UTILIZAÇÃO DA LINGUAGEM DE PROGRAMAÇÃO BEND PARA IMPLEMENTAÇÃO DE ALGORITMOS DE APRENDIZADO DE MÁQUINA: KNN, K-MEANS E ÁRVORES DE DECISÃO

João Arthur Pereira Alba, Danton Cavalcanti Franco Junior– Orientador

Curso de Bacharel em Ciência da Computação
Departamento de Sistemas e Computação
Universidade Regional de Blumenau (FURB) – Blumenau, SC – Brasil

jalba@furb.br, dfranco@furb.br

Resumo: *A linguagem de programação Bend promete facilitar a execução de programas de maneira paralela em placas gráficas. Para tanto, este trabalho apresenta a implementação de três algoritmos de aprendizado de máquina — KNN, K-Means e Árvores de Decisão — utilizando Bend, além de versões equivalentes em Python, com o objetivo de comparar a performance entre as diferentes abordagens. Os experimentos realizados mediram o tempo de execução dos algoritmos sobre conjuntos de dados de diferentes tamanhos. Os resultados obtidos demonstram que, no estado atual da linguagem, Bend não entrega a facilidade de implementação ou a performance prometida, apresentando diversas limitações técnicas que dificultam seu uso prático.*

Palavras-chave*: Bend, Processamento paralelo, Placas gráficas, CUDA*

1 INTRODUÇÃO

Provenientes do campo do aprendizado de máquina, os algoritmos de classificação são amplamente utilizados em diversas áreas do conhecimento, devido à sua capacidade de categorizar dados em grupos predefinidos, com base em características ou padrões presentes nas amostras (Duda; Hart; Stork, 2000). Métodos como K-Nearest Neighbor (KNN), K-Means e árvores de decisão são conhecidos por sua simplicidade e eficiência. O algoritmo KNN, por exemplo, classifica os dados de acordo com seus vizinhos mais próximos (Hastie; Tibshirani; Friedman, 2009), enquanto o K-Means agrupa dados em clusters por similaridade de características (Duda; Hart; Stork, 2000). Esses algoritmos desempenham papel fundamental em áreas como a saúde (Sisodia; Sisodia, 2018), educação (Wolff et al., 2013), entre outros. Embora tais algoritmos sejam eficientes, a execução de classificadores em grandes volumes de dados pode ser custosa em termos de tempo e capacidade de processamento, e é nesse ponto que abordagens paralelas mostram-se vantajosas.

As Unidades de Processamento Gráfico (GPUs) têm sido cada vez mais utilizadas além de sua função original de renderização gráfica, sendo aproveitadas para acelerar diferentes tipos de aplicações. Sua arquitetura massivamente paralela, composta por milhares de núcleos, permite que a GPU execute diversas operações ao mesmo tempo, o que é ideal para algoritmos que envolvem grandes volumes de dados e tarefas repetitivas, como diversos algoritmos de classificação (Kirk; Hwu, 2016). A utilização de GPUs neste contexto tem grande impacto na redução do tempo de processamento, principalmente ao explorar o paralelismo presente nos cálculos. Assim, as GPUs são essenciais para a implementação de algoritmos complexos que necessitam de grande capacidade computacional, oferecendo uma alternativa poderosa ao processamento sequencial tradicional de Unidades Centrais de Processamento (CPUs) (Kirk; Hwu, 2016).

O uso desses componentes para criação de programas paralelos pode ser realizado através de ferramentas como a Interface de Programação de Aplicação (API) Compute Unified Device Architecture (CUDA), desenvolvida pela NVIDIA (Kirk; Hwu, 2016). CUDA oferece uma interface de comunicação com a placa gráfica, permitindo que desenvolvedores aproveitem a arquitetura das GPUs para paralelizar suas aplicações. Apesar disso, programar em CUDA pode ser uma tarefa desafiadora, exigindo um profundo entendimento tanto da arquitetura do hardware quanto das técnicas de paralelização adequadas, o que dificulta sua adoção por desenvolvedores menos experientes (Yamato, 2021). A programação em CUDA também impõe a necessidade de lidar com questões complexas de sincronização, alocação de memória e balanceamento de carga entre a CPU e a GPU, tornando o desenvolvimento mais complexo do que em ambientes de programação tradicionais.

Criada pela empresa Higher Order Company (HOC) e com o objetivo de tornar trivial a programação paralela, a linguagem de programação Bend, incluindo o compilador HVM2 (Higher-order Virtual Machine 2), promete criar algoritmos que sejam executados nativamente na GPU, utilizando o poder do processamento paralelo, mas mantendo uma sintaxe de alto nível (Taelin, 2024). A linguagem faz uso de dois sistemas matemáticos para traduzir o código criado em

instruções que são automaticamente executadas pela GPU: Cálculo lambda e combinadores de interação, este último proposto por Lafont (1997).

Considerando a atualidade do tema, o objetivo deste trabalho é avaliar a viabilidade da implementação dos algoritmos KNN, K-Means e árvores de decisão na linguagem de programação Bend. Para isso, foi necessário implementar os algoritmos em Bend, mensurar a performance obtida durante os testes e comparar os resultados encontrados com implementações nativas em CUDA. Por fim, foi analisada a facilidade e simplicidade das implementações em Bend.

2 FUNDAMENTAÇÃO TEÓRICA

Esse capítulo aborda fundamentações e conceitos sobre aprendizado de máquina, placas gráficas, CUDA, e a linguagem de programação Bend, assim como artigos relacionados ao trabalho, abordando os algoritmos KNN, K-Means e árvores de decisão.

2.1 CONCEITOS, TÉCNICAS E/OU FERRAMENTAS

Esta subseção descreve os assuntos que fundamentarão o trabalho: algoritmos de aprendizado de máquina, placas gráficas e processamento paralelo, e a linguagem de programação Bend.

2.1.1 ALGORITMOS DE APRENDIZADO DE MÁQUINA

Segundo Marsland (2014), o campo do aprendizado de máquina é sobre fazer computadores modificarem ou adaptarem suas ações, para que seus resultados futuros sejam mais precisos. Da mesma maneira, Mitchell (1997) define que "o campo do aprendizado de máquina está preocupado com a questão de como construir programas que melhoram automaticamente com a experiência". No contexto deste trabalho, serão apresentados conceitos de duas áreas do aprendizado de máquina: algoritmos de classificação e algoritmos de agrupamento.

De acordo com Marsland (2014), o problema da classificação consiste em usar valores de entrada e decidir a qual classe os mesmos pertencem, baseando-se no treinamento com exemplos de cada classe. Esse tipo de abordagem se baseia no aprendizado supervisionado, no qual é conhecida a resposta correta para certo valor de entrada (Marsland, 2014). Geralmente, este treinamento se dá pela criação de um modelo, onde dados de treinamento são compilados conforme o algoritmo escolhido, e posteriormente, este modelo é utilizado para prever a classe de novos valores.

O algoritmo de árvores de decisão, apesar de comum, vem se popularizando pela sua simplicidade e eficiência. O modelo é baseado em criar perguntas sobre as características do dado de entrada, e baseado na resposta, seguir um novo caminho com outras perguntas, até que seja alcançada uma "folha" da árvore, que irá resultar em uma classe (Marsland, 2014).

O algoritmo K-Nearest Neighbor (K vizinhos mais próximos), diferente de outros algoritmos, não cria um modelo a ser utilizado na predição. Para encontrar a classe de uma entrada, o algoritmo calcula a distância entre todas as características do objeto alvo e dos objetos de exemplo, e dessa maneira, a classe da maioria dos K vizinhos mais próximos se torna a classe do objeto alvo (Mitchell, 1997). Pelo fato de não criar um modelo, este algoritmo pode apresentar uma baixa performance quando utilizado com uma grande base de treinamento.

Ainda na área do aprendizado de máquina, mas diferente dos algoritmos de classificação, os algoritmos de agrupamento não são supervisionados, ou seja, a informação sobre as classes corretas dos exemplos fornecidos não está disponível no momento do treinamento, apenas suas características. Nesse contexto, o algoritmo K-Means é utilizado para relacionar os objetos-alvo de uma base de dado, utilizando as suas características e criando centroides para identificar os grupos. A operação se baseia na criação de centroides aleatórios, que serão utilizados como referência para os objetos-alvo mais próximos dele, e após isso, a posição do centroide é recalculada a partir dos próprios objetos-alvo. Após algumas iterações, os objetos estarão relacionados a um centroide, demonstrando que fazem parte do mesmo grupo (Hastie; Tibshirani; Friedman, 2009).

Esses três algoritmos foram escolhidos como base para esse estudo por conta de sua simplicidade de implementação, o que os torna mais adequados para uma primeira exploração de uma linguagem como Bend. Além disso, são amplamente utilizados nesta área de estudo, o que facilita a busca por bibliotecas que implementem estes algoritmos.

2.1.2 PLACAS GRÁFICAS, PROCESSAMENTO PARALELO E CUDA

As Unidades de Processamento Gráfico (GPUs) foram inicialmente projetadas para acelerar o processamento de gráficos em computadores, mas ao longo do tempo, evoluíram para uma poderosa ferramenta de uso geral, oferecendo uma capacidade de processamento maior que a de uma CPU (Owens et al., 2008).

De acordo com Owens et al. (2008), a arquitetura das GPUs é composta por centenas ou milhares de núcleos capazes de executar múltiplas instruções simultaneamente, o que as torna ideais para tarefas computacionalmente

intensivas. Por conta da sua capacidade de processamento paralelo, foram criadas ferramentas como a API CUDA, que fornece aos desenvolvedores a capacidade de utilizar GPUs de maneira generalizada, criando uma interface de comunicação com a arquitetura paralela (Kirk; Hwu, 2016).

