Relatório final POC II
Utilizando Representações de Incorporação Semântica e Sintática para Aprimorar a Análise de Árvores de Programação Genética

Diogo Neiss
DCC UFMG
Aluno
Belo Horizonte, Brasil
diogoneiss@dcc.ufmg.br

Gisele Pappa
DCC UFMG
Orientadora
Belo Horizonte, Brasil
glpappa@dcc.ufmg.br

***Resumo*—Este relatório abordou técnicas avançadas em aprendizado de máquina, focando em programação genética e redes neurais para grafos, com ênfase em um artigo na área publicado recentemente. Reproduzimos as técnicas descritas, identificando problemas e melhorias, gerando avanços nos resultados obtidos anteriormente. Descobrimos que, apesar dos desafios em complexidade computacional e representação de dados, estas técnicas têm potencial para resolver certos problemas. Destacamos as redes neurais hiperbólicas pela sua eficácia em representar dados hierárquicos e grafos e *large language models*. Concluímos que a continuação da pesquisa pode trazer avanços significativos no aprendizado de máquina e modelagem de dados complexos, principalmente para problemas de otimização.**

***Palavras chave*—Aprendizado de Máquina, Redes Neurais, Transformers, Grafos, Algoritmos Genéticos, Programação Genética, Regressão Simbólica**

I. Introdução

Este trabalho se propõe a continuar a pesquisa desenvolvida em [1], visando explorar melhor as descobertas, gerar visualizações e aperfeiçoar representações, melhorando as técnicas e estendendo-as para inclusão de atributos semânticos do problema. Essas linhas de pesquisa não foram completamente exploradas no artigo, portanto planejamos estudá-las e documentar nossos resultados e descobertas. Em seguida, tentaremos estender esse modelo para suportar atributos semânticos da representação.

Antes de detalhar o objeto de estudo desse projeto, vamos introduzir alguns termos:

1) **Regressão Simbólica:** É capaz de encontrar expressões matemáticas de qualquer forma, podendo até incluir operadores não-lineares. É comumente representada por uma árvore de expressões.
2) **Programação Genética:** Técnica de otimização bioinspirada que emula a evolução natural para encontrar soluções para problemas complexos
3) **Espaços de *Embedding*:** Permitem transformar dados complexos em espaços vetoriais de menor dimensão, preservando as relações estruturais essenciais entre os pontos. Muito usada para converter texto em representações numéricas.
4) **Arquiteturas *Encoder-Decoder*:** As arquiteturas *encoder-decoder* são uma classe de modelos que primeiramente convertem uma entrada em uma representação latente através de uma operação de codificação (*encoder*), que também pode resultar em um espaço de *embedding* e, depois, reconstroem alguma forma de saída a partir dessa representação, decodificando-a (*decoder*).
5) **Redes Neurais em Grafos:** As redes neurais em grafos são a forma mais abstrata de aprendizado de máquina, devido à ausência de restrições estruturais, como dados tabulares, texto, imagens, de forma que possam trabalhar com dados em grafos, capturando padrões e características nos dados.

A. Aplicações e Motivação

Uma necessidade de vários algoritmos de programação genética é a representação. Ao trabalhar com árvores ou grafos, desejamos ter alguma estrutura capaz de expressar os atributos estruturais e semânticos e ainda ter expressividade para outras operações úteis, como semelhança ou distância. Para entender esse problema, vamos ver um exemplo de árvore representando uma regressão simbólica de uma função arbitrária. Note que essas árvores que a programação genética usa para representar regressões simbólicas se assemelham bastante com árvores sintáticas abstratas utilizadas em compiladores. Essa estrutura se mostra eficiente, permitindo a criação expressões aninhadas, como o caso do log com outra expressão dentro.

Como podemos observar nas figuras a seguir, as duas árvores são sintaticamente diferentes, já que a adição teve sua ordem invertida, mas o significado semântico se manteve, uma vez que a adição é comutativa. Se desejarmos verificar semelhança sintática entre árvores isomorfas com uma métrica de distância de edição, essa verificação fica extremamente

custosa conforme as arvores aumentam e sua linearização também.

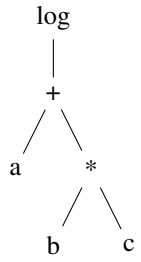

Fig. 1. $\log(a + (b \cdot c))$

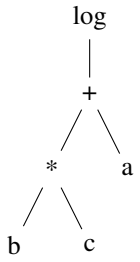

Fig. 2. $\log((b \cdot c) + a)$

Ter uma representação focada na vizinhança de cada vértice da árvore facilitaria bastante essa operação, já que a estrutura inteira da árvore não seria mais utilizada como base. Vamos imaginar que essa foi a codificação de cada uma das árvores, com a única diferença sendo o último valor de cada, que difere em 0.1

$$\vec{e_1} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \quad \vec{e_2} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4.1 \end{bmatrix}$$

Ao assumir um espaço euclidiano e realizar operações de semelhança cosseno e distância euclidiana entre os dois, teríamos uma distância ≈ 0 e similaridade ≈ 1, ou seja, rapidamente podemos verificar o quão próximas são sintaticamente. Algumas estratégias de evolução incluem estímulo a diversidade de indivíduos na população, como no caso de *fitness sharing*, que muitas vezes leva em conta genótipo e fenótipo (sintaxe e semântica).

Entretanto, se faz necessário analisar mais que a estrutura da solução, já que temos um problema de otimização a ser resolvido: precisamos calcular qual o erro de cada programa e apenas a estrutura da árvore não nos dá essa informação, sendo necessário verificar também a semântica do problema.

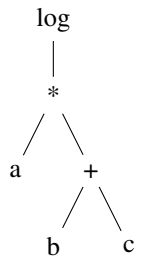

Fig. 3. $\log(a \cdot (b + c))$

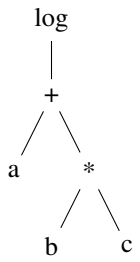

Fig. 4. $\log(a + (b \cdot c))$

Nesse outro exemplo dois vértices tiveram as operações alteradas. Para observar similaridade semântica, podemos criar um conjunto de dados que ao fazer o cálculo do desvio quadrático médio dos erros dessas duas árvores elas teriam erros próximos, manifestando semelhança semântica. Com isso, notamos que é benéfico para estratégias de diversidade levar os dois lados em consideração, utilizando uma forma de representação rápida capaz de codificar distância sintática.

Outra aplicação interessante é a visualização de modelos ao longo do tempo. Outras heurísticas, como *particle swarm optimization*, permitem analisar e visualizar a movimentação das soluções candidatas ao longo do tempo, por serem inerentemente vetoriais, facilitando o ajuste de parâmetros, algo que não é trivial de ser feito com programações genéticas, que requerem muitas vezes análises numéricas dos resultados de várias execuções com parâmetros variados para se chegar a alguma conclusão. Tendo um método confiável de representação, podemos utilizar esses vetores para estudar melhor o problema a ser otimizado, seu espaço de soluções e o impacto de parâmetros, com algoritmos podendo ser executados para automatizar e acelerar essa tarefa

Por fim, um problema conhecido em programações genéticas é a localidade, isto é, como mudanças na sintaxe impactam a semântica. Com o auxílio de representações em menos dimensões poderíamos, por exemplo, ter uma noção de vizinhança no espaço de soluções e realizar alterações no espaço de busca de acordo com isso, controlando a variância de soluções se desejado.

B. Objetivos Gerais

Com uma compreensão melhor dos pré-requisitos, podemos definir então os objetivos gerais desse trabalho como sendo o aperfeiçoamento/estudo de técnicas de *embedding* de programações genéticas e tópicos associados, de acordo com uma lista de linhas possíveis de aprofundamento.

