condições mais rigorosas de avaliação do que os outros trabalhos.

A Tabela 14 posiciona esses resultados de generalização lado a lado.

Tabela 14 – Acurácia de classe em condições de generalização entre fontes de dados distintas.

Modelo	Treino	Teste	Acc. classe
<i>Random Forest</i>	<i>GFG</i>	<i>CorCoD</i>	44,0%
AutoGluon	<i>GFG</i>	<i>CorCoD</i>	73,9%
GNN referência	GFG + <i>CodeComplex</i>	GFG + <i>CodeComplex</i>	67,6% ^a

^a: Acurácia de validação

Nota: a avaliação do modelo GNN utiliza divisão aleatória entre as duas fontes, não constituindo um teste de domínio cruzado estrito. Realizar treinamento sobre um e testar o outro constitui trabalho futuro.

Fonte: Elaborado pelo autor (2026).

4.5 DISCUSSÃO

Os resultados apresentados nesse capítulo permitem uma interpretação integrada do comportamento do modelo e das escolhas arquiteturais que determinam seu desempenho.

O achado mais significativo dos estudos de ablação é a dependência crítica do bloco *SOFT_COMPLEXITY*, com o modelo perdendo 28,2 pp em acurácia, a perda mais significativa comparada a qualquer outro componente. Esse resultado revela que *BibliotecaAtomica* acumula conhecimento estatístico ao longo de todo o conjunto de dados para cada átomo. Na prática, antes mesmo que a rede neural processe a topologia do grafo, os nós já carregam uma distribuição de probabilidade sobre as oito classes inferidas de todas as ocorrências anteriores daquele tipo de átomo. As camadas de passagem refinam e contextualizam esse sinal, mas não o constroem do zero e a capacidade preditiva do modelo reside na combinação dele com o pré-processamento realizado pela biblioteca.

O desempenho por classe reforça essa interpretação: a classe $O(n^3)$ obteve o maior F1 com 85,6 pp enquanto a $(O(\log n))$ apresentou 40,5 pp. Anotações estruturais como três laços aninhados são ao mesmo tempo similares e frequentes o suficiente para que a biblioteca acumule votos confiáveis, enquanto anotações de fluxo presente são pobremente encontradas e avaliadas. O sinal discriminativo de um módulo logarítmico e um linear não se manifesta na topologia do grafo sem análise semântica das variáveis de controle, departamento em que essa arquitetura vacila. Essa conclusão

denota-se também pelos resultados de ablação de arestas, no qual as estruturas de fluxo de controle concentram o sinal de complexidade.

A rede neural de grafo com isomorfismo supera as outras formas de camada de mensagem, apresentando desempenho equivalente ao teste Weisfeiler-Lehman e também distinguindo multiconjuntos de vizinhos com maior clareza (XU *et al.*, 2019). A combinação desse achado com o das arestas demonstra que o problema de classificação de complexidade também depende do reconhecimento de padrões de aninhamento.

A diferença entre o desempenho entre o modelo treinado apenas sobre o *GeeksForGeeks* e o modelo completo indicia que o volume e a diversidade dos dados têm impacto significativo sobre a qualidade das representações aprendidas pelas camadas de passagem de mensagens. Isso é esperado em GNN's, cujos parâmetros precisam generalizar sobre distribuições de grafo variadas. A adição do *CodeComplex* não apenas aumenta o volume de exemplos, mas diversifica as distribuições estruturais dentro de cada classe.

A comparação com trabalhos anteriores expõe uma contradição entre generalidade e desempenho. O *Random Forest* de Pfitscher *et al.* (2023) e o AutoGluon de Rodenbusch (2023) obtêm métricas superiores porque operam sobre características diretas das classes alvo, enquanto o modelo GNN necessita inferir esses padrões a partir da topologia do grafo sem acesso explícito a essas contagens. Em contrapartida, o modelo é capaz de operar sobre oito linguagens diferentes em comparação a seus pares que operam apenas sobre Java. Além disso, o *Random Forest* de Pfitscher *et al.* (2023) demonstrou redução de 36 pp ao ser avaliado fora de seu domínio de treinamento, evidenciado superajuste às características manuais de um repositório específico.

