Universidade Federal do ABC

Engenharia de Informação

Aprendizado de Máquina em Sistemas Embarcados para Detecção de Falhas em Máquinas Rotativas

Aluno: Ítalo Milhomem de Abreu Lanza

Orientador: Ricardo Suyama

Santo André
2025

Universidade Federal do ABC

Engenharia de Informação

Aprendizado de Máquina em Sistemas Embarcados para Detecção de Falhas em Máquinas Rotativas

Aluno: Ítalo Milhomem de Abreu Lanza

Orientador: Ricardo Suyama

Trabalho de Graduação apresentado ao curso de Engenharia de Informação da Universidade Federal do ABC, como parte dos requisitos necessários para a obtenção do grau de Bacharel em Engenharia de Informação.

Santo André
2025

Resumo

As máquinas elétricas rotativas são responsáveis por grande parte do consumo energético industrial e, devido ao seu papel crítico, falhas não detectadas podem gerar custos elevados de manutenção e paralisações. Nesse contexto, estratégias de manutenção preditiva vêm ganhando destaque, especialmente quando apoiadas em técnicas de monitoramento de sinais. Embora métodos tradicionais, como análise de vibração e termografia, sejam consolidados, apresentam limitações de custo, complexidade ou aplicação em larga escala. A análise acústica, aliada a técnicas de *machine learning*, surge como alternativa promissora, permitindo a detecção não intrusiva de falhas.

Neste trabalho, investigou-se a adaptação da abordagem proposta por Shubita et al., que utiliza sinais acústicos processados por modelos de aprendizado de máquina para a detecção de falhas em máquinas rotativas. A metodologia original, baseada em ferramentas proprietárias, foi substituída por um fluxo de desenvolvimento inteiramente construído com bibliotecas de código aberto em Python e implementado em hardware acessível. Os resultados demonstraram que a solução reproduziu os níveis de acurácia reportados no estudo de referência, inclusive em sistemas embarcados, validando a eficácia do *pipeline* proposto. Além de confirmar a robustez da biblioteca Emlearn para aplicações em *Edge ML*, o estudo também evidenciou limitações relacionadas ao processamento em tempo real, apontando oportunidades para futuras otimizações e integrações com aquisição direta de sinais acústicos.

Palavras-chave: Manutenção preditiva, Máquinas rotativas, Análise acústica, Aprendizado de máquina, Edge ML, Sistemas embarcados

Abstract

Rotating electrical machines account for a significant share of industrial energy consumption, and due to their critical role, undetected failures can result in high maintenance costs and production downtime. In this context, predictive maintenance strategies have gained increasing relevance, especially when supported by signal monitoring techniques. Although traditional methods, such as vibration analysis and thermography, are well-established, they present limitations in terms of cost, complexity, or large-scale applicability. Acoustic analysis, combined with *machine learning* techniques, emerges as a promising alternative, enabling non-intrusive fault detection.

In this work, we investigated the adaptation of the approach proposed by Shubita et al., which employs acoustic signals processed by machine learning models for fault detection in rotating machines. The original methodology, based on proprietary tools, was replaced by a development flow entirely built with open-source Python libraries and implemented on low-cost hardware. The results showed that the solution reproduced the accuracy levels reported in the reference study, including in embedded systems, validating the effectiveness of the proposed *pipeline*. In addition to confirming the robustness of the Emlearn library for *Edge ML* applications, the study also highlighted limitations related to real-time processing, pointing to opportunities for future optimizations and integration with direct acoustic signal acquisition.

Keywords: Predictive maintenance, Rotating machines, Acoustic analysis, Machine learning, Edge ML, Embedded systems