CUDA é uma plataforma de computação paralela e uma interface de programação desenvolvida pela NVIDIA em 2007, que permite aos desenvolvedores utilizarem a GPU para computação geral. Geralmente utilizada nas linguagens C e C++, a API possibilita a interação CPU-GPU através de código, o que permite que tarefas computacionalmente intensas sejam processadas de maneira paralela e com mais velocidade (Kirk; Hwu, 2016).

2.1.3 A LINGUAGEM DE PROGRAMAÇÃO BEND

Criada pela empresa brasileira Higher Order Company, a linguagem de programação Bend foi disponibilizada para o público em 2024. A linguagem tem como objetivo facilitar a programação paralela em GPUs, evitando problemas que podem surgir quando essa arquitetura é utilizada, como condições de corrida, deadlocks e necessidade de sincronização (Yamato, 2021). Para isso, Bend utiliza o compilador HVM2, criado pela mesma empresa, para executar algoritmos na GPU de maneira automática.

O compilador HVM2 utiliza o sistema de combinadores de interação descrito por Lafont (1997), que, por ser turing completa, permite a criação de qualquer sistema computacional possível. Combinadores de interação são criados sobre redes de interação, que são baseadas em uma estrutura de nós, ligações entre os mesmos, e regras de interação entre as ligações, fazendo com que as interações possam ser resolvidas até que um resultado seja alcançado (Lafont, 1997). Por conta de sua arquitetura, redes de interação podem ser nativamente resolvidas em paralelo, onde várias interações podem ser calculadas ao mesmo tempo, sem interferir no restante da rede.

Bend foi criada como uma linguagem com sintaxe de alto nível e fácil entendimento e utiliza o compilador HVM2 para traduzir um código simples em operações que sejam executadas na GPU (Taelin, 2024). Apesar do lançamento recente, da ausência de alguns recursos básicos, e outras limitações, a linguagem demonstrou bons resultados em análises de performance segundo o fabricante, provando sua capacidade de executar códigos de alto nível nativamente em GPUs (Taelin, 2024).

2.2 TRABALHOS CORRELATOS

Nesta seção são apresentados trabalhos com características semelhantes aos principais objetivos do trabalho desenvolvido. O primeiro correlato (Quadro 1) detalha o trabalho de Garcia et al. (2008), no qual foi implementado com sucesso o algoritmo K-Nearest Neighbor de maneira paralela, utilizando o processamento de uma placa gráfica. O segundo trabalho (Quadro 2), de Hong-tao et al. (2009), aborda a implementação paralela do algoritmo K-Means, também com o uso de uma placa gráfica e da API CUDA. O terceiro trabalho (Quadro 3), de Nasridinov, Lee e Park (2014), demonstra a implementação do algoritmo de árvores de decisão de maneira paralela, utilizando como base uma abordagem híbrida entre CPU e GPU.

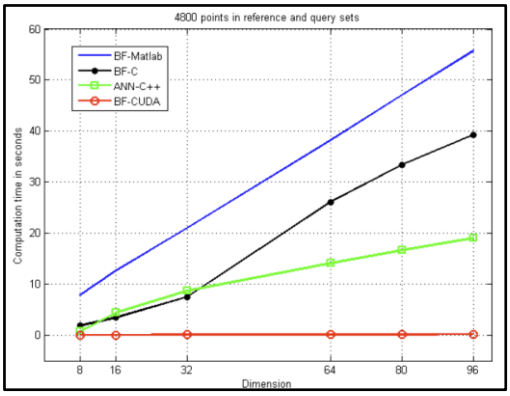
Quadro 1 - Fast k nearest neighbor search using GPU

Referência	Garcia *et al*. (2008)
Objetivos	Acelerar a execução do algoritmo KNN utilizando uma placa gráfica de uso geral.
Principais funcionalidades	Utilizar a API CUDA e uma arquitetura paralela para melhorar a performance do algoritmo K-Nearest Neighbor, principalmente da abordagem de força bruta do algoritmo. Além disso, demonstrar a eficiência da aplicação comparando-a a outras implementações disponíveis, tal como a encontrada no aplicativo MatLab e o algoritmo ANN.
Ferramentas de desenvolvimento	API CUDA, com o algoritmo implementado na linguagem C++.
Resultados e conclusões	Garcia *et al*. (2008) apresentaram resultados promissores com sua implementação em CUDA, obtendo um tempo de execução 407 vezes menor do que o algoritmo original do MatLab, e 148 vezes menor que a biblioteca ANN em C++. Além disso, a implementação baseada em placas gráficas demonstrou ser pouco afetada pela dimensionalidade dos dados de entrada, algo que aumenta consideravelmente o tempo de processamento nos demais algoritmos.

Fonte: elaborado pelo autor (2025).

Como observado, Garcia *et al*. (2008) implementaram com sucesso o algoritmo K-Nearest Neighbor de maneira paralela, utilizando como base a API CUDA. Garcia *et al*. (2008) tinha como objetivo testar a viabilidade de utilizar placas gráficas de propósito geral para executar o algoritmo KNN em uma base de dados, além de comparar seu tempo de execução com outras abordagens conhecidas. Para fins de comparação, foram utilizados os tempos de processamento de 3 outros métodos: a implementação do algoritmo encontrada no aplicativo MatLab, uma implementação na linguagem de programação C, e uma biblioteca em C++ chamada ANN, que, segundo Garcia *et al*. (2008), é um algoritmo baseado em aproximação conhecido por ser altamente otimizado. A performance alcançada pelo projeto superou em diversas vezes as demais bibliotecas, sendo inclusive mais eficiente que a implementação baseada em aproximação da biblioteca ANN (Figura 1).

Figura 1 - Comparação de diferentes implementações do KNN



Fonte: Garcia *et al.* (2008).

Quadro 2 - K-Means on Commodity GPUs with CUDA

Referência	Hong-tao *et al.* (2009)
Objetivos	Acelerar a execução do algoritmo K-Means utilizando uma placa gráfica de uso geral.
Principais funcionalidades	Utilizara API CUDA e uma arquitetura paralela para melhorar a performance do algoritmo K-Means, encarregando a placa gráfica de calcular as etapas com custo de processamento maior. Para isso, a implementação se comunica com a GPU para que esta trabalhe em duas partes cruciais do algoritmo: a atribuição da classe dos objetos-alvo, e o cálculo da média dos centroides. Os resultados também foram comparados com outras implementações disponíveis, a fim de validar sua eficiência.
Ferramentas de desenvolvimento	API CUDA.
Resultados e conclusões	Hong-tao *et al.* (2009) apresentaram resultados promissores com sua implementação em CUDA, demonstrando tempos de processamento 27 a 56 vezes menores quando comparados ao mesmo algoritmo executado inteiramente na CPU.

Fonte: elaborado pelo autor (2025).

Hong-tao et al. (2009) implementaram com sucesso o algoritmo K-Means de maneira paralela, utilizando como base a API CUDA. Hong-tao et al. (2009) focou em utilizar a performance da GPU para paralelizar o processamento de dois segmentos do algoritmo. A primeira etapa é a atribuição da classe dos objetos-alvo, onde cada objeto da base de dados tem sua distância calculada em relação a todos os centroides, e o mais perto determinará sua classe. O segundo segmento é o cálculo da média dos centroides, que considera os valores de todos os objetos daquela classe para resultar em uma nova posição do centroide. Os resultados do projeto também foram satisfatórios, demonstrando um aumento significativo na velocidade de processamento do algoritmo.

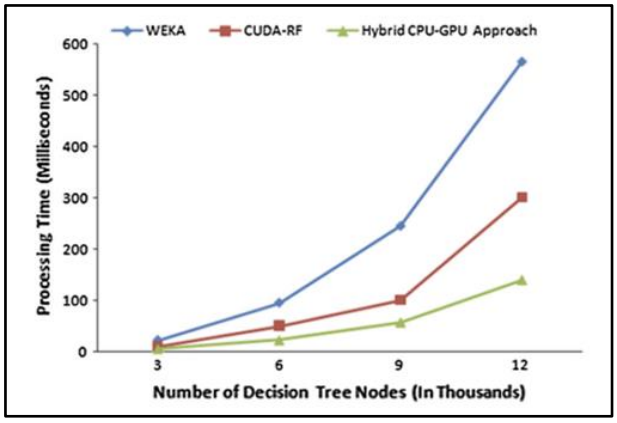
Quadro 3 – Decision tree construction on GPU: ubiquitous parallel computing approach.

Referência	Nasridinov; Lee; Park (2014)
Objetivos	Acelerar a execução do algoritmo de árvore de decisão utilizando uma abordagem híbrida, distribuindo a execução entre uma placa gráfica de uso geral e um processador.
Principais funcionalidades	Utilizar a capacidade de processamento das placas gráficas que, segundo Nasridinov, Lee e Park (2014), existem em todos os computadores pessoais, juntamente do processador, para acelerar a execução do algoritmo de árvores de decisão. Também foram comparados o tempo de processamento e o gasto de energia de diferentes implementações do algoritmo, buscando validar sua eficácia.
Ferramentas de desenvolvimento	API CUDA.
Resultados e conclusões	Os resultados foram comparados com outras duas implementações do algoritmo: uma implementação completamente no processador, chamada Weka, e uma completamente na placa gráfica. O tempo de processamento do algoritmo projetado foi aproximadamente 5 vezes menor que a biblioteca Weka, e 1,5 vezes menor que a implementação baseada puramente na placa gráfica.