No primeiro semestre (POC 1), focamos nas seguintes tarefas ao longo das 16 semanas:

a. Estudo de ajustes na linearização da árvore para otimização do *encoder*
b. Aperfeiçoamento das técnicas de treinamento do modelo atual com variação de parâmetros, ajuste de entrada e aumento da base de treinamento
c. Análise de similaridade entre representações de árvores isomorfas e pseudo isomorfas
d. Análise de similaridade entre representações com redundâncias
e. Inclusão de técnicas de redes neurais em grafos para codificação das árvores
f. Análise da evolução de indivíduos no espaço de *embeddings* por geração

Para o segundo semestre (POC II), os desenvolvimentos foram:

i. Identificação de melhorias na representação dos dados de entrada do *transformer*
ii. Investigação de viés e *leakage* nos dados de treino e teste
iii. Aperfeiçoamento da técnica de geração de dados sintética
iv. Concepção de um pipeline de geração de dados, limpeza, treino, coleta de métricas e análise de artefatos com *frameworks* adequados
v. Análise dos resultados e ajustes na experimentação

vi. Execução do *pipeline* com os parâmetros desejados
vii. Agregação de métricas e elaboração do relatório final

Alguns tópicos mostraram novas linhas de pesquisa e melhorias, divergindo o projeto da expectativa de tarefas inicial e avançando em áreas inesperadas.

O resultado final do trabalho ao final da disciplina de POC II foi uma pesquisa sólida sobre as ramificações dos estudos e análises descritos anteriormente, gerando experimentos e dados para publicações futuras e continuação em meu mestrado.

II. Revisão teórica

A. Regressão Simbólica

A regressão simbólica busca modelar relações funcionais entre variáveis através da identificação de equações simbólicas. Ao contrário das técnicas de regressão tradicionais, a regressão simbólica tem a capacidade de encontrar expressões matemáticas de qualquer forma, de acordo com um conjunto de operadores matemáticos base e restrições impostas, porém sofre de um custo computacional considerável, especialmente em cenários onde a dimensionalidade dos dados é alta ou a estrutura da relação subjacente é particularmente complexa. A capacidade de descobrir estruturas de relações matemáticas não-lineares e intrincadas a torna uma ferramenta indispensável para modelar fenômenos complexos em diversas áreas, quando o custo elevado de treinamento compensa a redução de erro do modelo.

Formalmente, consiste em encontrar uma função matemática arbitrária a partir de um conjunto de entradas e saídas de acordo com um conjunto de operadores definido. Ela possui expressividade maior que as regressões comuns, como a linear, que modela apenas funções lineares $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$, e também da regressão polinomial, que segue a forma $y = \beta_0 + \beta_1 x + \beta_2 x^2 + \ldots + \beta_n x^n$, justamente por possibilitar a escolha de operadores do problema de acordo com conhecimentos da forma dos dados, então se fosse conhecida a presença de operadores trigonométricos em algum lugar da função, poderíamos fazer uma regressão linear estendida, incluindo sin e cos, aumentando consideravelmente a capacidade de representação do problema.

Definição II.1 (Regressão Simbólica)**.** Seja $\mathcal{X} \subset \mathbb{R}^n$ o espaço de entrada, onde n é o número de variáveis que caracterizam um exemplo de dados. O objetivo da regressão simbólica é encontrar uma função $f : \mathcal{X} \to \mathbb{R}$ tal que $f(\mathbf{x}) \approx y$, para cada $(\mathbf{x}, y)$ em um conjunto de dados $D \subset \mathcal{X} \times \mathbb{R}$, minimizando uma métrica de erro E, que mede a discrepância entre os valores preditos $f(\mathbf{x})$ e os valores verdadeiros y em D e utilizando um conjunto de operadores matemáticos predefinidos.

B. Programação Genética e Algoritmos Evolutivos

É um método de otimização heurística bioinspirada que emula a evolução natural para encontrar soluções para problemas complexos sem instruir diretamente o computador como fazê-lo. É frequentemente utilizado para criar uma regressão simbólica, em que uma população de "programas" é evoluída e uma otimização é feita, minimizando o erro, mas também pode ser feita para criar códigos fontes de linguagem de programação como *LISP* e *C*. Este conceito foi explorado em profundidade em [2].

- **Genótipo e Fenótipo:**
  - *Genótipo:* Refere-se à representação codificada de uma solução no espaço de busca. Na PG, os genótipos podem ser representados de várias formas, como árvores, grafos, ou sequências de código. Também pode ser visto como a representação sintática.
  - *Fenótipo:* É a manifestação concreta do genótipo, ou seja, a solução específica que o genótipo representa. O mapeamento entre genótipo e fenótipo pode ser simples ou complexo, dependendo do problema em mãos, e pode ser interpretado como o aspecto semântico do problema. No caso de uma programação genética, o fenótipo poderia ser gerar a expressão matemática codificada no genótipo e calcular o erro entre o resultado real, conhecido do conjunto de dados, e o valor gerado pela substituição de variáveis na expressão.
- **Função de Aptidão (Fitness):** É uma métrica que avalia a qualidade de uma solução individual (fenótipo). Serve para guiar o processo de seleção, favorecendo os indivíduos que apresentam melhor aptidão, semelhante ao conceito darwinista de seleçao natural. Pode ser alterada de acordo com a natureza do programa, com um bom exemplo de função de fitness sendo o desvio quadrático médio (RMSE) entre os valores gerados pelo programa e o valor real.
  Para entender melhor esse exemplo, vamos imaginar que queremos encontrar a função $f(x) = 2x_1 + 3x_2 - 3$. Ainda não sabemos que essa é de fato a função, temos apenas um conjunto de dados que nos possibilitaria aproximá-la. Imagine então que chegamos no genótipo $f(x) = 3x_1 + 2x_2 - 2$, que se parece bastante com a original. Calcularemos então os valores gerados pela função (fenótipo) e compararemos com os valores de x e y_{fenotipo} dados no conjunto de treinamento

x_1	x_2	y_{real}	y_{fenotipo}	$y_f - y_r$
1	2	5	7	2
2	4	13	12	-1
3	3	12	13	1

  Analisando as diferenças, poderíamos calcular a soma de termos quadrados, que será $4+1+1$, dividir pelo total de itens, e fazer a raiz quadrada, chegando no RMSE através da seguinte conta $\sqrt{\frac{2^2+-1^2+1^2}{3}} = \sqrt{2}$, que representa o quanto nosso fenótipo se aproxima da resposta do problema desejado.
- **Mutação:** É operador genético que faz modificações aleatórias em um indivíduo, permitindo a exploração de novas áreas no espaço de busca. O processo pode envolver a alteração, adição ou remoção de nós em uma árvore, por exemplo.

- **Crossover (Recombinação):** O crossover é um operador genético que combina partes de dois indivíduos pais para gerar um ou mais descendentes. Ele promove a troca de informações genéticas entre os indivíduos, facilitando a convergência para soluções ótimas.
- **Técnicas de Seleção:** São estratégias para escolher os indivíduos que serão pais da próxima geração, removendo indivíduos indesejados. Algumas técnicas populares são por torneio, em que os indivíduos são escolhidos para um ”torneio” de tamanho fixo e o vencedor (o de melhor fitness) é selecionado. Outra técnica é Seleção Proporcional à Aptidão, em que os indivíduos são selecionados com uma probabilidade proporcional ao seu valor de fitness. Um fator de escolha para as técnicas é a pressão seletiva, que impacta a velocidade de convergência do algoritmo. Se escolhermos uma lógica de seleção baseada em fitness comparativo, podemos sofrer de convergência prematura e não sair de mínimos locais, então é importante sempre incluir um elemento estocástico de escolha de soluções potencialmente ”ruins” que podem eventualmente sofrer mutação ou cruzamento e formar soluções melhores
- **Diversidade e *fitness sharing*** Manter a diversidade genética na população é crucial para evitar a convergência prematura para ótimos locais, visto que se as soluções compartilham um nicho (uma região do espaço de busca onde as soluções são semelhantes), elas podem acabar competindo por recursos, o que pode levar a uma redução da variedade de soluções possíveis. Técnicas como a introdução de mutações, *niching* e estratégias de seleção diversificadas ajudam a manter a diversidade populacional desejada. Uma estratégia interessante de *niching* é *fitness sharing*, em que se ajusta a função de *fitness* de cada indivíduo com base na densidade de soluções no seu nicho. A ideia é dividir a ”recompensa” de *fitness* entre os indivíduos que são semelhantes.

C. Espaços de Embedding / Latentes

Os espaços de *embedding* representam uma técnica de transformação de dados complexos em espaços de menor dimensão, mantendo, no entanto, as relações estruturais essenciais entre os pontos de dados, possibilitando, por exemplo, o cálculo da semelhança entre diferentes entradas.