Por fim, os resultados da ablação de regularização indicam que a função de perda ordinal com peso $w = 0,4$ melhora o desempenho de validação em até 2 pp. Esse ganho apresenta coerência com a estrutura do problema, já que as oito classes formam uma escala ordinal em que erros de maior distância são mais graves do que entre classes vizinhas. A penalidade ordinal direciona o gradiente de modo a reduzir proporcionalmente os erros mais distantes, o que é especialmente relevante para as classes com exemplos insuficientes, tornando a calibração nessas regiões insatisfatórias.

5 CONCLUSÃO

Este trabalho investigou a viabilidade de classificação automática de complexidade de tempo de algoritmos por meio de redes neurais de grafo. O objetivo geral foi atingido, tendo sido desenvolvido, treinado e avaliado um modelo capaz de classificar funções em oito classes de complexidade estabelecidas. As seções seguintes disseram sobre o atendimento aos objetivos, as contribuições do trabalho, suas limitações e as direções identificadas para desenvolvimento futuro.

5.1 ATENDIMENTO AOS OBJETIVOS ESPECÍFICOS

Os cinco objetivos específicos definidos na introdução foram integralmente atendidos ao longo do desenvolvimento do trabalho

O primeiro objetivo, examinar diferentes representações de código-fonte para grafos, foi cumprido pela implementação de um processo de dez leitores que constrói múltiplas representações, sendo elas grafo de dependência estrutural derivado da AST, grafo de fluxo de controle, grafo de dependência de dados, relações de dominância e subgrafos de comunidades. A análise de ablação evidenciou que as arestas de fluxo de controle carregam o sinal mais discriminativo.

O segundo objetivo, tratar conjuntos de dados públicos para treinamento e validação, foi atendido pela integração dos conjuntos *GeeksForGeeks* (RODENBUSCH, 2023) e *CodeComplex* (BAIK *et al.*, 2024), com desenvolvimento de módulos de conversão, normalização de rótulos e mapeamento manual supervisionado, produzindo 37 608 exemplos válidos em oito linguagens.

O terceiro objetivo, empregar modelos GNN sobre os dados tratados, foi cumprido pela implementação da arquitetura *ComplexityGNN*, que utiliza modelos GIN, GCN, GraphSAGE e GAT (XU *et al.*, 2019; KIPF; WELLING, 2017; HAMILTON *et al.*, 2017; VELIČKOVIĆ *et al.*, 2017).

O quarto objetivo, avaliar experimentalmente o desempenho, foi atendido pelo conjunto de métricas reportado no Capítulo 4. O modelo de referência atingiu acurácia exata de 67,6%, e AUC-ROC de 0,926 para a classificação binária de eficiência, com Cohen's κ de 0,595 indicando concordância moderada (LANDIS; KOCH, 1977). Os estudos de ablação revelaram que o bloco *SOFT_COMPLEXITY* é o componente mais crítico da arquitetura, com redução de 27,9pp ao ser removido. Os modelos empregados tiveram todos desempenho moderado, sendo o modelo GIN o que se sobressaiu, mas com baixa diferença.

O quinto objetivo, comparar os resultados com modelos existentes, foi cumprido pela análise da Seção 4.4. O modelo proposto obtém desempenho inferior ao

Random Forest de Pfitscher *et al.* (2023) (–22,2 pp) e ao *AutoGluon* de Rodenbusch (2023) (–10,0 pp) na acurácia de classe, mas opera sobre oito linguagens sem engenharia manual de características, condição metodologicamente mais rigorosa que os trabalhos comparados.

5.2 CONTRIBUIÇÕES

O trabalho produziu contribuições tanto no plano metodológico quanto no plano técnico, descritas a seguir.

A principal contribuição metodológica é a demonstração de que a representação de código-fonte como grafo de dependências, combinada com uma rede neural de grafos sem características manuais, produz um classificador de complexidade funcional e extensível a novas linguagens. Não foi encontrado correspondente direto à biblioteca de topologias moleculares na literatura consultada, constituindo a principal novidade arquitetural deste trabalho.

A análise de ablação conduzida contribui com evidência empírica sobre a importância relativa de cada componente: a função de soma da GIN, o mecanismo de Jumping Knowledge, o agrupamento combinado e a penalidade ordinal na função de perda são identificados como as decisões de maior impacto positivo sobre o desempenho.