Fonte: elaborado pelo autor (2025).

O trabalho de Nasridinov, Lee e Park (2014) se propôs a utilizar placas gráficas para, em conjunto com processadores, melhorar a performance do algoritmo de árvores de decisão. Nasridinov, Lee e Park (2014) promoveram uma arquitetura onde as etapas do algoritmo são divididas entre a CPU e a GPU, aproveitando a especialidade de cada componente para conduzir uma parte do processo. Os resultados apresentados, tanto em tempo de processamento quanto em gasto de energia, foram satisfatórios (Figura 2). Comparando o método proposto com a biblioteca Weka, que oferece o mesmo algoritmo, houve uma redução de até 5 vezes no tempo de processamento.

Figura 2 - Comparação de diferentes implementações de árvores de decisão



Fonte: Nasridinov, Lee e Park (2014).

Diante do cenário apresentado, pode-se observar que a utilização de placas gráficas para acelerar a execução de diversos algoritmos de aprendizado de máquina é um tema recorrente, provando a viabilidade de paralelizar a execução de tais processos. No entanto, ainda não foi testada a capacidade da linguagem de programação Bend de implementar tais algoritmos, fazendo uso da capacidade nativa da linguagem de paralelizar cálculos através da GPU.

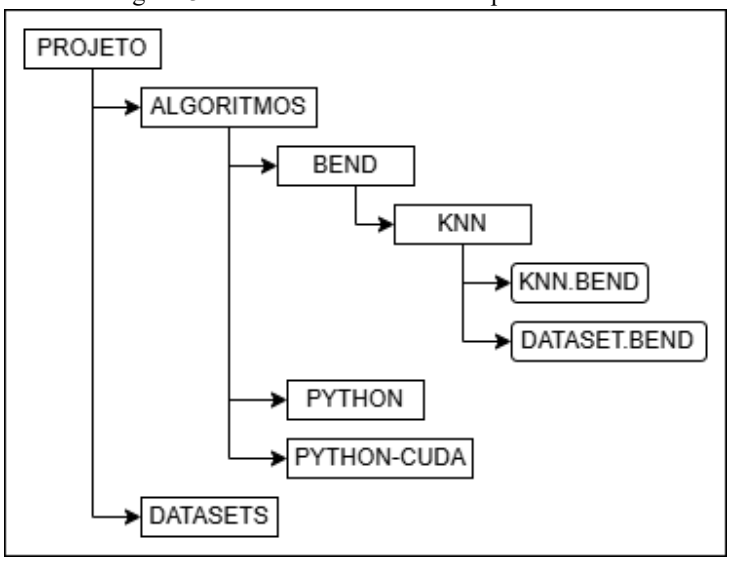
3 DESCRIÇÃO DA IMPLEMENTAÇÃO

Esta seção está dividida em dois segmentos. O primeiro descreve a especificação geral do trabalho, e o segundo explica como todas as partes do trabalho foram implementadas, com destaque principal ao código criado para os algoritmos na linguagem Bend.

3.1 ESPECIFICAÇÃO

A Figura 3 demonstra como está organizada a estrutura de pastas de todos os arquivos utilizados no trabalho. A pasta de algoritmos contém todos os arquivos executáveis que foram criados durante a implementação, sejam eles para a obtenção de resultados finais para comparação entre as diferentes linguagens, estudo dos algoritmos, ou criação de datasets. A pasta de datasets comporta os datasets criados artificialmente que foram utilizados como base para os três algoritmos, em todas as linguagens exploradas.

Figura 3 - Estrutura do trabalho implementado



Fonte: elaborado pelo autor (2025)

3.2 IMPLEMENTAÇÃO

Esta seção explicita como foram implementados os algoritmos utilizados neste trabalho, em todas as linguagens analisadas.

3.2.1 Python

A primeira linguagem utilizada para a implementação dos algoritmos foi Python. Esta parte do trabalho teve como objetivo o estudo e maior entendimento da estrutura e funcionamento dos algoritmos KNN, K-Means e árvores de decisão. Primeiramente, a biblioteca Scikit-Learn foi utilizada para construir versões executáveis simples desses algoritmos, que serviram como referência para comparar os resultados obtidos com a versão desenvolvida manualmente.

Após utilizar a biblioteca para obter resultados de comparação, todos os algoritmos foram implementados manualmente, sem a utilização de bibliotecas externas à linguagem. Isto forneceu um entendimento mais profundo do fluxo geral de cada algoritmo, o que se mostrou indispensável para as próximas etapas do trabalho.

3.2.2 Datasets

Como alvo para execução de todos os algoritmos que seriam desenvolvidos, datasets artificiais foram criados utilizando a função `make_blobs` da biblioteca Scikit-Learn, em Python. Esta função cria pontos em N dimensões baseado nas configurações fornecidas pelo usuário. Considerando que todos os algoritmos deste trabalho podem ser usados para classificar e agrupar pontos no espaço que tenham uma classe relacionada, esta função se demonstrou muito eficaz pela simplicidade e capacidade de parametrização para gerar tais pontos.

Quadro 4 - Função para criação de datasets

```
X, y = make_blobs(
    n_samples=30000,
    n_features=10,
    centers=5,
    cluster_std=1.5,
    random_state=42
)
```

Fonte: elaborado pelo autor (2025).

Como pode ser visto no Quadro 4, a função permite que a geração dos pontos seja parametrizada de algumas maneiras diferentes:

a) `n_samples`: quantos pontos devem ser criados;
b) `n_features`: quantos atributos cada ponto deve ter;
c) `centers`: ao redor de quantos centros os pontos devem se acumular. Também representa o número total de classes no dataset;
d) `cluster_std`: a taxa de desvio padrão dos centros, que representa quão longe do seu centro os pontos podem ser criados;
e) `random_state`: define a semente para o gerador aleatório de números, garantindo que os resultados de diversas execuções sejam sempre os mesmos.

Com os dados criados pela função, alguns pontos foram aleatoriamente separados para serem usados como teste, e ambas as listas de pontos foram salvas em arquivos de texto.

3.2.3 Bend

Nesta subseção, serão detalhadas as implementações em Bend dos três algoritmos que esse trabalho se propôs a avaliar: KNN, K-Means e árvores de decisão.

3.2.3.1 KNN

O algoritmo K-Nearest Neighbors se baseia em calcular a distância entre um ponto de teste e todos os pontos de treino de um dataset, ordená-los pela distância, e decidir pela maioria dos K pontos mais próximos qual é a classe resultante mais provável para o ponto de teste. Trata-se de um algoritmo simples, porém com custo computacional elevado, pois exige que todas as distâncias sejam calculada a cada nova predição.

Na implementação em Bend, a estrutura de dados mais utilizada foi a lista encadeada, que é a lista padrão oferecida pela linguagem. Por conta disso, espera-se que esta implementação não faça tanto proveito de um processamento paralelo em múltiplos núcleos, levando em conta a natureza sequencial de listas encadeadas, que, ao contrário de árvores e vetores, não pode ser paralelizada.

Para o KNN, pontos foram definidos como objetos com duas variáveis:

a) *features*, construída em uma lista encadeada de *f24,* representando os valores de cada dimensão do ponto;
b) *label*, construída em um único valor *u24,* representando a classe do ponto.

Neste contexto, *f24* e *u24* representam tipos de dados, sendo eles, respectivamente: um dado numérico de ponto flutuante com 24 bits de precisão, e um inteiro sem sinal de 24 bits.

Como mostra o Quadro 5, o método de entrada do algoritmo recebe uma lista de pontos de treinamento, uma lista de pontos de teste, e um valor K, que será usado na contagem de vizinhos mais próximos. A partir desse método, a lista de pontos de teste é percorrida recursivamente de maneira sequencial, um ponto por vez, para obter a classe mais provável para aquele ponto, e a lista de retorno é continuamente construída com os resultados das predições.

Quadro 5 - Função inicial do algoritmo KNN

```
1   def predict(points_train: List(Point),
2               points_test: List(Point), k: u24) -> List(u24):
3     match points_test:
4       case List/Nil:
5         return List/Nil
6       case List/Cons:
7         return List/Cons{head: predict_single(points_train, points_test.head, k),
8                          tail: predict(points_train, points_test.tail, k)}
9
10  def predict_single(points_train: List(Point), point: Point, k: u24) -> u24:
11    top_k = calc_distances_k(points_train, point, k)
12    return get_majority(top_k)
```

Fonte: elaborado pelo autor (2025).

A função `predict_single` é a encarregada de prever a classe de um ponto de teste, utilizando uma lista de pontos como treinamento. Para isso, é necessário calcular a distância entre o ponto de treino e todos os pontos de teste, ordená-los pelo resultado, e considerar apenas os K pontos de treino mais próximos ao ponto de teste. Este processo pode ser feito de maneira simples, acumulando todas as distâncias em uma lista, ordenando-a ao fim do processo e selecionando as K primeiras. Apesar disso, uma abordagem mais eficiente envolve manter na lista apenas as K distâncias mais próximas, e inserir novas entradas na lista de maneira já ordenada, evitando assim que todas as distâncias fiquem salvas na memória de execução do programa, o que foi uma limitação encontrada na implementação deste algoritmo.