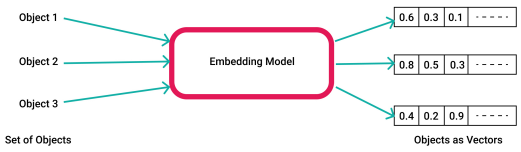

Fig. 5. Entradas e saídas de um modelo de embedding

Com um espaço de *embeddings* resultante de um *encoder*, podemos facilitar operações computacionais em grandes conjuntos de dados e ter sua visualização e análise em uma forma mais gerenciável e compreensível. Essa técnica é amplamente utilizada em arquiteturas modernas de *deep learning* aplicações populares, como o *ChatGPT*. O termo *latente* frequentemente é usado como sinônimo, porém nem sempre é um conceito intercambiável.

Diversos modelos, como *Variational autoencoders* e *Word2Vec*, aprendem a criar espaços de *embeddings* representativos da natureza do problema. Um exemplo interessante é a aritmética de representações, em que no artigo [3] foi notado que retirando ”Homem” de ”Irmão” e somando ”Mulher”, podemos derivar um vetor muito próximo de ”Irmã”, ou de produzir relações como ”Homem está para Mulher assim como Irmão está para Irmã”, em que a semântico foi capturada com sucesso na representação no espaço.

D. Arquiteturas Encoder-Decoder e Transformers

As arquiteturas *encoder-decoder*, também conhecidos como *autoencoders*, são uma classe de modelos que primeiramente convertem uma entrada em uma representação latente através de uma operação de codificação (*encoder*), que também pode resultar em um espaço de *embedding* e, depois, reconstroem alguma forma de saída a partir dessa representação, decodificando-a (*decoder*). Este tipo de arquitetura é considerado o estado da arte e está presente em diversas categorias de problemas, como geração de texto, classificação de imagens, tradução automática, sistemas de recomendação, dentre outros [4]

À medida que essa abordagem evoluiu, surgiu a arquitetura de *Transformers*, publicada em [5], que revolucionou ainda mais o campo da aprendizagem profunda. A principal inovação desta arquitetura é a introdução de mecanismos de atenção, que permitem ao modelo focar em diferentes partes da entrada para cada palavra na saída, de forma adaptativa. Este mecanismo de atenção, muitas vezes chamado de ”atenção autoregressiva” ou ”atenção *multi-headed*”, permite que o modelo pese diferentes partes da entrada de forma diferente, possibilitando aprendizagens mais complexas e nuances sobre as relações entre diferentes partes do *input*, resultanto em janelas de contexto muito maiores que outros modelos recorrentes.

Além disso, os *Transformers* eliminam a necessidade de recorrência na arquitetura, permitindo paralelização mais eficiente durante o treinamento, o que resulta em uma redução significativa do tempo de treinamento. Eles também têm a capacidade de capturar dependências de longo alcance na entrada, algo que modelos convencionais tinham imensa dificuldade devido a *vanishing gradients*. Em termos de performance, ultrapassaram consistentemente os modelos anteriores e estabeleceram novos padrões de estado da arte, mostrando que modelos capazes de implementar mecanismos de atenção são o melhor em uma ampla gama de tarefas, difundindo a arquitetura em diversas áreas além de texto, como visão computacional e aprendizado de máquina em grafos.

E. Aprendizado de máquina em Grafos e Árvores

As redes neurais em grafos representam uma extensão das redes neurais convencionais, que são especializadas na análise e modelagem de dados estruturados, como dados tabulares, texto, imagens, de forma que possam trabalhar com dados

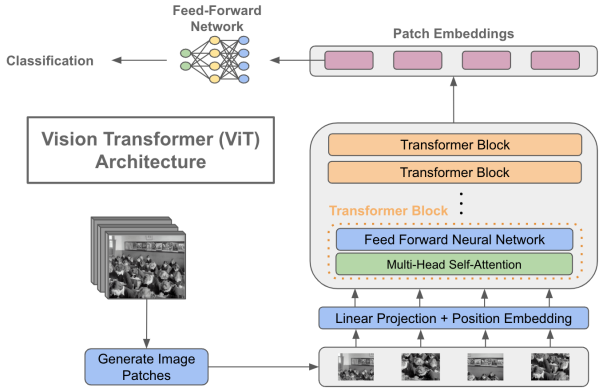

Fig. 6. Transformers aplicados a classificação de imagens

em grafos. Elas são capazes de capturar padrões e características em dados que possuem relações não-euclidianas de tamanhos e estruturas variáveis, sendo amplamente aplicadas em áreas como a bioinformática, a análise de redes sociais e a detecção de fraudes financeiras, analisando os vértices, arestas e vizinhanças para gerar suas conclusões de acordo com o modelo desejado. Seu maior problema é a representação do modelo, que precisa ser convertido em uma representação estruturada para executar alguma arquitetura de aprendizado de máquina convencional.

As árvores são um tipo especial de grafo (grafos direcionados acíclicos), que possibilitam representação de dados em tarefas que envolvem dados hierárquicos ou recursivos, como os exemplos vistos anteriormente de regressões simbólicas.

Diversas pesquisas têm adaptado e aplicado transformadores e outras redes recorrentes de grafos para trabalhar com árvores, gerando variantes como os "Tree Transformers", que alteram o mecanismo de *self-attention* para incentivar a aderência a estruturas de árvore, como [6] e [7].

Outros trabalhos exploraram técnicas específicas para contextos de regressão simbólica e análise molecular com *autoencoders variacionais* para representar árvores, codificando-as em um espaço latente para posterior uso em tarefas como reconstrução e recuperação de fórmulas, como visto em [8] e [9].

F. Large language models (LLMs)

Large language models ou Modelos de Linguagem de Grande Escala (LLMs) foram inicialmente propostos no artigo [10] e são a maior inovação recente para sistemas de inteligência artificial treinados para entender e gerar linguagem natural. Eles são treinados em extensos conjuntos de dados, como milhões de páginas da internet, aprendendo padrões, estruturas de linguagem e uma ampla gama de informações de forma não supervisionada, generalizando bem para situações e contextos inéditos. Modelos em voga, como o *ChatGPT4*, são membros dessa classe. São especialmente úteis para diversas tarefas por sua versatilidade, uma vez que podem realizar uma variedade de tarefas relacionadas à linguagem, como tradução, resumo de textos, geração de respostas, etc., sem a necessidade de treinamento específico para cada tarefa. Além disso, são extensivamente adaptáveis, aprendendo a partir de instruções ou exemplos de diversas modalidades sem muita estruturação e pré-processamento, enquanto ainda capturam nuances e complexidades da linguagem, tornando-os eficazes em tarefas de linguagem natural.

G. Modelo LLaMA

O artigo *LLaMA: Open and Efficient Foundation Language Models* [11] introduz o LLaMA, uma coleção de foundation language models indo de 7 bilhões a 65 bilhões de parâmetros. Estes modelos foram treinados com trilhões de tokens utilizando apenas datasets públicos, com o LLaMA-13B superando o GPT-3 (175B) na maioria dos benchmarks, enquanto o LLaMA-65B mostrou-se competitivo com os modelos de ponta Chinchilla-70B e PaLM-540B.

Modelos de Linguagem de Grande Escala (LLMs) têm demonstrado sua habilidade em realizar novas tarefas com poucos exemplos ou instruções textuais. Esta capacidade emergiu com o aumento do tamanho dos modelos, levando a uma linha de pesquisa focada na ampliação desses modelos. No entanto, trabalhos recentes sugerem que melhores desempenhos podem ser alcançados por modelos menores treinados com mais dados, dentro de um orçamento computacional definido.

A avaliação dos modelos LLaMA em benchmarks de raciocínio matemático, como MATH e GSM8k, revela que o LLaMA-65B superou o Minerva-62B em problemas de matemática do ensino médio, apesar de não ter sido especificamente treinado em dados matemáticos.

Recentemente a Meta publicou o LLaMA 2 [12], com resultados melhores que o antecessor em uma ampla gama de tarefas e também trabalhando com dados disponíveis publicamente. Esse modelo foi liberado para a comunidade de pesquisa, possibilitando *fine-tuning* e aplicações em outras áreas, algo que consideramos em nosso trabalho, principalmente pela performance em tarefas matemáticas.