Do ponto de vista técnico, a arquitetura *ComplexityGNN* foi projetada com modularidade como princípio central. Cada componente é configurável independentemente via *GNNConfig*, sem exigir alterações no restante do processo. Essa propriedade reduz substancialmente o custo de investigação de novas hipóteses: adicionar uma variante de camada, um novo tipo de aresta ou uma função de regularização alternativa requer modificações localizadas e bem delimitadas, tornando o sistema adequado para uso como base de pesquisa em trabalhos futuros.

5.3 LIMITAÇÕES

Apesar dos resultados obtidos, o trabalho apresenta limitações que devem ser consideradas na interpretação dos resultados e na definição de próximos passos.

O modelo apresenta dificuldade estrutural para classificar $O(\log n)$ ($F1 = 0,405$), classe cujo padrão característico não se manifesta na topologia do grafo sem análise semântica das variáveis de controle. As classes $O(2^n)$ e $O(n!)$ possuem suporte insuficiente no conjunto de dados para conclusões robustas. O processo de leitura apresentou comportamento exponencial nos passes de segmentação e detecção de comunidades, impondo restrições à escalabilidade do método. O rótulo de complexidade proveniente do arquivo para todas as suas funções constitui uma fonte de ruído

para funções secundárias com comportamento assintótico distinto do arquivo de origem.

Os estudos de ablação evidenciaram que os leitores de arestas semântica, como dependência de dados e dominância, não produziram ganho mensurável de desempenho nas configurações. Esse fator e o tempo curto de processamento dos leitores dessas características sugere que esses leitores podem não estar operando de forma plena. Apenas as arestas estruturais e de fluxo de controle contribuíram com sinal discriminativo efetivo para o modelo atual.

5.4 TRABALHOS FUTUROS

Durante o desenvolvimento desse trabalho, percebeu-se setores que poderiam ser aprimorados e atividades desenvolvidas. Graças a modularidade do trabalho, há um conjunto de direções objetivas e de fácil nível arquitetural.

Cabe avaliar o desempenho da arquitetura quando treinada apenas sobre um dos conjuntos e testado sobre o outro, permitindo caracterizar com maior exatidão a capacidade de generalização do modelo, em condição análoga ao experimento conduzido por Rodenbusch (2023) para o *Random Forest*. Nesse contexto, a adoção de aprendizado contrastivo, estratégia proposta por Park *et al.* (2025), é uma direção promissora para melhorar a separabilidade das classes mais complexas.

A reformulação dos leitores de arestas semânticas, com enfoque nos passes de dependência de dados e de dominância, é outra demanda. Os resultados do estudo ablativo da arquitetura indicam que esses leitores não estão contribuindo com sinal útil ao modelo, o que indica possível falha na extração ou na codificação das arestas. Uma reimplementação mais rigorosa desses passes, com validação sobre exemplos controlados, poderia ser capaz de extrair dados úteis para a classificação de complexidade.

A implementação de propriedades globais de grafos como características adicionais, como coeficiente de agrupamento, diâmetro, distribuição de grau e densidade, representam outra direção complementar que poderia enriquecer a representação do grafo sem alterar a arquitetura de passagem de mensagens. Essas métricas capturam aspectos estruturais globais que as operações de agrupamento não necessariamente preservam, apesar de que como mostrado. Esse projeto seria de menor prioridade, dado que o fluxo estrutural foi mostrado ser pouco contribuinte para o desempenho.

A substituição das bibliotecas de aprendizado profundo, como *Pytorch* e *Net-workX*, por uma implementação própria e personalizada para esse tópico das operações de passagem de mensagens e agrupamento, permitiria maior controle sobre o processo de inferência, redução de dependências e potencial ganho de desempenho. Essa refatoração também contribuiria para uma compreensão mais aprofundada dos mecanismos internos do modelo.

Por fim, a exposição do modelo como uma API pública constituiria a etapa de consolidação do trabalho em uma ferramenta utilizável por desenvolvedores. A aplicação receberia códigos-fonte, retornando a classe de complexidade prevista ou até poderia atuar em tempo real sobre funções e algoritmos em desenvolvimento.

REFERÊNCIAS

AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. **Compilers: Principles, Techniques, and Tools**. 2. ed. Boston: Pearson Addison Wesley, 2006.