Quadro 6 - Cálculo das K distâncias mais próximas

```
1   def calc_distances_k(points_train: List(Point),
2                        point: Point, k: u24) -> List(MeasuredPoint):
3     (list_length, list) = List/length(points_train)
4     bend top_k = List/Nil, count = 0, current = points_train:
5       when count < list_length:
6         match current:
7           case List/Nil:
8             return top_k
9           case List/Cons:
10            new_distance = MeasuredPoint { point: current.head,
11                                           distance: calc_distance(current.head, point) }
12            top_k = insert_top_k(top_k, new_distance, k)
13            return fork(top_k, count + 1, current.tail)
14      else:
15        return top_k
```

Fonte: elaborado pelo autor (2025).

O Quadro 6 demonstra como o algoritmo calcula as K distâncias mais próximas. Para isso, é utilizada a estrutura de controle de fluxo `bend`. Essa estrutura é reduzida a uma simples recursão, mas ela facilita a construção de um resultado ao utilizar acumuladores, que são inicializados na linha 4 e repassados na chamada de `fork`, na linha 13. Com essa estrutura, não é necessário passar tais acumuladores na assinatura da própria função, nem criar funções auxiliares que seriam utilizadas apenas na recursão.

O cálculo de distância utilizado nesse algoritmo é o de distância euclidiana, que mede a separação linear entre dois pontos no espaço. Este cálculo foi implementado através de uma função recursiva que percorre todas as dimensões de dois pontos, soma o quadrado das diferenças entre os pares de valores, e retorna a raiz quadrada do total.

Com a lista dos K pontos de treino mais próximos ao ponto de teste, é necessário calcular qual classe possui presença majoritária entre os pontos. Para isso, a lista é percorrida e é mantido controle das diferentes classes e quantas vezes cada uma delas aparecem na listagem, retornando ao final a classe que possuir a maior contagem. Nesta implementação, a contagem de classes foi feita através da criação de uma árvore, sendo que cada nó da árvore armazena as informações de qual classe está sendo contada, quantas vezes aquela classe apareceu na lista, e seus nós filhos, a esquerda e a direita do nó pai.

Como apresenta o Quadro 7, a função `get_majority` recebe a lista dos K pontos mais próximos, cria uma árvore com as classes e quantas vezes elas estão contidas na lista, e retorna a classe que possui a maior contagem. Para encontrar a classe majoritária, a árvore é percorrida pela função `find_max`, reduzindo os nós da árvore recursivamente utilizando a função `max`, que retorna o maior valor entre os passados como argumento para a função.

Quadro 7 - Contagem da maioria entre o K pontos mais próximos

```
1   def get_majority(points: List(MeasuredPoint)) -> u24:
2     label_tree = build_label_tree(points, LabelTree/Nil)
3     (max_label, max_value) = find_max(label_tree)
4     return max_label
5
6   def build_label_tree(points: List(MeasuredPoint), label_tree: LabelTree) ->
7   LabelTree:
8     match points:
9       case List/Nil:
10        return label_tree
11      case List/Cons:
12        open MeasuredPoint: points.head
13        open Point: points.head.point
14        return build_label_tree(points.tail, insert_label_into(label_tree,
15  points.head.point.label))
16
17  def find_max(label_tree: LabelTree) -> (u24, u24):
18    match label_tree:
19      case LabelTree/Nil:
20        return (0, 0)
21      case LabelTree/Node:
22        max_label_left, max_count_left = find_max(label_tree.left)
23        max_label_right, max_count_right = find_max(label_tree.right)
24        return max(label_tree.label, label_tree.count, max_label_left,
25                   max_count_left, max_label_right, max_count_right)
```

Fonte: elaborado pelo autor (2025).

Com a classe resultante calculada, o processamento chega ao fim para esse ponto de teste. A classe prevista do ponto é acumulada na lista geral do algoritmo e o próximo ponto começa a ser calculado. Desta maneira, o resultado final do algoritmo é uma lista de valores *u24*, representando a classe de todos os pontos de teste calculados, na mesma ordem em que eles foram fornecidos. Como dito anteriormente, pela implementação ser baseada em listas encadeadas, o processamento tem pouca capacidade de paralelização, o que não é o ideal para esse contexto.

3.2.3.2 K-Means

Diferente dos outros algoritmos deste trabalho, o K-Means não é um algoritmo de classificação, mas sim de agrupamento. Este algoritmo se baseia em utilizar centróides, que servem como núcleos para os pontos, para encontrar semelhanças entre os pontos, e atribuir uma classe a todos os pontos que sejam similares espacialmente. Para isso, o algoritmo possui duas etapas, que podem ser repetidas N vezes: atribuir os pontos ao centróide mais próximo, e recalcular a posição do centróide.

Com o objetivo de tornar o algoritmo mais eficiente e permitir que ele seja processado de maneira paralela, essa implementação utilizou uma árvore de pontos como base para a implementação. Cada nó da árvore armazena a informação de um ponto e de dois nós filhos, um à esquerda e outro à direita. Um ponto é definido da mesma maneira que o algoritmo anterior: uma lista encadeada de valores decimais representando as *features*, e um inteiro representando o *centroid_label*.

Conforme apresentado no Quadro 8, o fluxo principal do algoritmo utiliza recursão para repetir N vezes as principais funções do algoritmo, que realizam a atribuição dos pontos e a realocação dos centróides. O método recebe a árvore de pontos como entrada, além de uma lista de pontos que são usados como centróides iniciais. Também são usadas variáveis de controle `count` e `max_count` para limitar o máximo de repetições do algoritmo. Ao fim da execução, é retornada a árvore de pontos, com suas classes já atribuídas, e a lista de centróides com suas posições recalculadas.

Quadro 8 - Função inicial do algoritmo K-Means

```
1   def fit(point_tree: PointTree, centroids: List(Point),
2           count: u24, max_count: u24) -> (PointTree, List(Point)):
3     if count < max_count:
4       new_point_tree = allocate_points(point_tree, centroids)
5       new_centroids = relocate_centroids(new_point_tree, centroids)
6       return fit(new_point_tree, new_centroids, count+1, max_count)
7     else:
8       return (point_tree, centroids)
```

Fonte: elaborado pelo autor (2025).

A função `allocate_points` é a primeira no fluxo de agrupamento do algoritmo. Conforme detalhado no Quadro 9, os pontos da árvore são percorridos recursivamente, apenas substituindo a classificação do ponto conforme o centróide mais próximo encontrado. Por operar sob uma árvore, essa função pode ser paralelizada, onde ambos os filhos esquerdo e direito de um nó são processados ao mesmo tempo e de maneira independente.

Quadro 9 - Função para atribuição dos pontos

```
1   def allocate_points(pointTree: PointTree, centroids: List(Point)) -> PointTree:
2     match pointTree:
3       case PointTree/Nil:
4         return pointTree
5       case PointTree/Node:
6         point = pointTree.point
7         open Point: point
8         (closest_centroid_label, closest_centroid_distance) = find_closest(point,
9                                                                            centroids)
10        return PointTree/Node {point: Point {features: point.features,
11                                             centroid_label: closest_centroid_label},
12                               left: allocate_points(pointTree.left, centroids),
13                               right: allocate_points(pointTree.right, centroids)}
14
15  def find_closest(point: Point, centroids: List(Point)) -> (u24, f24):
16    match centroids:
17      case List/Nil:
18        return (9999999, 999999999.9)
19      case List/Cons:
20        centroid = centroids.head
21        open Point: centroid
22        distance = calc_distance(point, centroid)
23        (next_closest_label, next_closest_distance) = find_closest(point,
24                                                                   centroids.tail)
25        return min(centroid.centroid_label, distance, next_closest_label,
26                   next_closest_distance)
```

Fonte: elaborado pelo autor (2025).

Para encontrar o centróide mais próximo de um ponto, o método `find_closest` percorre a lista de centróides, calculando a distância entre cada centróide analisado e o ponto em questão. A cada passo da recursão, a função `min` é utilizada para comparar a menor distância encontrada até o momento com a distância do centróide atual, determinando assim qual é o mais próximo.

Após os pontos serem atribuídos aos seus centróides, o algoritmo recalcula a posição ótima dos centróides, baseado na média da posição de todos os pontos relacionados a si. Para isso, o algoritmo recursivamente cria um mapa que contém as *features* de cada ponto da árvore, utilizando a classe do centróide a qual o ponto está relacionado como chave desse mapa, conforme dita o Quadro 10. Após ter os objetos criados, os mapas são fundidos entre o nó pai e seus nós filhos, acumulando assim todas as *features* de cada centróide, mantendo ainda a contagem de quantos pontos foram encontrados em cada grupamento.

Quadro 10 - Função para realocar os centróides

```
def relocate_centroids(pointTree: PointTree,
                       centroids: List(Point)) -> List(Point):
  sums_map = create_sums_map(pointTree, centroids)
  return apply_new_averages(sums_map, centroids)

def create_sums_map(pointTree: PointTree,
                    centroids: List(Point)) -> Map(FeaturesSum):
  match pointTree:
    case PointTree/Nil:
      return Map/empty()
    case PointTree/Node:
      point = pointTree.point
      open Point: point
      current_sums_map = {point.centroid_label:
                                        FeaturesSum {features_sum:point.features,
                                                     count: 1}}
      left_sums_map = create_sums_map(pointTree.left, centroids)
      right_sums_map = create_sums_map(pointTree.right, centroids)
      left_right = sum_maps(left_sums_map, right_sums_map, centroids)
      return sum_maps(left_right, current_sums_map, centroids)
```

Fonte: elaborado pelo autor (2025).

Com o mapa de *features* criado, a função `apply_new_averages` é encarregada de calcular a média dos valores somados e atribuir o resultado aos centróides, reposicionando-os no espaço. Como demonstra o Quadro 11, a estrutura de controle `bend` é utilizada para recriar a lista de centróides, recuperando suas *features* através da função `divide_list_by_count`, que calcula a média de todos os valores encontrados no mapa pela contagem de pontos salva.