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Símbolos

Sigla	Descrição
ANOVA	Analysis of Variance (Análise de Variância)
CART	Classification and Regression Trees (Árvores de Classificação e Regressão)
CF	Crest Factor (Fator de Crista)
DSP	Digital Signal Processing (Processamento Digital de Sinais)
ECDF	Empirical Cumulative Distribution Function (Função de Distribuição Cumulativa Empírica)
FFT	Fast Fourier Transform (Transformada Rápida de Fourier)
FPU	Floating-Point Unit (Unidade de Ponto Flutuante)
IDE	Integrated Development Environment (Ambiente de Desenvolvimento Integrado)
IF	Impulse Factor (Fator de Impulso)
IIR	Infinite Impulse Response (Resposta ao Impulso Infinita)
KNN	K-Nearest Neighbors (K-Vizinhos Mais Próximos)
MCU	Microcontroller Unit (Unidade Microcontroladora)
ML	Machine Learning (Aprendizado de Máquina)
MF	Margin Factor (Fator de Margem)
RMS	Root Mean Square (Valor Quadrático Médio)
SF	Shape Factor (Fator de Forma)
SOS	Second-Order Section (Seção de Segunda Ordem)
SVN	Support Vector Machines (Máquinas de Vetores de Suporte)
SPI	Serial Peripheral Interface (Interface Periférica Serial)
UFABC	Universidade Federal do ABC

Sumário

1 Introdução

As máquinas elétricas rotativas são componentes críticos em processos industriais, sendo responsáveis por mais de 60% da energia consumida em setores como o automotivo e o de geração de energia [1]. A indisponibilidade desses equipamentos, frequentemente causada por falhas mecânicas não detectadas, pode resultar em perdas superiores a US$ 260 mil por hora em linhas de produção de alta complexidade [2].

Dentre as estratégias de manutenção, destacam-se três abordagens principais: (i) a **corretiva**, que atua após a ocorrência da falha, gerando custos elevados com reparos e paralisações; (ii) a **preventiva**, baseada em intervalos fixos, que muitas vezes substitui componentes ainda funcionais; e (iii) a **preditiva**, que utiliza dados em tempo real para antecipar falhas, reduzindo a necessidade de intervenções corretivas inesperadas e otimizando custos e disponibilidade [3]. Para máquinas rotativas, cujo desgaste progressivo exige monitoramento contínuo, a abordagem preditiva é essencial, embora deva ser entendida como complementar — e não substitutiva — às demais práticas de manutenção.

Entre os métodos de monitoramento preditivo, técnicas baseadas em **análise de vibração** são as mais consolidadas, utilizando sensores piezoelétricos para detectar anomalias em engrenagens e rolamentos [4]. Alternativas como **termografia infravermelha** identificam superaquecimento em componentes, enquanto a **análise de óleo** verifica a presença de partículas metálicas indicativas de desgaste [5]. Entretanto, essas abordagens exigem equipamentos especializados, instalação invasiva ou coleta manual de amostras, limitando sua aplicação em larga escala.

Como alternativa, a **análise de sinais acústicos** emerge como uma solução não intrusiva, capaz de detectar falhas como trincas em rolamentos ou desalinhamentos em engrenagens sem contato físico com a máquina [6]. Um único microfone MEMS (Sistema Microeletromecânico), de baixo custo (cerca de US$ 1,50 por unidade [7]), pode monitorar múltiplos componentes simultaneamente, mesmo em ambientes de difícil acesso. No entanto, a interpretação manual desses sinais — tradicionalmente realizada por técnicos experientes — é subjetiva e inviável para monitoramento contínuo. Por exemplo, um rolamento com trinca interna emite pulsos de alta frequência (8–16 kHz) mascarados por ruído ambiente, exigindo técnicas avançadas de filtragem (ex: *wavelet*) e processamento espectral [8].

É nesse cenário que técnicas de *machine learning* (ML) ganham relevância. Algoritmos supervisionados, como árvores de decisão e redes neurais, podem aprender padrões complexos em dados acústicos brutos, correlacionando características no domínio do tempo (ex: RMS, curtose) e frequência (ex: picos espectrais) com estados de falha específicos [9]. A integração desses modelos em dispositivos de borda (*edge ML*) permite análise em tempo real, sem dependência de conexão com a nuvem, ideal para ambientes industriais com restrições de infraestrutura.

1.1 Objetivo

Este trabalho propõe uma adaptação do método proposto de Shubita et al. [9], substituindo ferramentas proprietárias (MATLAB) por um *pipeline* baseado em Python e bibliotecas de código aberto (e.g. Scikit-learn [10], emlearn [11]) e hardware acessível (ST STM32 Nucleo-F446RE ao invés da placa STM32F407VG). Além disso, analisa-se o impacto de decisões técnicas tomadas devido à diferença entre as ferramentas e tecnologias utilizadas.