H. Redes neurais convolucionais em grafos

Relembrando, um grafo $G(V, E)$ é uma estrutura de dados contendo um conjunto de vértices (nós) $i \in V$ e um conjunto de arestas $e_{ij} \in E$ conectando os vértices i e j. Se dois nós i e j estão conectados, $e_{ij} = 1$, e $e_{ij} = 0$ caso contrário. Se tivermos um grafo não direcionado $e_{ij} = e_{ji}$ se ele não for isso não é necessariamente verdade. Esta informação de conexão pode ser armazenada em uma Matriz de Adjacência A para representar o grafo.

Para entender o que uma Rede Neural de Grafos (GNN) faz, vamos entender as etapas que são realizadas em cada nó do grafo: passagem de Mensagens, agregação e atualização.

Um nó (ou vértice) representa uma entidade, tendo várias propriedades características. Estas propriedades são tipicamente representadas usando vetores em $\mathbb{R}^d$. Este vetor pode ser uma incorporação latente ou construído de uma maneira onde cada entrada é uma propriedade diferente da entidade, como um *one hot encoding* das variáveis categóricas.

A Vizinhança N_i de um nó i é definida como o conjunto de nós j conectados a i por uma aresta. Formalmente, $N_i = \{j : e_{ij} \in E\}$.

1) Passagem de Mensagens: Geralmente, nós com características ou propriedades semelhantes estão conectados uns aos outros. O GNN explora esse fato e aprende como e por que nós específicos se conectam uns aos outros, enquanto alguns não, examinando as Vizinhanças dos nós. Passagem de Mensagens é definida como o processo de tomar características desses nós vizinhos, transformá-las e "passá-las" para o nó de origem. Este processo é repetido, em paralelo, para todos os nós no grafo. Desta forma, todas as vizinhanças são examinadas ao final desta etapa.

Vamos imaginar um grafo em que o nó 2 possui a vizinhança $N_2 = \{1, 3, 4\}$. Pegamos cada uma das características dos nós x_1, x_3, e x_4, e as transformamos usando uma função F, como a transformação afim $F(x_j) = W_j \cdot x_j + b$, que é basicamente multiplicar por um peso W_j e somar um viés b.

2) Agregação: Agora que temos as mensagens transformadas $\{ F(x_1), F(x_3), F(x_4) \}$ passadas para o nó atual, 2, precisamos agregá-las de alguma maneira. Existem várias formas de combiná-las, como:

1) Soma: $= \sum_{j \in N_i} W_j \cdot x_j$
2) Média: $= \frac{\sum_{j \in N_i} W_j \cdot x_j}{|N_i|}$
3) Máximo: $= \max_{j \in N_i} \{W_j \cdot x_j\}$
4) Mínimo: $= \min_{j \in N_i} \{W_j \cdot x_j\}$

3) Atualização: Utilizando estas mensagens agregadas, a camada GNN agora precisa atualizar as características do nó original i. Ao final desta etapa de atualização, o nó deve saber não apenas sobre si mesmo, mas também sobre seus vizinhos. Isso é garantido ao pegar o vetor de características do nó i e combiná-lo com as mensagens agregadas. Novamente, uma operação simples de adição ou concatenação cuida disso. Usualmente é feita uma abordagem iterativa, com o valor de $H_0 = X$ no momento inicial.

4) Otimizações: Podemos realizar diversas otimizações no processo, como normalizações e regularizações. Vamos começar juntando etapas em uma só operação de convolução, expressa pela seguinte equação:

$$H^{(l+1)} = \sigma\left(D^{-1} \cdot A \cdot H^{(l)} \cdot W^{(l)}\right)$$

onde $H^{(l)}$ é a representação dos nós na camada l, A é a matriz de adjacência do grafo com autoconexões adicionadas, D é a matriz diagonal dos graus de $\hat{A}$, porém invertida, $W^{(l)}$ são os pesos da camada l, e σ é uma função de ativação não linear. É uma representação inocente, que pode ser melhorada.

A adição de autoconexões na matriz de adjacência $\hat{A} = A + I$ (onde I é a matriz identidade) ajuda a regularizar o processo de aprendizado, incluindo a informação do próprio nó na atualização das características, já que em uma matriz de adjacência $A_{i,i}$ geralmente é 0, já que autoconexões não foram adicionadas.

A multiplicação por $\hat{D}^{-\frac{1}{2}}$ antes e depois de $\hat{A}$ é uma forma de normalização, onde $\hat{D}$ é a matriz diagonal dos graus de $\hat{A}$. Esta normalização é crucial para evitar o problema de escalonamento das características dos nós. É equivalente anormalizar inicialmente por $\sqrt{d_n}$ e em seguida por $\sqrt{d_m}$, tal que todos os elementos da matriz de adjacência sejam divididos por $\sqrt{d_m} \cdot \sqrt{d_m}$

A forma final da equação representa, portanto, a atualização das características dos nós na próxima camada $(l + 1)$, onde cada nó é atualizado com base nas informações de seus vizinhos e de si mesmo, ponderadas pelos pesos da camada $W^{(l)}$, e transformadas pela função de ativação σ, utilizando $\hat{D}$ e $\hat{A}$.

$$H^{(l+1)} = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

[13] [8]

5) Variacional Auto Encoders: VAEs são algoritmos não supervisionados, isto é, treinados sem uma variável dependente associada. São classificados como modelos generativos, o que significa que eles aprendem a distribuição de probabilidade dos dados de entrada, entendendo e replicando a distribuição dos dados, além de poderem gerar novas amostras de dados que são semelhantes aos dados de treinamento. Esta capacidade os torna úteis em áreas como a síntese de imagens, modelagem de linguagem e outras aplicações onde a geração de novos conteúdos é desejável. [14]

A principal característica dos VAEs em relação a outros modelos generativos é a sua abordagem probabilística para aprender a representação latente, usando a divergência KL (Kullback-Leibler) para garantir que as representações latentes sejam distribuídas de maneira significativa.

Ele consiste em duas partes principais: o codificado e o decodificador:

O codificador mapeia os nós do grafo para um espaço latente. Para um grafo $G(V, E)$ com nós $i \in V$ e arestas $e_{ij} \in E$, o codificador é uma função que mapeia cada nó i para um vetor latente z_i no espaço latente. Em GVAEs, este mapeamento é geralmente feito por uma Graph Convolutional Network (GCN), que pode ser expressa como:

$$z_i = GCN(X, A)$$

onde X é a matriz de características dos nós e A é a matriz de adjacência do grafo.

O decodificador reconstrói o grafo a partir das representações latentes. Para um vetor latente z_i, o decodificador tenta reconstruir as arestas do grafo. Isso é frequentemente realizado por meio de um simples produto escalar entre os vetores latentes:

$$\hat{A}_{ij} = \sigma(z_i^T z_j)$$

onde $\hat{A}_{ij}$ é a probabilidade reconstruída de uma aresta existir entre os nós i e j, e σ é uma função de ativação, como a sigmoid.

O treinamento de um GVAE envolve a minimização da diferença entre o grafo de entrada e o grafo reconstruído, juntamente com a regularização imposta pela distribuição das representações latentes. Este processo é descrito pela função

de perda, que é uma combinação da reconstrução do grafo e dos termos de regularização da divergência KL.

Esta abordagem permite aprender representações compactas e significativas dos nós em grafos através de aprendizado não supervisionado, focando no aprendizado de representações de qualidade. Isso é fundamental para representações eficientes de árvores de programação genética.

I. Geometria hiperbólica

A geometria hiperbólica, uma das geometrias não euclidianas, surgiu do questionamento do quinto postulado de Euclides, conhecido como o postulado das paralelas, que afirma que por um ponto fora de uma linha, passa exatamente uma linha paralela a essa linha, levando ao desenvolvimento da geometria hiperbólica.