Quadro 11 - Função para aplicar novas posições aos centróides

```
def apply_new_averages(sums_map: Map(FeaturesSum),
                       centroids: List(Point)) -> List(Point):
  (list_length, list) = List/length(centroids)
  bend result = List/Nil, count = 0, current = centroids:
    when count < list_length:
      match current:
        case List/Nil:
          return result
        case List/Cons:
          centroid = current.head
          open Point: centroid
          features_sum = sums_map[centroid.centroid_label]
          open FeaturesSum: features_sum
          new_values = divide_list_by_count(features_sum.features_sum,
                                            features_sum.count)
          return fork(List/Cons {head: Point {features: new_values,
                                              centroid_label: centroid.centroid_label},
                                 tail: result},
                      count + 1,
                      current.tail)
    else:
      return result
```

Fonte: elaborado pelo autor (2025).

Com os pontos atribuídos e os centróides reposicionados, o processo se repete até que o limite de iterações seja alcançado. Dessa maneira, a cada iteração do algoritmo, os centróides se encontram mais perto de sua posição real. Como resultado, a função `fit` retorna tanto a árvore de pontos, que podem ter suas classes verificadas, quanto a lista de centróides, que podem ter suas posições extraídas para determinar o ponto central de cada agrupamento.

Por utilizar árvores ao invés de listas encadeadas como estrutura básica, este algoritmo deve apresentar resultados ainda melhores quando executado de forma paralela. Apesar disso, as iterações da função `fit` ainda devem ocorrer de forma sequencial, uma vez que cada etapa depende do resultado da anterior, o que impede que esse processo seja paralelizado.

3.2.3.3 Árvores de decisão

Este algoritmo é caracterizado pela construção de uma estrutura em formato de árvore, onde cada nó interno representa uma decisão baseada em um atributo dos dados de entrada, e cada folha representa uma classe de saída. Ao

contrário dos outros algoritmos implementados neste trabalho, as árvores de decisão criam um modelo a partir dos dados de treinamento, e este modelo pode ser reutilizado para todos os pontos de teste. Dessa maneira, a criação do modelo é custosa em termos de processamento, mas a predição de novos dados de teste é barata.

Na implementação deste algoritmo em Bend, a estrutura de dados utilizada como base para o treinamento foi uma árvore binária, cujos nós internos não armazenam informações, servindo apenas como estruturas para as folhas. O modelo resultante do treinamento é uma árvore com nós valorados, onde cada nó interno representa uma decisão baseada em um atributo do conjunto de dados. Cada decisão consiste na definição de um valor limiar para determinado atributo, ditando se o dado de entrada deverá seguir para o galho esquerdo ou direito da árvore, até que seja alcançada uma folha, que determina a classificação final do ponto de teste.

Também diferente das outras implementações deste trabalho, este algoritmo utilizou um mapa para armazenar os valores de cada atributo de um ponto. Em Bend, mapas são árvores binárias, e possuem métodos auxiliares para verificar a existência e acessar valores baseados em uma chave. Com o objetivo de facilitar a implementação, os índices de cada atributo do dataset, que são usados como chaves no mapa de cada ponto, são informados explicitamente no início da aplicação.

Para criar decisões durante o treinamento do algoritmo, o conceito de pureza é utilizado para determinar se um grupo de pontos deve ser dividido ou se todos já fazem parte da mesma classe. Nesta implementação, o cálculo de pureza utilizado foi o de *gini impurity*. Essa metodologia calcula a probabilidade de um elemento ser classificado incorretamente caso ele seja rotulado de maneira aleatória. Desta maneira, conjuntos de dados com impureza 0 terão apenas uma classificação possível. O Quadro 12 demonstra como este cálculo foi implementado.

Quadro 12 - Cálculo de impureza Gini

```
1   def calc_gini_impurity(point_tree: Tree(Point)) -> f24:
2     total = length_of(point_tree)
3     (labels_count, found_labels) = create_labels_count(point_tree)
4     (labels_num, _) = List/length(found_labels)
5
6     impurity_sum = 0.0
7     fold found_labels with impurity_sum:
8       case List/Nil:
9         return (1.0 - impurity_sum)
10      case List/Cons:
11        proportion = u24/to_f24(labels_count[found_labels.head])/ u24/to_f24(total)
12        new_value = proportion ** 2.0
13        return found_labels.tail((impurity_sum + new_value))
14
15  def create_labels_count(point_tree: Tree(Point)) -> (Map(u24), List(u24)):
16    fold point_tree:
17      case Tree/Leaf:
18        point = point_tree.value
19        open Point: point
20        return ({point.label: 1}, [point.label])
21      case Tree/Node:
22        (map_left, keys_left) = point_tree.left()
23        (map_right, keys_right) = point_tree.right()
24        return merge_maps(map_left, keys_left, map_right, keys_right)
```

Fonte: elaborado pelo autor (2025).

Como detalhado no Quadro 13, a função de treinamento do modelo, `grow_tree`, recebe uma árvore de pontos de teste e a lista de índices de todos os atributos de um ponto, além da profundidade máxima e atual da árvore treinada. Caso a árvore de decisões tenha alcançado sua profundidade máxima, ou a árvore de pontos recebida seja pura, o crescimento daquele galho é encerrado, e é retornada uma folha para a árvore, demonstrando que a linha de decisões levou a uma classificação final.

Quadro 13 - Função de treinamento da árvore de decisão

```
1   def grow_tree(point_tree: Tree(Point), every_feature_indexes: List(u24),
2                 max_depth: u24, current_depth: u24) -> DecisionTree:
3     if current_depth >= max_depth | calc_gini_impurity(point_tree) == 0.0:
4       (labels_count, found_labels) = create_labels_count(point_tree)
5       (major_label, major_count) = find_majority(labels_count, found_labels)
6       return DecisionTree/Leaf {prediction: major_label}
7     else:
8       (best_impurity, best_feature_index, best_threshold) =
9                                   find_best_split(point_tree, every_feature_indexes)
10      (maybe_left_tree, maybe_right_tree) =
11                        split_points(point_tree, best_feature_index, best_threshold)
12      left_branch = grow_tree(Maybe/unwrap(maybe_left_tree), every_feature_indexes,
13                                                          max_depth, current_depth + 1)
14      right_branch = grow_tree(Maybe/unwrap(maybe_right_tree),
15                               every_feature_indexes, max_depth, current_depth + 1)
16      return DecisionTree/Node {feature_index: best_feature_index,
17                                threshold: best_threshold,
18                                left: left_branch,
19                                right: right_branch}
```

Fonte: elaborado pelo autor (2025).

Caso não seja atendida nenhuma condição para criar uma folha, o algoritmo segue a procura por outro nó de decisão, utilizando a função `find_best_split`. Este método, juntamente do `find_best_split_feature`, percorre todos os atributos de todos os pontos da árvore, e separa a árvore em duas, se baseando no valor observado. A partir da separação, a pureza das árvores é calculada, levando em conta o peso de cada árvore. Após esse cálculo ser feito a partir de todos os pontos, o melhor resultado encontrado é levado como próxima decisão da árvore, retornando ao método inicial, separando os pontos restantes baseado no limiar encontrado e prosseguindo com o crescimento da árvore. Esse funcionamento é explicitado no Quadro 14.

Quadro 14 - Função para procurar a próxima decisão da árvore

```
1   def find_best_split(point_tree: Tree(Point),
2                       every_feature_indexes: List(u24)) -> (f24, u24, f24):
3     fold every_feature_indexes:
4       case List/Nil:
5         return (1.0, 9, 0.0)
6       case List/Cons:
7         (current_impurity, current_feature_index, current_threshold) =
8                     find_best_split_feature(point_tree, every_feature_indexes.head)
9         (next_impurity, next_feature_index, next_threshold) =
10                                                      every_feature_indexes.tail()
11        if current_impurity <= next_impurity:
12          return (current_impurity, current_feature_index, current_threshold)
13        else:
14          return (next_impurity, next_feature_index, next_threshold)
15
16  def find_best_split_feature(point_tree: Tree(Point),
17                              feature_index: u24) -> (f24, u24, f24):
18    full_tree = point_tree
19    fold point_tree:
20      case Tree/Leaf:
21        current_point = point_tree.value
22        open Point: current_point
23        (maybe_left_tree, maybe_right_tree) = split_points(full_tree,
24                              feature_index, current_point.features[feature_index])
25        impurity = check_trees_weight(maybe_left_tree, maybe_right_tree)
26        return (impurity, feature_index, current_point.features[feature_index])
27      case Tree/Node:
28        (left_impurity, left_feature_index, left_threshold) = point_tree.left()
29        (right_impurity, right_feature_index, right_threshold) = point_tree.right()
30        if left_impurity <= right_impurity:
31          return (left_impurity, left_feature_index, left_threshold)
32        else:
33          return (right_impurity, right_feature_index, right_threshold)
```

Fonte: elaborado pelo autor (2025).

Quando a árvore de decisão chegar em sua profundidade máxima, ou todos os pontos forem completamente separados em diferentes folhas, o treinamento do modelo terá finalizado, e um objeto `DecisionTree` é retornado como

resultado. Com base nesse objeto, o método `predict` recebe uma árvore de pontos de teste, e retorna a mesma árvore com os pontos reclassificados segundo a árvore de decisões. Conforme apresentado no Quadro 15, a função de predição percorre paralelamente todos os pontos da árvore, e invoca a função `predict_single` para classificar um ponto por vez.