2 Fundamentação Teórica

2.1 Detecção de Falhas em Máquinas Rotativas

O monitoramento de condições e o diagnóstico de falhas são componentes essenciais para a manutenção preditiva em ambientes industriais, visando garantir a confiabilidade e a segurança de equipamentos críticos [1, 12]. Máquinas rotativas, como motores e redutores, são um dos componentes mais importantes de inúmeros processos, e suas falhas inesperadas podem levar a paradas de produção e custos elevados [3].

As falhas nesses sistemas frequentemente se manifestam em componentes específicos devido ao desgaste e a estresses mecânicos. Entre as falhas mais comuns, destacam-se:

- **Falhas em rolamentos:** Defeitos como trincas ou pites nas pistas interna, externa ou nos elementos rolantes são uma das principais causas de paradas em máquinas rotativas e geram sinais vibracionais e acústicos característicos [2, 9].
- **Falhas em engrenagens:** Desgaste, dentes quebrados ou desalinhamento em caixas de engrenagens também são fontes comuns de problemas, alterando o padrão sonoro da operação normal da máquina [4].
- **Outras falhas mecânicas:** Problemas como desbalanceamento de eixos ou danos em componentes auxiliares, como ventoinhas de refrigeração, também introduzem assinaturas acústicas que podem ser detectadas [9, 12].

O presente trabalho foca na detecção dessas classes de falhas por meio da análise dos sinais sonoros emitidos pela máquina durante sua operação, uma abordagem não invasiva e de baixo custo. De fato, a análise de sinais acústicos é uma abordagem consolidada e eficaz para o monitoramento de condições e detecção de falhas em máquinas rotativas [6, 13]. A metodologia se baseia no princípio de que alterações no estado operacional de uma máquina, como o surgimento de um defeito em um rolamento, modificam o som emitido por ela [14]. A captura e o processamento digital desses sinais sonoros permitem identificar essas alterações de forma não invasiva, tornando-se uma alternativa poderosa a outras técnicas de manutenção preditiva [9].

O processo de análise pode ser dividido em quatro etapas principais:

1. **Aquisição de Sinais:** Conversão do sinal sonoro analógico em um formato digital.
2. **Pré-processamento:** Filtragem e preparação do sinal digital para remover ruídos e artefatos.
3. **Extração de *Features*:** Cálculo de métricas nos domínios do tempo e da frequência para quantificar as características do sinal.
4. **Classificação:** Identificação do estado da máquina a partir das *features* observadas.

Essas etapas são fundamentais para transformar o áudio bruto em um conjunto de dados estruturado que pode ser utilizado por algoritmos de aprendizado de máquina para classificação e diagnóstico [8].

2.1.1 Aquisição de Sinais

A aquisição é o primeiro passo do processamento digital e consiste na conversão de um sinal analógico, contínuo no tempo e na amplitude, em um sinal digital, discreto em ambas as dimensões [15].

O sinal de interesse deve ser captado por meio de um transdutor adequado, transformando uma grandeza física variante no tempo (como pressão acústica) em um sinal elétrico, que posteriormente é convertido para uma sequência de amostras. Esse último processo é realizado por um Conversor Analógico-Digital (ADC), que executa a operação de **amostragem**. A amostragem mede a amplitude do sinal analógico em intervalos de tempo regulares, definidos pela **frequência de amostragem** (f_s). A Figura 1 ilustra como funciona o processo de amostragem de um sinal analógico contínuo.