A curvatura da geometria hiperbólica é negativa, o que é uma diferença fundamental em relação à geometria euclidiana, que tem curvatura zero (plana). Em termos matemáticos, a curvatura K é expressa como:

$$K < 0 \quad \text{(Geometria Hiperbólica)}$$

$$K = 0 \quad \text{(Geometria Euclidiana)}$$

$$K > 0 \quad \text{(Geometria Esférica)}$$

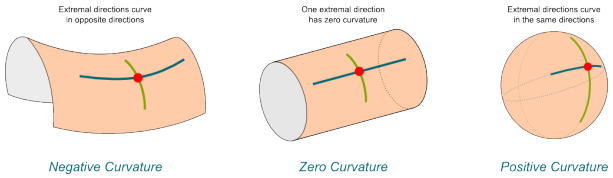

Fig. 7. Sela representando espaço hiperbólico

Na geometria euclidiana, a soma dos ângulos de um triângulo é sempre 180 graus, e as linhas paralelas nunca se encontram. Em contraste, na geometria hiperbólica:

1) A soma dos ângulos de um triângulo é sempre menor que 180 graus.
2) Existem infinitas linhas paralelas que passam por um ponto fora de uma linha dada.

Para conseguir caminhar entre as duas geometrias, utilizamos mapeamentos entre o espaço euclidiano, que é um espaço tangente a um ponto no espaço hiperbólico, e o espaço hiperbólico. O mapeamento exponencial converte um ponto num espaço euclidiano tangente para o espaço hiperbólico e o mapeamento logarítmico converte um ponto num espaço hiperbólico para o espaço tangente euclidiano.

Vamos agora fazer um comparativo entre distância e produto interno nas duas geometrias:

Distância:

$$d_{\text{hip}}(x,y) = \operatorname{arcosh}\left(1 + 2\frac{\|x-y\|^2}{(1-\|x\|^2)(1-\|y\|^2)}\right)$$

$$d_{\text{euc}}(x,y) = \|x-y\|$$

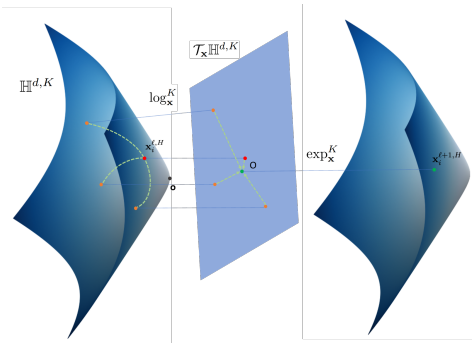

Fig. 8. Mapeamento entre espaço euclidiano tangente e hiperbólico

Produto interno:

$$\langle u,v\rangle_{\text{hip}} = -u_0v_0 + u_1v_1 + \cdots + u_nv_n$$

$$\langle u,v\rangle_{\text{euc}} = u_1v_1 + \cdots + u_nv_n$$

Existem várias representações de geometria hiperbólica, como Disco de Poincaré, Variedade Lorenziana, Modelo de Klein, dentre outros. Cada um tem suas fórmulas específicas de conversão e aplicações mais usadas. Por motivos de estabilidade numérica, a Variedade Lorenziana usualmente performa melhor em muitos modelos. [15]

J. Redes neurais hiperbólicas

O artigo ”Lorentzian Graph Convolutional Networks” [15] explora uma arquitetura para redes convolucionais de grafos utilizando geometria hiperbólica. O foco principal é melhorar a representação de grafos com estruturas hierárquicas ou de escala livre. Isso acontece devido a problemas na representação de estruturas dessa forma em espaço euclidiano, já que os dados se ”amontoam”.

Vamos considerar uma árvore com fator de ramificação b, que tem $(b+1)b^{l-1}$ nós no nível l e $\frac{(b+1)b^l-2}{b-1}$ nós nos níveis menores ou iguais a l. Assim, à medida que aumentamos os níveis da árvore, o número de nós cresce exponencialmente. Isso gera problemas no espaço euclidiano, já que nós equidistantes passam a se aproximar uns dos outros sem necessariamente serem mais semelhantes, apenas pela ocupação do espaço

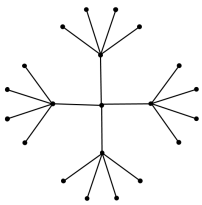

Fig. 9. Projeção de árvore em espaço de representações euclidiano

Uma possível solução para esse problema é usar um espaço curvado, em que conforme os pontos se distanciam do centro eles estão exponencialmente mais longes, evitando

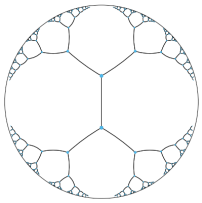

Fig. 10. Projeção de árvore em espaço de representações hiperbólico com b=2

a superocupação do espaço. A geometria hiperbólica fornece essa solução com elegância

[16]

Como pode ser visto, os pontos estão consideravelmente melhor posicionados. Os pontos na fronteira são difíceis de visualizar devido ao mapeamento para espaço euclidiano, que apresenta perda de informação.

Um problema de modelos hiperbólicos é que não são espaços vetoriais, o que significa que as operações definidas em espaços euclidianos, como agregação, passagem de mensagem e atualização não podem ser aplicadas em espaços hiperbólicos.

O modelo assegura que as operações de grafos, incluindo transformação de características e ativação não linear, sejam definidos e rigorosos para geometria hiperbólica. Uma característica distinta do LGCN é a agregação de vizinhança baseada no centróide da distância Lorentziana, garantindo que as características dos nós sejam agregadas de maneira matematicamente significativa. Além disso, o LGCN demonstra equivalência em operações de grafos em diferentes modelos de geometria hiperbólica, como o modelo de bola de Poincaré.

Os experimentos realizados mostram que o LGCN supera os métodos existentes em tarefas de predição de links e classificação de nós, além de apresentar menor distorção ao aprender a representação de grafos semelhantes a árvores em comparação com GCNs hiperbólicos existentes. O LGCN oferece uma nova arquitetura particularmente útil para problemas de árvores, uma vez que representam grafos hierárquicos direcionados, podendo proporcionar uma excelente plataforma para construção de nosso modelo de representação de árvores de programação genética.

III. Contribuições atuais

Esses foram os desenvolvimentos no decorrer das disciplina de POC I e II

A. Revisão Literária e de Software

1) **Revisão Bibliográfica**
   Foi necessária uma análise aprofundada da literatura existente e trabalhos relacionados, além de entendimento total do artigo principal [1]. Isso nos permitiu identificar um caminho futuro para o trabalho com redes neurais em grafos, principalmente por contribuições dos artigos [15] e [8].
2) **Busca e aprendizado de bibliotecas** O desenvolvimento do projeto demandou o reconhecimento e aprofundamento em bibliotecas especializadas de aprendizado de máquina e processamento de grafos. Uma das bibliotecas-chave exploradas foi a Geometric Graph Embedding[1], que fornece ferramentas avançadas para o embedding de grafos em geometrias não euclidianas. Adicionalmente, aprofundamo-nos em PyTorch[2] e sua extensão PyTorch Geometric[3], que oferecem um ecossistema rico para o desenvolvimento de redes neurais em grafos com operações eficientes e escaláveis. Além disso, exploramos a biblioteca gplearn[4], que implementa algoritmos de programação genética para aprendizado de máquina simbólico, visando a construção de bases de dados sintéticas expressivas e parametrizáveis.
3) **Estudo de representações semânticas** Foi estudado o problema de geração de representações que consideram atributos sintáticos e semânticos e um obstáculo notável é conseguir transpor conhecimento matemático do problema, como regras algébricas, para um modelo treinável. Analisando várias opções formulamos um experimento de substituir os atributos de cada nó, que atualmente são um label com a operação matemática utilizada, por um embedding dessa operação baseado em *LLM*. Dessa forma, poderíamos codificar todas as operações do nosso conjunto de labels nas suas representações ao executar o modelo de linguagem. Em seguida, realizaremos redução de dimensionalida nos dados para remover dimensões desnecessária, já que estamos lidando apenas com operações matemáticas. O modelo escolhido até o momento foi o LLaMA, devido à sua facilidade de utilização, treino com dados públicos e acesso gratuito.

B. Reprodução dos Resultados do Artigo, Identificação de Problemas e Melhorias de Código

O código utilizado para experimentação no artigo [1] foi recuperado e analisado, visando entender detalhes de implementação e identificar melhorias. O autor providenciou o código bruto, nos restando organizar e resolver problemas em diversos componentes erroneamente refatorados.