Quadro 15 - Função de predição dos pontos de teste

```
1   def predict(test_point_tree: Tree(Point), decision_tree: DecisionTree) ->
2   Tree(Point):
3     fold test_point_tree:
4       case Tree/Leaf:
5         current_point = test_point_tree.value
6         open Point: current_point
7         return !Point {features: current_point.features, label:
8   predict_single(current_point, decision_tree)}
9       case Tree/Node:
10        return ![test_point_tree.left(), test_point_tree.right()]
11
12  def predict_single(point: Point, decision_tree: DecisionTree) -> u24:
13    fold decision_tree:
14      case DecisionTree/Leaf:
15        return decision_tree.prediction
16      case DecisionTree/Node:
17        open Point: point
18        if point.features[decision_tree.feature_index] <= decision_tree.threshold:
19          return decision_tree.left()
20        else:
21          return decision_tree.right()
```

Fonte: elaborado pelo autor (2025).

Esta segunda função utiliza dos atributos do ponto para percorrer a árvore de decisões, direcionando o caminho à esquerda ou direita conforme o resultado da comparação entre o atributo do ponto e o limiar armazenado no nó atual. Esse processo é repetido até que uma folha seja alcançada, momento em que o valor contido na folha é atribuído à classificação do ponto.

Por utilizar árvores em toda a estrutura do algoritmo, inclusive no armazenamento dos atributos através de mapas, esta implementação deve obter o maior aumento de performance quando executada de maneira paralela.

3.2.4 Python-CUDA

Como principal objeto de comparação com as versões desenvolvidas em Bend, os três algoritmos foram também escritos em Python, mas desta vez utilizando as bibliotecas CuPy e CuML. Estas bibliotecas oferecem implementações nativas de todos os algoritmos abordados neste trabalho, executadas diretamente em GPUs através da API CUDA. Desta forma, optou-se por esta linguagem e bibliotecas devido à facilidade tanto no uso quanto na obtenção de resultados, reduzindo a chance de erros humanos comprometerem a imparcialidade da comparação.

Foram utilizados os mesmos datasets previamente adotados em Bend, com o apoio de uma função para realizar a leitura dos arquivos. Para cronometrar com precisão o tempo de execução dos algoritmos, foram usados eventos CUDA para registrar o início e fim das execuções, fazendo com que apenas as operações na placa de vídeo fossem consideradas no tempo final. Os quadrosQuadro 16, Quadro 17 eQuadro 18 demonstram como os algoritmos oferecidos pela biblioteca CuPy foram utilizados.

Quadro 16 - Implementação de KNN pela biblioteca CuPy

```
1   X_train, y_train = load_data('datasets/knn/train.txt')
2   X_test, y_test = load_data('datasets/knn/test.txt')
3
4   knn = KNeighborsClassifier(n_neighbors=5)
5
6   start = cupy.cuda.Event()
7   end = cupy.cuda.Event()
8   start.record()
9
10  knn.fit(X_train, y_train)
11  predictions = knn.predict(X_test)
12
13  end.record()
14  end.synchronize()
15  gpu_time_ms = cupy.cuda.get_elapsed_time(start, end)
```

Fonte: elaborado pelo autor (2025).

Quadro 17 - Implementação de K-Means pela biblioteca CuPy

```
X_test, y_test = load_data('datasets/kmeans/test.txt')
centroids, _ = load_data('datasets/kmeans/centroids.txt')

kmeans = KMeans(
    n_clusters=centroids.shape[0],
    init=centroids,
    max_iter=10
)

start = cupy.cuda.Event()
end = cupy.cuda.Event()
start.record()

kmeans.fit(X_test)
predictions = kmeans.predict(X_test)

end.record()
end.synchronize()
gpu_time_ms = cupy.cuda.get_elapsed_time(start, end)
```

Fonte: elaborado pelo autor (2025).

Quadro 18 - Implementação de árvores de decisão pela biblioteca CuPy

```
X_train, y_train = load_data('datasets/decision_trees/train.txt')
X_test, y_test = load_data('datasets/decision_trees/test.txt')

tree = cuRFC(max_features=10,
             n_estimators=1,
             split_criterion='gini',
             bootstrap=False,
             max_depth=10)

start = cupy.cuda.Event()
end = cupy.cuda.Event()
start.record()

tree.fit(X_train, y_train)
predictions = tree.predict(X_test)

end.record()
end.synchronize()
gpu_time_ms = cupy.cuda.get_elapsed_time(start, end)
```

Fonte: elaborado pelo autor (2025).

4 RESULTADOS

A partir das implementações desenvolvidas, os algoritmos foram executados sobre os conjuntos de dados previamente gerados, e seus respectivos tempos de processamento foram registrados. Para a execução dos algoritmos, foi utilizado um servidor virtualizado que possuía uma placa de vídeo NVIDIA RTX4090 e um processador AMD EPYC de 64 núcleos. Para cada algoritmo, as seguintes implementações foram executadas:

a) bend run-rs: implementação em Bend, utilizando o compilador em Rust, que não possui processamento paralelo;
b) bend run-c: implementação em Bend, utilizando o compilador em C, que possui processamento paralelo apenas entre os núcleos da CPU;
c) bend run-cu: implementação em Bend, utilizando o compilador CUDA, que possui processamento paralelo na GPU;
d) python-lib: implementação em Python, utilizando a biblioteca Scikit-learn para executar os algoritmos na CPU;
e) python-impl: implementação em Python, desenvolvida manualmente sem a utilização de bibliotecas externas, que não possui processamento paralelo;
f) python-cuda: implementação em Python, utilizando as bibliotecas CuPy e CuML, que possui processamento paralelo na GPU.

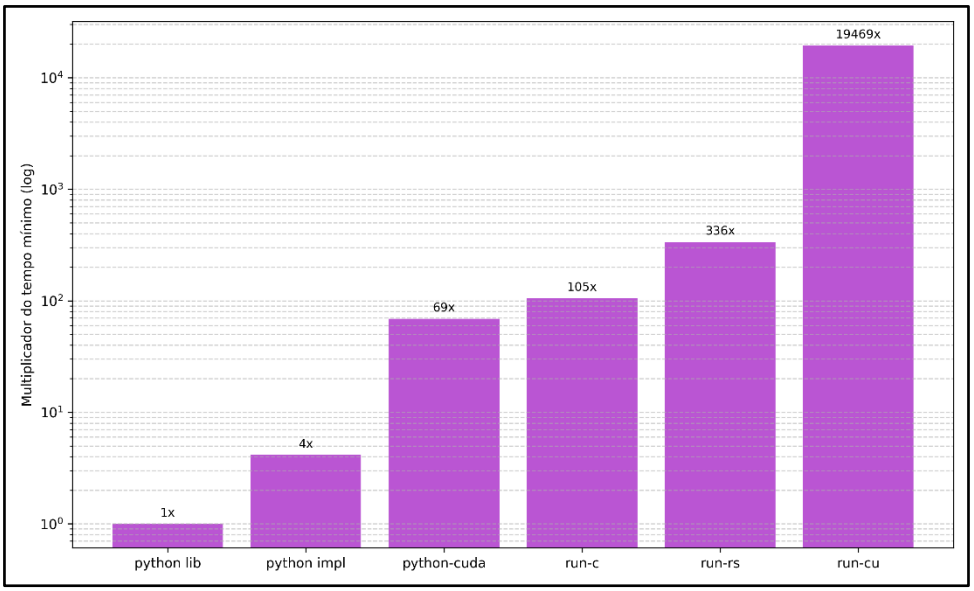
Nas implementações em Bend, embora todos os algoritmos tenham funcionado corretamente quando compilados em C, suas execuções em CUDA apresentaram diversos erros, principalmente relacionados à memória disponível. Por conta dessas limitações, só foi possível executá-los na GPU por meio do Bend quando os datasets continham uma

quantidade reduzida de pontos a serem processados. Considerando essa restrição, cada algoritmo foi avaliado com dois tamanhos de dataset: um conjunto menor, que permitiu a execução em todas as implementações, inclusive na GPU com Bend, e um conjunto maior, no qual foram desconsideradas as execuções na GPU com Bend.

Além disso, a partir de certa quantidade de pontos, todas as execuções em Bend apresentavam problemas que corrompiam o retorno de resultados, tornando inviável a execução dessas implementações. Por conta disso, os datasets para KNN e K-Means foram limitados a 30 mil pontos de treinamento, e 1300 pontos para o algoritmo de árvores de decisão.

O primeiro algoritmo analisado foi o K-Nearest Neighbors, utilizando um dataset com 100 pontos de treinamento e 100 pontos de teste. A Figura 4 apresenta a relação entre os tempos de execução das diferentes implementações. As versões em Python se mostraram mais eficientes que todas as implementações em Bend, sendo que o menor tempo registrado foi da biblioteca Scikit-learn. Apesar de contar com a capacidade de processamento da GPU, a implementação da biblioteca CuML apresentou um tempo de execução 69 vezes maior que a o da biblioteca Scikit-learn, sendo inclusive mais lenta que a implementação própria do autor. Esse resultado provavelmente se deve ao pequeno número de pontos no dataset, o que faz com que o *overhead* proveniente da utilização da GPU seja maior que os ganhos proporcionados pelo paralelismo. Ao analisar a performance das execuções em Bend, é visível o aumento de performance do processamento paralelo na CPU, mesmo utilizando poucos núcleos. Por outro lado, a execução por meio do compilador CUDA apresentou um desempenho muito inferior, sendo 20 mil vezes mais lenta que a versão da biblioteca Scikit-learn, e quase 200 vezes mais lenta que a versão compilada em C.