BAIK, S.-Y. *et al.* **CodeComplex: Dataset for Worst-Case Time Complexity Prediction**. 2024. ArXiv preprint arXiv:2401.08719. Publicado em: Findings of ACL: EMNLP 2025, pp. 19616–19638. Disponível em: <https://arxiv.org/abs/2401.08719>. Acesso em: 22 mar. 2025.

BIELIK, P.; RAYCHEV, V.; VECHEV, M. PHOG: Probabilistic Model for Code. *In: Proceedings of Machine Learning Research (PMLR)*. [S.l.: s.n.], 2016. p. 2933–2942.

BISHOP, C. M. **Pattern Recognition and Machine Learning**. New York: Springer, 2006. (Information Science and Statistics). ISBN 978-0-387-31073-2.

BLOCH, T.; BORRMANN, A.; PAUWELS, P. Graph-based learning for automated code checking – Exploring the application of graph neural networks for design review. **Advanced Engineering Informatics**, v. 58, p. 102137, 2023.

BLONDEL, V. D.; GUILLAUME, J.-L.; LAMBIOTTE, R.; LEFEBVRE, E. Fast unfolding of communities in large networks. **Journal of Statistical Mechanics: Theory and Experiment**, v. 2008, n. 10, p. P10008, 2008.

BREIMAN, L. Random Forests. **Machine Learning**, v. 45, n. 1, p. 5–32, 2001.

BRUNSFELD, M. *et al.* **Tree-sitter: An incremental parsing system for programming tools**. 2025. Disponível em: <https://tree-sitter.github.io/tree-sitter/>.

BUTEREZ, D.; JANET, J. P.; KIDDLE, S. J.; OGLIC, D.; LIÒ, P. Graph Neural Networks with Adaptive Readouts. *In: Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*. Curran Associates, Inc., 2022. p. 19746–19758. Disponível em: https://proceedings.neurips.cc/paper_files/paper/2022/file/7caf9d251b546bc78078b35b4a6f3b7e-Paper-Conference.pdf.

CHEN, Y. *et al.* **Semi-Supervised and Unsupervised Deep Visual Learning: A Survey**. 2022. ArXiv preprint arXiv:2208.11296. Disponível em: <https://arxiv.org/abs/2208.11296>. Acesso em: 21 mar. 2025.

COOPER, K. D.; HARVEY, T. J.; KENNEDY, K. A simple, fast dominance algorithm. **Software Practice and Experience**, v. 4, p. 1–10, 2001.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms**. 3. ed. [S.l.]: MIT Press, 2009. Versão em português: Algoritmos: Teoria e Prática.

DAIGAVANE, A. *et al.* **Node-Level Differentially Private Graph Neural Networks**. 2021. ArXiv preprint arXiv:2111.15521. Disponível em: <https://arxiv.org/abs/2111.15521>. Acesso em: 15 mar. 2025.

- FAWCETT, T. An introduction to ROC analysis. **Pattern Recognition Letters**, v. 27, n. 8, p. 861–874, 2006.
- FERRANTE, J.; OTTENSTEIN, K. J.; WARREN, J. D. The Program Dependence Graph and Its Use in Optimization. **ACM Transactions on Programming Languages and Systems**, v. 9, n. 3, p. 319–349, 1987.
- FEY, M.; LENSSEN, J. E. Fast Graph Representation Learning with PyTorch Geometric. *In: ICLR Workshop on Representation Learning on Graphs and Manifolds*. [S.l.: s.n.], 2019.
- GAO, Y.; GU, X.; ZHANG, H. *et al.* Runtime Performance Prediction for Deep Learning Models with Graph Neural Network. *In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2023. p. 368–380.
- GILMER, J.; SCHOENHOLZ, S. S.; RILEY, P. F.; VINYALS, O.; DAHL, G. E. Neural Message Passing for Quantum Chemistry. *In: Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*. [S.l.]: PMLR, 2017. p. 1263–1272.
- GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016. Disponível em: <https://www.deeplearningbook.org/>.
- HAGBERG, A. A.; SCHULT, D. A.; SWART, P. J. Exploring Network Structure, Dynamics, and Function using NetworkX. *In: Proceedings of the 7th Python in Science Conference*. Pasadena, CA USA: [s.n.], 2008. p. 11–15.
- HAMILTON, W. L.; YING, R.; LESKOVEC, J. Inductive Representation Learning on Large Graphs. *In: Advances in Neural Information Processing Systems 30 (NIPS 2017)*. [S.l.]: Curran Associates, Inc., 2017. p. 1024–1034.
- HIN, D.; KAN, A.; CHEN, H.; BABAR, M. A. LineVD: Statement-level vulnerability detection using graph neural networks. *In: Proceedings of the 19th International Conference on Mining Software Repositories*. Pittsburgh, PA, USA: ACM, 2022. p. 596–607.
- HOCHREITER, S.; SCHMIDHUBER, J. Long Short-Term Memory. **Neural Computation**, v. 9, n. 8, p. 1735–1780, 1997.
- International Union of Pure and Applied Chemistry. **cis-trans isomers**. 2025. *In: IUPAC Compendium of Chemical Terminology (Gold Book)*, 5th ed. Disponível em: <https://goldbook.iupac.org/terms/view/C01093>. Acesso em: 22 mar. 2025.
- IOFFE, S.; SZEGEDY, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *In: International Conference on Machine Learning (ICML 2015)*. [S.l.]: PMLR, 2015. p. 448–456.
- JANIKOW, C. Z. A knowledge-intensive genetic algorithm for supervised learning. **Machine Learning**, v. 13, n. 2-3, p. 189–228, 1994.
- KIPF, T. N.; WELLING, M. Semi-Supervised Classification with Graph Convolutional Networks. *In: 5th International Conference on Learning Representations (ICLR 2017)*. Toulon, France: [s.n.], 2017. Disponível em: <https://openreview.net/forum?id=SJU4ayYgl>.