Uma área que demandou atenção adicional foi o pré processamento de dados para treino e inferência, possibilitando a extensão para experimentos com mais graus de liberdade através de uma interface única baseada em *DOT Language*, formato agnóstico de implementação para representação de grafos.

Foram observados problemas no pré processamento, em que a árvore de expressões era convertida em uma árvore completa, preenchida com tokens especiais, linearizada em um vetor de tokens e em seguida convertida para uma representação numérica categórica dos tokens. Isso é problemático pela desconsideração de comutatividade, permitindo que o modelo

[1]https://github.com/iesl/geometric-graph-embedding
[2]https://pytorch.org/
[3]https://pytorch-geometric.readthedocs.io/
[4]https://gplearn.readthedocs.io/

não consiga inferir a semântica de operações corretamente, apenas atributos estruturais da árvore, ou seja, se mudarmos a ordem de uma adição, a representação se distanciará significativamente no espaço, problema que representações eficientes não devem apresentar significativamente. Isso motivou a experimentação com representações alternativas para as expressões, como pós-fixada e com redução de constantes para um único token.

Por fim, identificamos dados duplicados entre teste e treino, resultando em vazamento do teste, de forma que o modelo na etapa de teste estava sendo avaliado com informações já vistas e treinadas. O problema da duplicação resultava em um conjunto de dados enviesado, piorando a performance e afetando a asserção de representatividade do modelo.

C. Consolidação de Técnica Para Geração de Árvores de Expressão

Foi necessária a criação de alguma técnica de geração sintética de dados de acordo com nossos parâmetros de experimentação, generalizando melhor o modelo para o experimento e resultando em maior significância estatística para nossas técnicas. É importante diferenciar os tipos de bases de dados no *pipeline* de experimentação. Existem duas bases de dados envolvidas: de função e de árvore. É importante esclarecer que em um regressor simbólico, se $h_{\max}$ é a altura máxima permitida para árvore, então $\mu = 2^{h_{\max}} - 1$, onde μ é o número máximo de nós nas árvores de cada geração.

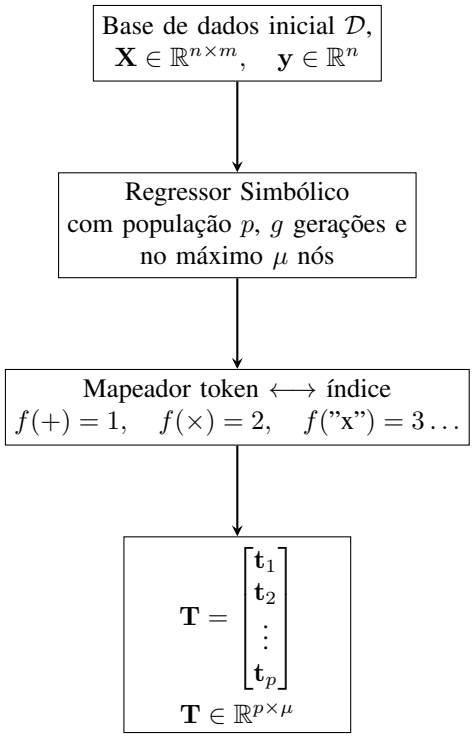

Fig. 11. Processo de geração de uma base de dados de árvores derivada de uma base de dados tabular

Para a base de dados inicial, é possível optar pelo uso de um arquivo existente ou uma estratégia de criação de conjuntos de variáveis X e valores dependentes Y para compor uma base sintética. Dessa forma, é permitido escolher uma distribuição real ou sintética para alimentar a geração sintética de árvores, formando a base de dados de entrada $\mathcal{D}$.

Em seguida, escolhemos parâmetros $\mathcal{P}$ para o regressor simbólico limitando altura máxima e mínima da árvore e aumentando o coeficiente de parcimônia (penaliza soluções muito grandes). Dessa forma, garantimos que grande parte das árvores obedeça uma altura máxima parametrizada no experimento, já que o modelo atual é treinado com essa altura máxima em consideração. Se árvores maiores forem geradas, utilizamos duas estratégias: remoção com substituição por outras aleatórias até completar a base de dados ou poda, em que substituímos todos os nós na altura máxima permitida por folhas com *labels* terminais, como variáveis ou constantes.

Criaremos então um regressor simbólico $\mathcal{R}\langle\mathcal{D}, \mathcal{P}\rangle$ que será executado com inícios amostrados de resultados anteriores, possibilitando que nossa base de dados represente diversas árvores aleatórias próxima do início e gradativamente adquira mais propriedades de programações genéticas, já que estará adquirindo árvores geradas por operações genéticas e selecionadas de acordo com sua qualidade para o problema sintético, se aproximando de uma base de testes representativa para problema de aprendizado de representações para árvores de programação genética.

As árvores de cada geração serão concatenadas e transformadas por um mapeador, transformando-as em atributos numéricos, formato da representação do problema, resultando na base de dados matricial $\mathbf{T} \in \mathbb{R}^{p \times \mu}$.

D. Melhorias na linearização

Foi observado que a linearização da árvore com um algoritmo de busca em profundidade (*DFS*) padrão, chamado de *in-order*, resulta na perda de contexto da estrutura da árvore e seus operadores em expressões grandes, perdendo capacidade de representação de uma expressão genérica. Para resolver isso, duas hipóteses de melhorias foram estabelecidas

1) *postfix*: *DFS* em modo pós-fixo ou notação polonesa reversa, representando $1{+}2$ como $12{+}$. Isso tem o benefício de dispensar parênteses ou *tokens* de espaçamento, já que a linearização da árvore segue um formato de pilha.
2) *use_single_const*: As constantes são transformadas em *bins* de amplitude definida, discretizando um intervalo contínuo em n tokens possíveis. Foi observado que muitos valores eram subrepresentados nas bases de treino, falhando nos casos de teste. Muitos algoritmos de regressão simbólica praticam a remoção de constantes como uma das etapas, rodando um otimizador posterior na melhor árvore para cálculo dos valores corretos para aquela forma de função, o que inspirou a inclusão desse experimento.
3) Ambos: Aplicar ambas as técnicas e ver se o resultado é melhor que elas individualmente.

Codificamos isso em nosso experimento com uma função das duas variáveis binárias $E(p, c) = 2p + c$ com $p, c \in \{0, 1\}$,

convenção numérica usada extensivamente em nosso *pipeline* de experimentação.

E. Diversificação e Regularização das Bases de Dados

Relembrando a técnica, a geração de árvores para o modelo é sintética, derivada de outra base de dados, sintética ou real. Utilizamos os vetores X e cada valor de y para treinar um regressor simbólico, parametrizado para garantir máxima diversidade com g gerações de população p. Ao final de g gerações, teremos $g \cdot p$ árvores, formando então a base de dados sintética derivada.

Executamos o algoritmo de geração de árvores de expressão utilizado no artigo [1] e agrupamos/filtramos os dados resultantes de diversas combinações de parâmetros, revelando conclusões interessantes, como mostrado na figura fig. 14. Em seguida, desenvolvemos um algoritmo algorithm 1 capaz de combinar todos os datasets sem repetições.