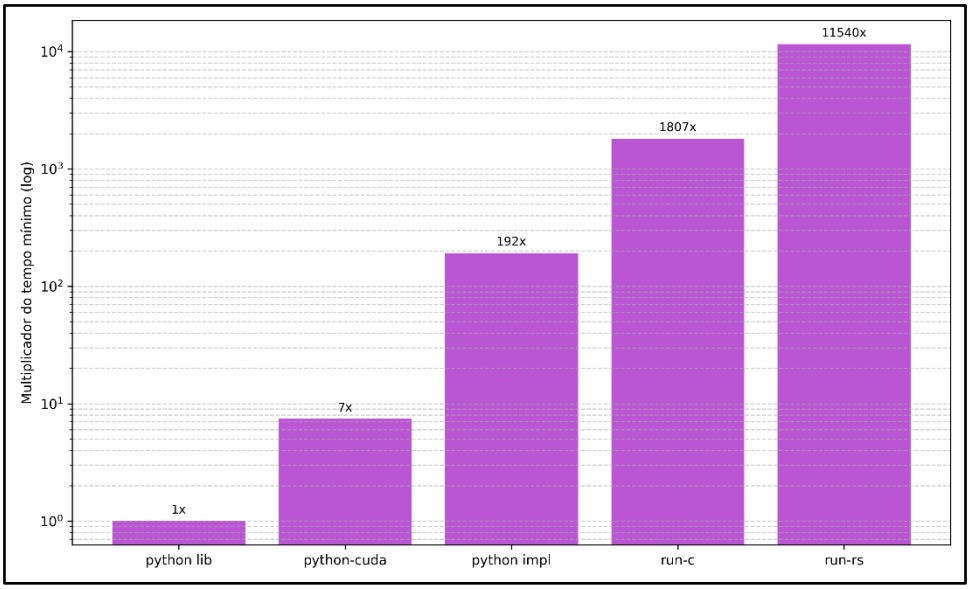
Figura 4 - Resultados do algoritmo KNN com o menor dataset



Fonte: elaborado pelo autor (2025).

Para o segundo teste do algoritmo, o dataset utilizado possuía 30 mil pontos de treinamento e 100 pontos de teste. A Figura 5 demonstra como o aumento do dataset fez com que a implementação da biblioteca CuML performasse melhor em comparação com as outras, mas ainda assim, 100 pontos de teste não são o suficiente para fazer com que essa implementação seja mais eficiente que a da biblioteca Scikit-learn, que é altamente otimizada e não sofre com *overhead* por conta da GPU. Também fica claro como o processamento paralelo em Bend, mesmo que apenas na CPU, reduz o tempo total de processamento do algoritmo, sendo computado 6 vezes mais rápido que a execução sequencial.

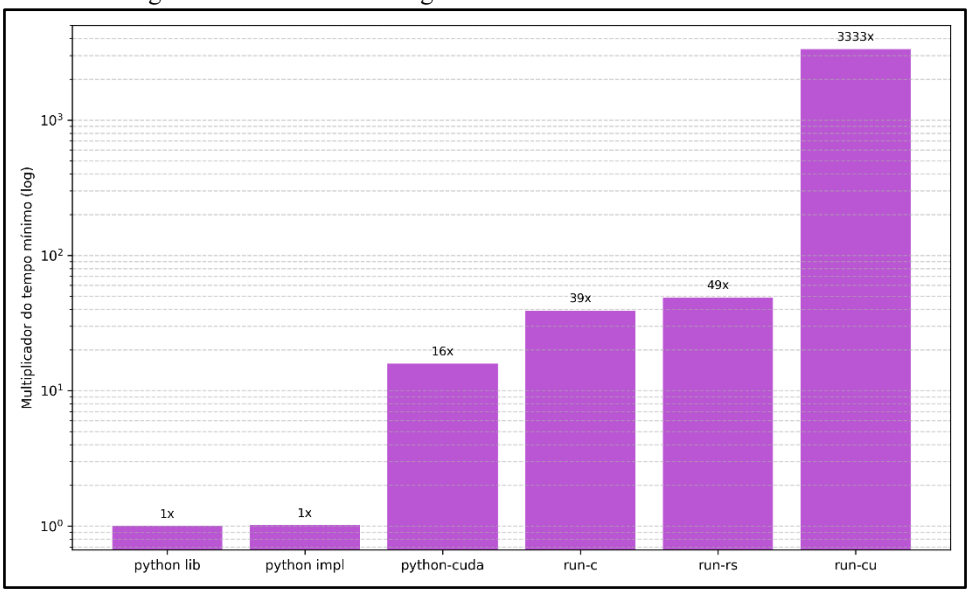
Figura 5 - Resultados do algoritmo KNN com o maior dataset



Fonte: elaborado pelo autor (2025).

O segundo algoritmo analisado foi o K-Means, utilizando um dataset pequeno de 50 pontos, divididos entre 5 centróides. Como visto na Figura 6, por conta do pequeno conjunto de dados utilizados, não houve uma diferença notável entre as execuções sequenciais e as paralelas, sendo observável apenas uma diminuição de velocidade em ambas as execuções que dependem da GPU: python-cuda e run-cu. Ainda assim, considerando a diferença entre o algoritmo compilado em CUDA através do Bend e as outras implementações, fica claro que a falta de otimização afeta desproporcionalmente a linguagem, diminuindo muito sua performance.

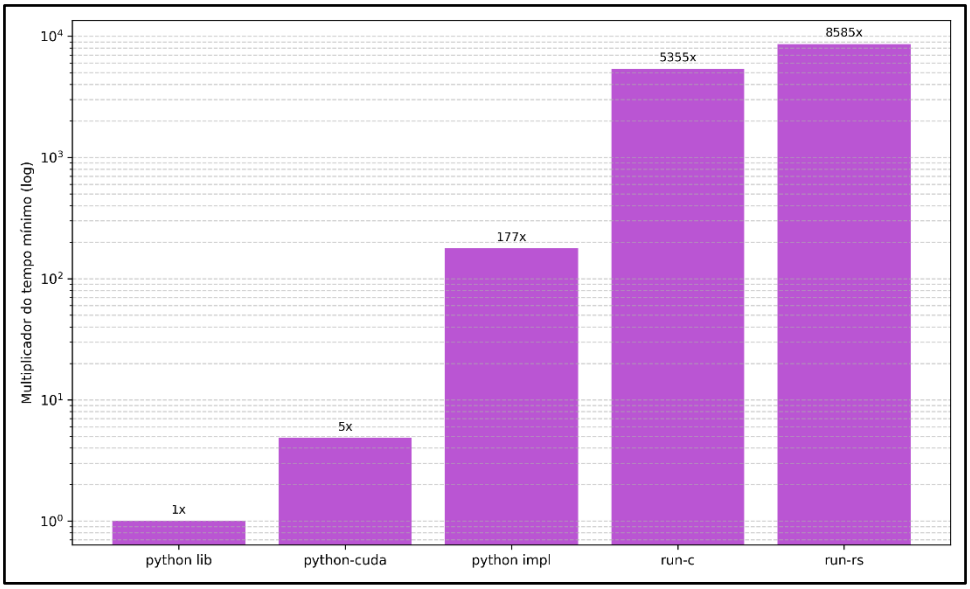
Figura 6 - Resultados do algoritmo K-Means com o menor dataset



Fonte: elaborado pelo autor (2025).

Utilizando um dataset maior, contendo 30 mil pontos, os tempos de execução foram registrados e apresentados na Figura 7. Neste contexto, a implementação baseada na GPU da biblioteca CuML se mostra mais eficiente que a implementação própria do autor, mas o dataset utilizado ainda não possui um conjunto de dados grande o suficiente para essa versão ser mais eficiente que a implementação da biblioteca Scikit-learn. Seguindo a tendência observada nos resultados anteriores, a execução do algoritmo em Bend, executada em ambos os compiladores, apresentou desempenho mais de 5 mil vezes inferior ao melhor resultado obtido, evidenciando novamente a falta de otimização na linguagem.

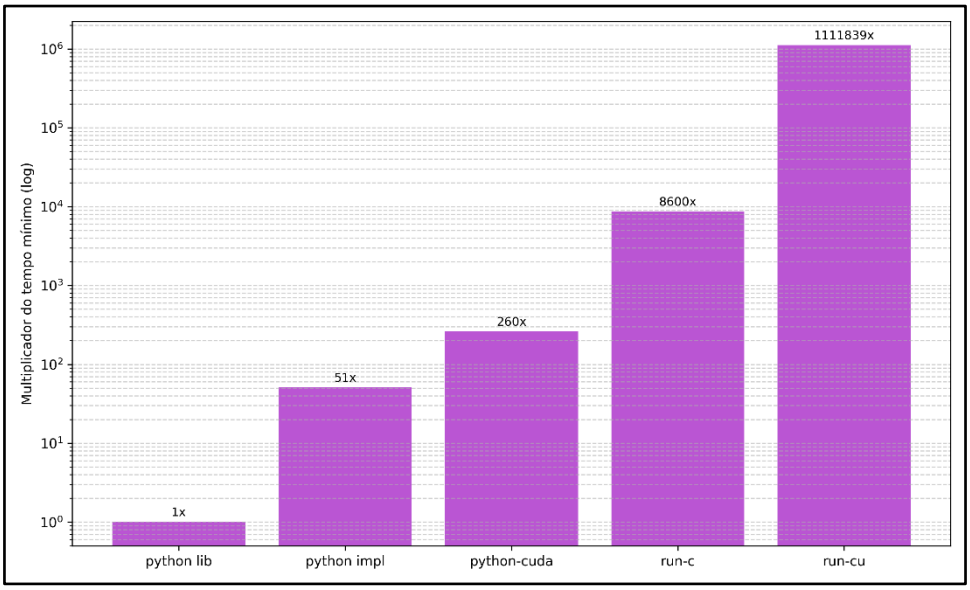
Figura 7 - Resultados do algoritmo K-Means com o maior dataset



Fonte: elaborado pelo autor (2025).