KLEINBERG, J.; TARDOS, E. **Algorithm design**. [S.l.]: Pearson Education India, 2006.

KULLBACK, S.; LEIBLER, R. A. On Information and Sufficiency. **The Annals of Mathematical Statistics**, JSTOR, v. 22, n. 1, p. 79–86, 1951.

LANDIS, J. R.; KOCH, G. G. The measurement of observer agreement for categorical data. **Biometrics**, v. 33, n. 1, p. 159–174, 1977.

LOSHCHILOV, I.; HUTTER, F. Decoupled weight decay regularization. **International Conference on Learning Representations**, 2019. Disponível em: <https://openreview.net/forum?id=Bkg6RiCqY7>.

LUCAS, S. The origins of the halting problem. **Journal of Logical and Algebraic Methods in Programming**, v. 121, p. 100687, 2021.

MCCABE, T. J. A Complexity Measure. **IEEE Transactions on Software Engineering**, SE-2, n. 4, p. 308–320, 1976. Disponível em: <https://ieeexplore.ieee.org/document/1702388>. Acesso em: 4 jul. 2025.

MCCULLOCH, W. S.; PITTS, W. A Logical Calculus of the Ideas Immanent in Nervous Activity. **Bulletin of Mathematical Biophysics**, v. 5, p. 115–133, 1943.

MITCHELL, T. M. **Machine Learning**. [S.l.]: McGraw-Hill, 1997. Disponível em: <http://www.cs.cmu.edu/~tom/mlbook.html>.

NOGARA, V. G. A. **ComplexityGNN: análise de complexidade de códigos com rede neural de grafos**. 2026. Disponível em: <https://codigos.ufsc.br/vitor.nogara/gnncomplexity>. Acesso em: 12 jul. 2026.

PARK, S.; HAHN, J.; ORWIG, E.; KO, S.-K.; HAN, Y.-S. Advanced code time complexity prediction approach using contrastive learning. **Engineering Applications of Artificial Intelligence**, v. 151, p. 110631, 2025. ISSN 0952-1976.

PASZKE, A. *et al.* PyTorch: An Imperative Style, High-Performance Deep Learning Library. *In: Advances in Neural Information Processing Systems 32*. [S.l.]: Curran Associates, Inc., 2019. p. 8024–8035.

PFITSCHER, R. J.; RODENBUSCH, G.; DIAS, A.; VIEIRA, P.; FOUTO, N. Estimating Code Running Time Complexity with Machine Learning. *In: Brazilian Conference on Intelligent Systems (BRACIS 2023)*. [S.l.]: Springer, 2023. (Lecture Notes in Computer Science), p. 400–414.

Python Software Foundation. **ast — Abstract Syntax Trees**. [S.l.], 2025. Acesso em: 29 mar. 2025. Disponível em: <https://docs.python.org/3/library/ast.html>.