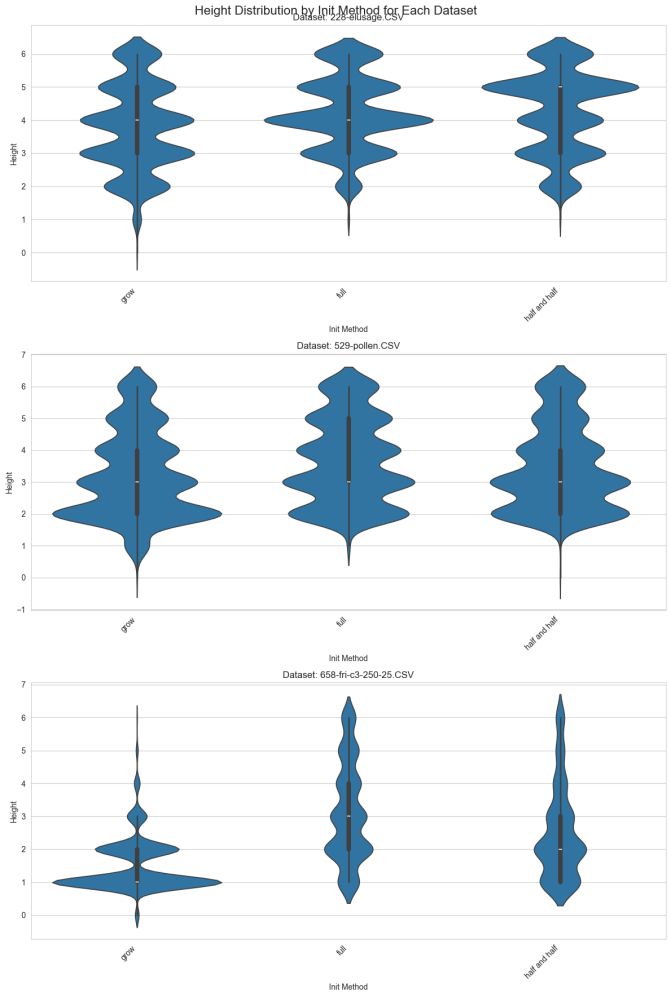

Fig. 12. Distribuição de alturas de cada árvore gerada em cada dataset por tipo de inicialização

Nota-se que a distribuição de alturas muda muito de acordo com a dimensionalidade e propriedades da distribuição de dados da base utilizada, o que pode desbalancear a representatividade de alturas e introduzir viés no modelo, algo que não é completamente resolvido combinando todas as bases possíveis. Para isso, decidimos repetir a geração de datasets com técnicas diferentes de início de árvore no regressor genético e concatenar as árvores resultantes, gerando uma distribuição razoávelmente uniforme em termos de alturas.

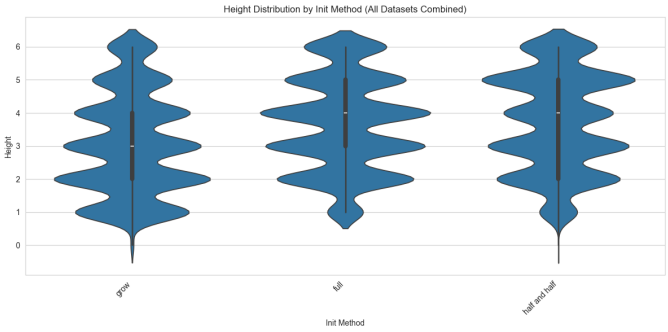

Fig. 13. Distribuição de alturas de cada árvore gerada em cada dataset por tipo de inicialização

O algoritmo final para criação da base de dados é descrito a seguir, com o método *create_tree_df* sendo responsável pela instanciação, execução e coleta de resultados do regressor simbólico dadas duas variáveis binárias: *postfix* e *use_single_const*

Algorithm 1 Create Trees Given All Datasets and Methods

Require: $\mathcal{D}$ (set of datasets), $\mathcal{M}$ ("grow", "full", "half and half"), postfix, use_const_refs
Ensure: F (final DataFrame), A (all metrics)

$F \leftarrow \emptyset$
$A \leftarrow \emptyset$
for all $d \in \mathcal{D}$ **do**
  for all $m \in \mathcal{M}$ **do**
    $\langle D_i, M_i \rangle \leftarrow$ create_tree_df$(d, m,$ postfix, use_const_refs$)$
    $R \leftarrow D_i \cap F$
    $I_R \leftarrow \{j : D_i[j] \in R\}$
    $D_i \leftarrow D_i \setminus \{D_i[j] : j \in I_R\}$
    $F \leftarrow F \cup D_i$
    $M_i \leftarrow M_i \setminus \{M_i[j] : j \in I_R\}$
    $A \leftarrow A \cup M_i$
  end for
end for
  return F, A

A técnica de regularização utilizada foi a remoção parcial de árvores semelhantes, aumentando a diversidade do modelo. Para isso, criamos *bins* primários baseados na altura de cada árvore. Dentro de cada um, foram criados 10 *bins* de quantis, separando então cada *fitness* em decis. Por fim, um agrupamento terciário linear foi feito dentro de cada decil de acordo com a *fitness*. Com o *bin* terciário gerado, são removidos $\mathcal{P}\%$ de cada *bin* terciário, resultando na remoção de árvores com fitness parecidas. Essa técnica ainda está em sua infância, com perspectiva de exploração futura em outros artigos de pesquisa.

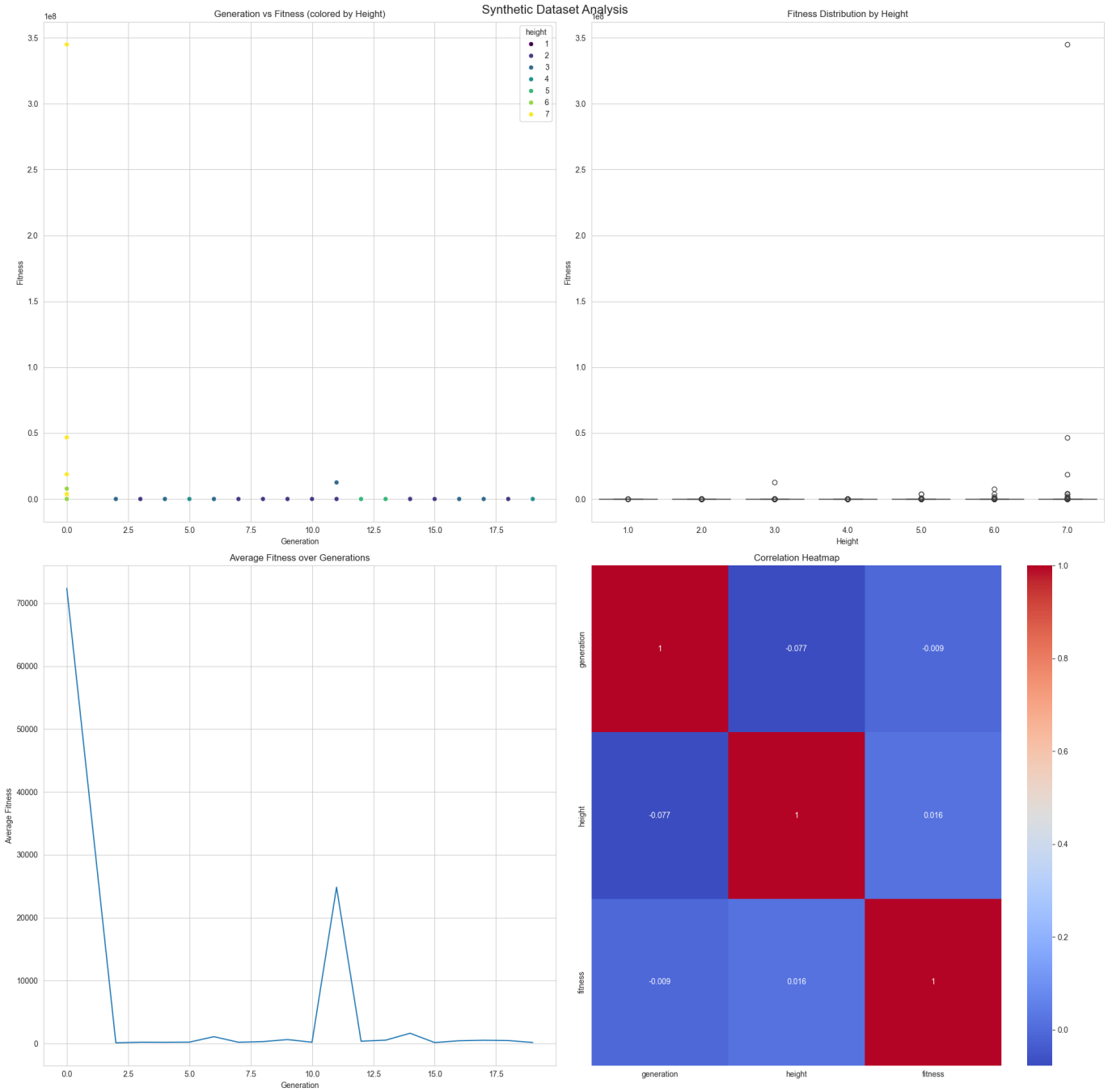

Fig. 14. Análise do dataset sintético gerado pelos datasets

F. Avaliação de qualidade e extensão do código

O código bruto utilizado para o experimento carecia de interfaces dedicadas para manipulação e extensão de processos de pré processamento, então foi criada uma nova classe no modelo capaz de realizar as operações faltantes e fornecer *API*'s flexíveis para codificação e decodificação através de formato comum, permitindo eficiente integração em todos os momentos do pipeline de treino e execução do modelo. Todos os arquivos principais e lógicas em *notebooks* foram extraídos para módulos autocontidos e um contexto global de configuração de hiper parâmetros.