Como último algoritmo, as árvores de decisão foram aplicadas sobre um dataset com 100 pontos de treinamento e 100 pontos de teste. Para este algoritmo em específico, o compilador Rust apresentava erros durante a execução: *HVM output had no result (An error likely occurred)*. Por conta disso, esse compilador foi removido dos testes. Como visto na Figura 8, a implementação da biblioteca Scikit-learn continua sendo a mais eficiente, seguida pelas outras implementações em Python. Além disso, ambas as execuções em Bend obtiveram resultados não satisfatórios.

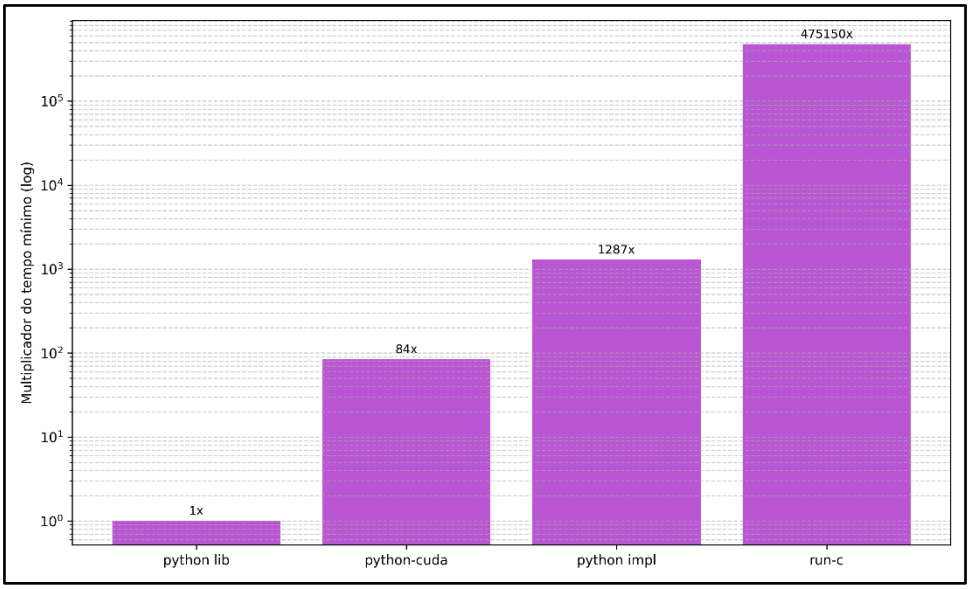
Figura 8 - Resultados do algoritmo árvores de decisão com o menor dataset



Fonte: elaborado pelo autor (2025).

Como segundo teste para o algoritmo de árvores de decisão, as implementações foram executadas sobre um conjunto de dados com 1300 pontos de treinamento e 100 pontos de teste. A Figura 9 apresenta os resultados obtidos em cada implementação executada sobre o dataset. Pode-se observar que as versões em Python continuam sendo mais eficientes que a implementação em Bend.

Figura 9 - Resultados do algoritmo árvores de decisão com o maior dataset



Fonte: elaborado pelo autor (2025).

5 CONCLUSÕES

O objetivo de implementar e avaliar a performance de diversos algoritmos de aprendizado de máquina na linguagem de programação Bend foi atendido. Apesar disso, por conta dos problemas, limitações técnicas e instabilidades encontrados durante o desenvolvimento e os testes, a linguagem demonstrou não ser tão acessível e simples quanto prometia.

Para que se obtenha qualquer vantagem por meio do processamento paralelo, um programa deve ser escrito em Bend com base em uma lógica paralela desde o início. O funcionamento e a estrutura de um programa precisam ser completamente repensados dentro dessa abordagem, uma vez que o fluxo de execução não pode seguir uma lógica sequencial tradicional, exigindo que as operações sejam atômicas e independentes entre si. Além disso, a ausência de certos recursos na linguagem, como vetores, dificulta o desenvolvimento, exigindo a reformulação de rotinas simples com o uso de estruturas mais complexas e menos eficientes.

Devido à ausência de estruturas de iteração e de outros elementos comuns em linguagens imperativas, o usuário também deve se adaptar a um paradigma funcional para utilizar a linguagem, se restringindo ao uso de recursões, variáveis imutáveis, funções puras e mônades. Para desenvolvedores não familiarizados com este paradigma, tarefas simples podem se tornar mais complexas e o código resultante pode não ter uma boa performance.

Os resultados obtidos em relação ao tempo de execução dos algoritmos também não foram satisfatórios quando comparados às implementações em Python. A baixa performance das execuções na CPU era esperada, por conta da falta de otimização da linguagem em comparação à linguagens mais consolidadas. Apesar disso, esperava-se que essa diferença seria compensada pela capacidade de processamento paralelo da GPU, o que não se confirmou nos testes realizados. A limitação do tamanho dos datasets durante os testes também não proporcionou uma comparação justa entre os algoritmos, visto que seria necessário um conjunto de dados significativamente maior para que o ganho de performance proporcionado pelo processamento paralelo superasse o *overhead* criado pelo uso da GPU.

5.1 PROBLEMAS ENCONTRADOS

Como mencionado anteriormente, o maior problema enfrentado durante o desenvolvimento foi a utilização do compilador CUDA para a execução dos algoritmos implementados. Os três compiladores disponibilizados: Rust, C e CUDA, deveriam ser equivalentes, permitindo que um programa fosse executado de forma consistente entre eles. Contudo, apesar de diversas tentativas, não foi alcançado um funcionamento satisfatório do compilador CUDA, principalmente devido aos erros de estouro de memória que aconteciam ao utilizar um conjunto com algumas centenas de dados. O compilador Rust também não possui todas as funcionalidades encontradas no compilador C, como interação com I/O (entrada/saída).

A interação com I/O na linguagem Bend também é de difícil implementação, e requer grandes alterações no programa para realizar operações simples, como imprimir na tela algum valor arbitrário. Além disso, diversas funcionalidades normalmente presentes em bibliotecas padrão precisam ser implementadas manualmente pelo usuário, como conversores de string para valores numéricos, funções de máximo e mínimo, manipulação de listas, entre outras.

Algumas funções implementadas também apresentavam comportamento incorretos, como no caso da divisão de uma string por um delimitador, que retornava as *substrings* em ordem invertida, exigindo correções adicionais.

Além dos pontos levantados, é importante ressaltar que a empresa criadora de Bend, a HOC, anunciou o lançamento da próxima versão da linguagem, Bend2, e consequentemente, o encerramento do projeto original. Levando isso em conta, em conjunto com os resultados e limitações encontrados, não é sugerido que Bend seja explorado em trabalhos futuros.

REFERÊNCIAS

DUDA, R. O.; HART, P. E.; STORK, D. G. **Pattern Classification**. 2. ed. Nashville, TN, USA: John Wiley & Sons, 2000.

GARCIA, V. *et al*. **Fast k nearest neighbor search using GPU**. 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops. Anais...IEEE, 2008.

HASTIE, T.; TIBSHIRANI, R.; FRIEDMAN, J. H. **The elements of statistical learning: Data mining, inference, and prediction**. 2. ed. Nova Iorque, NY, USA: Springer, 2009.

HONG-TAO, B. *et al*. **K-means on commodity GPUs with CUDA**. 2009 WRI World Congress on Computer Science and Information Engineering. Anais...IEEE, 2009.

KIRK, D. B.; HWU, W.-M. W. **Programming massively parallel processors: A hands-on approach**. 3. ed. Oxford, England: Morgan Kaufmann, 2016.

LAFONT, Y. Interaction combinators. **Information and computation**, v. 137, n. 1, p. 69–101, 1997.

MARSLAND, S. **Machine learning: An algorithmic perspective, second edition**. 2. ed. Filadélfia, PA, USA: Chapman & Hall/CRC, 2014.

MITCHELL, T. **Machine Learning**. Nova Iorque, NY, USA: McGraw-Hill Professional, 1997.

NASRIDINOV, A.; LEE, Y.; PARK, Y.-H. **Decision tree construction on GPU: ubiquitous parallel computing approach.** Computing, v. 96, n. 5, p. 403–413, 2014.

OWENS, J. D. *et al*. GPU Computing. **Proceedings of the IEEE. Institute of Electrical and Electronics Engineers**, v. 96, n. 5, p. 879–899, 2008.

SISODIA, D.; SISODIA, D. S. Prediction of Diabetes using Classification Algorithms. **Procedia computer science**, v. 132, p. 1578–1585, 2018.

TAELIN, V. **Higher order company**. Disponível em: <https://higherorderco.com>. Acesso em: 22 set. 2024.

WOLFF, A. *et al*. **Improving retention: Predicting at-risk students by analysing clicking behaviour in a virtual learning environment**. Proceedings of the Third International Conference on Learning Analytics and Knowledge. **Anais**...New York, NY, USA: ACM, 2013.

YAMATO, Yoji. Study and evaluation of automatic GPU offloading method from various language applications. **International Journal Of Parallel, Emergent And Distributed Systems**, [S.l.], v. 37, n. 1, p. 22-39, 6 set. 2021. Informa UK Limited.