PyTorch Contributors. **torch.optim — PyTorch Documentation**. [S.l.], 2025. Acesso em: 29 mar. 2025. Disponível em: <https://pytorch.org/docs/stable/optim.html>.

RODENBUSCH, G. B. **Desenvolvimento de API para Predição da Complexidade de Tempo de Execução de Códigos por Meio de AutoML**. 2023. Monografia (Trabalho de Conclusão de Curso (Bacharel em Engenharia Mecatrônica)) — Universidade Federal de Santa Catarina, Centro Tecnológico de Joinville, Joinville, 2023.

- ROSENBLATT, F. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. **Psychological Review**, v. 65, n. 6, p. 386–408, 1958.
- ROSENDAHL, M. **Automatic complexity analysis**. 1989. Tese (Doutorado) — University of Copenhagen, Department of Computer Science, 1989.
- RUSSELL, S. J.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. 3. ed. [S.l.]: Prentice Hall, 2010. Disponível em: <https://people.engr.tamu.edu/guni/csce625/slides/AI.pdf>.
- SAMUEL, A. L. Some Studies in Machine Learning Using the Game of Checkers. **IBM Journal of Research and Development**, v. 3, n. 3, p. 210–229, jul 1959.
- SEEDGEWICK, R.; WAYNE, K. **Algorithms**. 4. ed. Upper Saddle River, NJ: Addison-Wesley Professional, 2011.
- SIKIC, L. *et al.* Graph Neural Network for Source Code Defect Prediction. **IEEE Access**, v. 10, p. 10402–10415, 2022.
- SIKKA, J. *et al.* Learning Based Methods for Code Runtime Complexity Prediction. *In: Advances in Information Retrieval. ECIR 2020. Lecture Notes in Computer Science*. Cham: Springer, 2020. v. 12035, p. 313–325.
- Stanford Institute for Human-Centered AI. **The 2024 AI Index Report**. [S.l.], 2024. Disponível em: <https://hai.stanford.edu/ai-index/2024-ai-index-report>.
- TAN, P.-N.; STEINBACH, M.; KUMAR, V. **Introduction to Data Mining**. [S.l.]: Pearson/Addison-Wesley, 2005. Disponível em: https://www.ceom.ou.edu/media/docs/upload/Pang-Ning_Tan_Michael_Steinbach_Vipin_Kumar_-_Introduction_to_Data_Mining-Pe_NRDk4fi.pdf.
- THIEMEL, C. *et al.* **javalang: A pure Python library for parsing Java**. GitHub, 2025. Disponível em: <https://github.com/c2nes/javalang>.
- TRINAJSTIĆ, N. **Chemical Graph Theory**. 2. ed. Boca Raton: CRC Press, 1992.
- TURING, A. M. Computing Machinery and Intelligence. **Mind**, v. 59, n. 236, p. 433–460, 1950. Disponível em: <http://www.jstor.org/stable/2251299>. Acesso em: 22 mar. 2025.
- VELIČKOVIĆ, P. *et al.* **Graph Attention Networks**. 2017. ArXiv preprint arXiv:1710.10903.
- WADE, L. G. **Química Orgânica**. 7. ed. São Paulo: Pearson Prentice Hall, 2012.
- WU, C. **Graph Representation Learning: from Kernel to Neural Networks**. 2021. Tese (Doutorado) — Institut Polytechnique de Paris, 2021. Disponível em: <https://tel.archives-ouvertes.fr/tel-03662478>. Acesso em: 22 mar. 2025.
- XU, K.; HU, W.; LESKOVEC, J.; JEGELKA, S. How Powerful are Graph Neural Networks? *In: 7th International Conference on Learning Representations (ICLR 2019)*. [s.n.], 2019. Disponível em: <https://openreview.net/forum?id=ryGs6iA5Km>.

XU, K. *et al.* Representation Learning on Graphs with Jumping Knowledge Networks. *In: Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*. [S.l.]: PMLR, 2018. v. 80, p. 5449–5462.

YU, Q.; LIU, X.; ZHOU, Q. *et al.* Code classification with graph neural networks: Have you ever struggled to make it work? **Expert Systems with Applications**, v. 233, p. 120978, 2023.

ZHOU, J. *et al.* Graph neural networks: A review of methods and applications. **AI Open**, v. 1, p. 57–81, 2020.