Integramos o framework de *MLOps* Mlflow[5], capaz de gerenciar experimentos já rodados, coleta de parâmetros, métricas, tags, artefatos a até modelos treinados para reprodução futura, permitindo fácil comparação e análise dos resultados intermediários ao longo da disciplina. Isso permitiu a criação de um *pipeline* completo de gestão, coleta e análise

[5]https://mlflow.org/

de dados, servindo de base e *benchmark* para estudos futuros nessa área, sem necessidade de coletar dados manualmente e criar visualizações com bibliotecas como *matplotlib*.

Um módulo pricipal de gestão de experimentos foi criado, capaz de criar permutações dos parâmetros de experimento desejados e agendar sua execução integrada com o Mlflow, pulando se algum experimento daquele tipo já foi executado.

Com a coleta de métricas, geração de dados e treinamento bem estabelecidos já é possível experimentar melhor com os dados para validar as hipóteses, porém apenas minimizar erro e maximizar acurácia não é um indicativo de uma hipótese correta, já que o modelo pode aprender correlações espúrias. Para isso, criamos diversas *downstream tasks*, que são tarefas executadas após o treino do modelo em problemas derivados para avaliação de qualidade.

Foram utilizados dois axiomas para projetar essas tarefas:

1) Agrupamento: Tokens se organização no espaço latente de forma adequada para uma representação eficiente, aproximando-se de elementos parecidos. Com isso, podemos montar um problema de agrupamento e analisar uma representação de dimensionalidade reduzida e avaliar suas métricas, como erro de reconstrução e proporção de variância explicada por cada dimensão. Para essa finalidade foram escolhidos *PCA*, *T-SNE* e *TSVD*, que executamos para coleta de métricas para análise numérica e análise visual com *scatter plots* e dendrogramas.

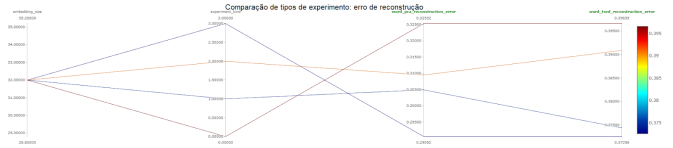

Fig. 15. Erro de reconstrução em experimento exemplo

2) Propriedades matemáticas: Como discutido anteriormente, modelos que aprendem operações matemáticas devem ser capazes de entender que suas árvores que tiveram apenas algumas operações comutativas trocadas são iguais, enquanto modelos que inverteram operações não comutativas são diferentes.

No intuito de medir essa propriedade, foi criada uma sequência de métodos capaz de identificar nós com operações potencialmente problemáticas, que foram separadas em dois grupos:

- Operações idempotentes: Se essas operações comutativas forem aplicadas sucessivamente árvores não deveriam ser menos semelhantes, como o caso de trocar $a+b$ por $b+a$ em algum nó profundo da árvore, já que a expressão final é a mesma
- Operações destrutivas: Se essas operações não comutativas forem aplicadas sucessivamente árvores deveriam ser menos semelhantes, como o caso de trocar a/b por b/a em algum nó profundo da árvore, já que a expressão deixou de ser equivalente.

Em seguida, uma matriz de árvores é criada, de forma que o eixo x represente quantas operações destrutivas foram realizadas e o eixo y o número de operações idempotentes. O elemento $M_{1,3}$ seria uma árvore com 3 operações destrutivas aplicadas e uma idempotente.

Por fim, calculamos a similaridade de todos os elementos com $M_{0,0}$, que representa a árvore original sem modificações. Se nossa representação é eficiente, em um grande número de exemplos haverá pouca variação de similaridade no eixo y, com notável mudança no eixo x.

Isso foi implementado com a geração de um vetor inicial Y, representando as árvores com operações idempotentes, e em seguida a matriz foi construída iterativamente aplicando cada x, representando uma coluna, linhas Y, até todos os valores de x serem gerados

G. Modelos com redes neurais em grafos (GNN's)

O código desenvolvido e resultados de experimentos encontrados foram feitos com a intenção de criar um *benchmark* para mudança de arquitetura, superando o problema de linearização com grafos. Nosso esforço na área se concentrou na revisão de literatura e prototipação de bases de dados unificadas para treinamento das *GNN*'s e aprendizado de *labels* com grandes modelos de linguagem.

IV. Próximos passos e conclusão

Os experimentos realizados para aperfeiçoamento de linearização evidenciaram que a modificação de ordem de operações é uma melhoria notável simples para o modelo, com 10% de melhoria em algumas tarefas *downstream* observadas. Outro ponto importante é a refutação de algumas técnicas e conclusões em [1], resolvendo equívocos e aperfeiçoando algoritmos. Temos como pontos relevantes:

- Melhoria de diversidade nas bases de dados, melhorando generalização
- Limpeza dos dados e remoção de duplicatas
- Identificação e remoção de vazamento treino/teste, que indevidamente aumentava a acurácia do modelo
- Critérios de parada aperfeiçoados com inclusão de parâmetro de paciência e redução do ϵ mínimo
- Coleta de métricas adicionais para validação dos resultados do modelo

Apesar dos desafios associados, as metodologias examinadas mostram um potencial significativo para solucionar questões complexas no âmbito de representação de árvores de programação genética. Em particular, as redes neurais hiperbólicas emergem como uma ferramenta poderosa para a representação eficaz de dados hierárquicos e grafos complexos. O trabalho será consolidado em artigos científicos para revistas e conferências, com a linha de pesquisa sendo desenvolvida em meu mestrado, onde pretendo refinar estas abordagens pode abrir caminhos e interseções de pesquisa ainda inexploradas, principalmente com grafos e grandes modelos de linguagem.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Victor Caetano, Matheus Cândido Teixeira, and Gisele Lobo Pappa. "Symbolic Regression Trees as Embedded Representations". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '23. Lisbon, Portugal: Association for Computing Machinery, 2023, pp. 411–419. ISBN: 9798400701191. DOI: 10.1145/3583131.3590423. URL: https://doi.org/10.1145/3583131.3590423.

[2] Wolfgang Banzhaf et al. *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers Inc., 1998.

[3] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. "Linguistic Regularities in Continuous Space Word Representations". In: *HLT-Naacl*. 2013, pp. 746–751.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[5] Ashish Vaswani et al. "Attention is all you need". In: *arXiv preprint arXiv:1706.03762* (2017).

[6] Qi He, Joao Sedoc, and Jordan Rodu. "Trees in transformers: a theoretical analysis of the transformer's ability to represent trees". In: *arXiv preprint arXiv:2112.11913* (2021).

[7] Seohyun Kim et al. "Code prediction by feeding trees to transformers". In: *arXiv preprint arXiv:2003.13848* (2020).

[8] Zichao Wang et al. "Scientific Formula Retrieval via Tree Embeddings". In: *2021 IEEE International Conference on Big Data (Big Data)*. IEEE. 2021, pp. 1493–1503.

[9] Sander de Bruin, Vadim Liventsev, and Milan Petkovic. "Autoencoders as tools for program synthesis". In: *arXiv preprint arXiv:2108.07129* (2021).

[10] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: 2019. URL: https://api.semanticscholar.org/CorpusID:160025533.

[11] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL].

[12] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL].

[13] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. 2017. arXiv: 1609.02907 [cs.LG].

[14] Thomas N. Kipf and Max Welling. *Variational Graph Auto-Encoders*. 2016. arXiv: 1611.07308 [stat.ML].

[15] Yiding Zhang et al. *Lorentzian Graph Convolutional Networks*. 2021. arXiv: 2104.07477 [cs.LG].

[16] Pierre Collignon. *Notes on Hyperbolic Geometry*. https://collignon.github.io/2020/07/notes-on-hyerpbolic-geometry/. 2020.