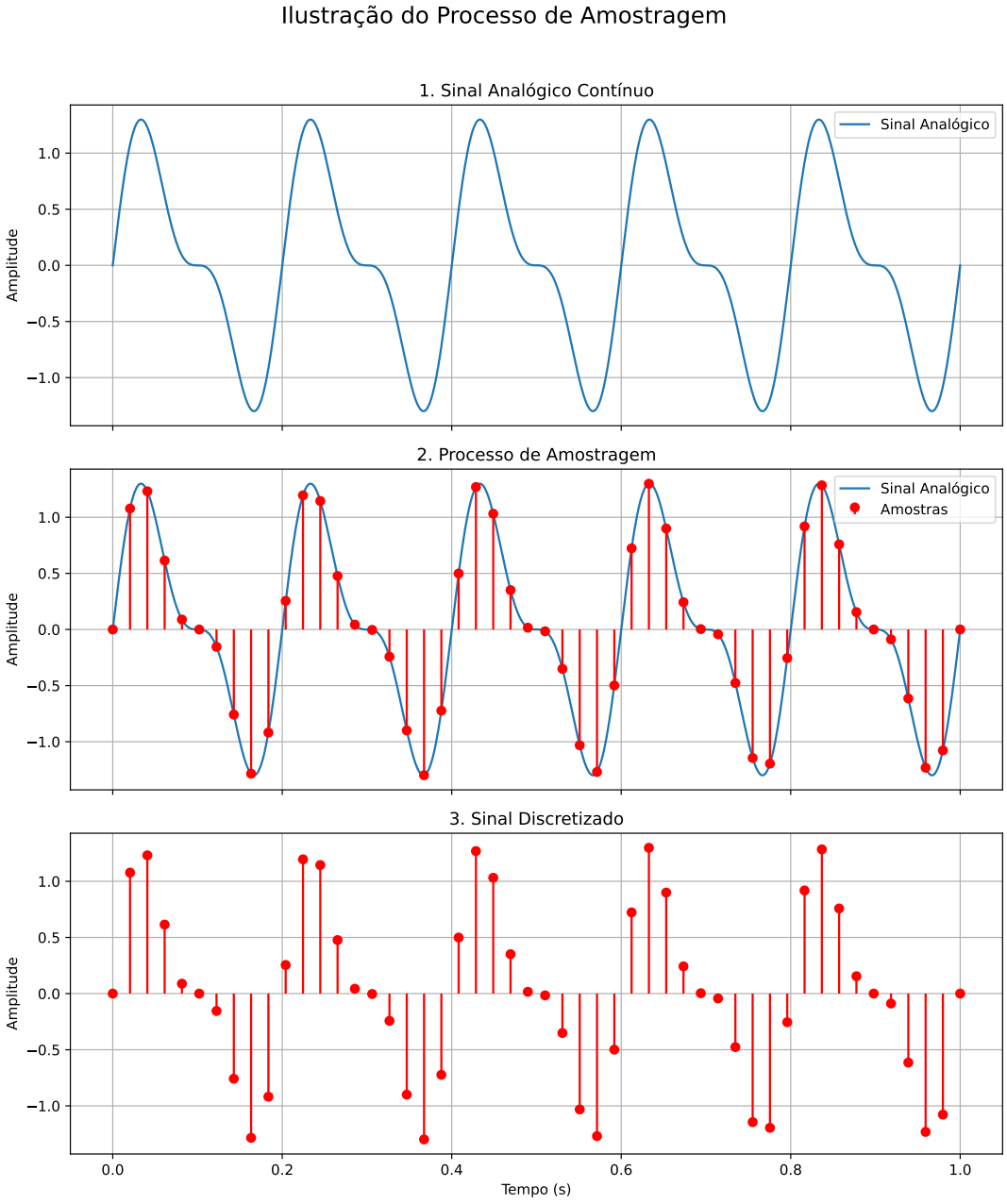

Figura 1: Ilustração do processo de amostragem, em que o sinal analógico contínuo (onda suave) é medido em intervalos de tempo regulares, gerando um conjunto de amostras discretas (pontos) que representam o sinal original em formato digital.

Para garantir que o sinal digital seja uma representação fiel do sinal analógico original, a frequência de amostragem deve obedecer ao **Teorema de Nyquist-Shannon**. O teorema declara que a frequência de amostragem deve ser estritamente maior que o dobro da frequência máxima contida no sinal ($f_s > 2 \cdot f_{max}$) [15]. O não cumprimento deste critério resulta em um erro de *aliasing*, isto é, altas frequências

do sinal são erroneamente interpretadas como baixas frequências, corrompendo a informação espectral conforme ilustrado na Figura 2.

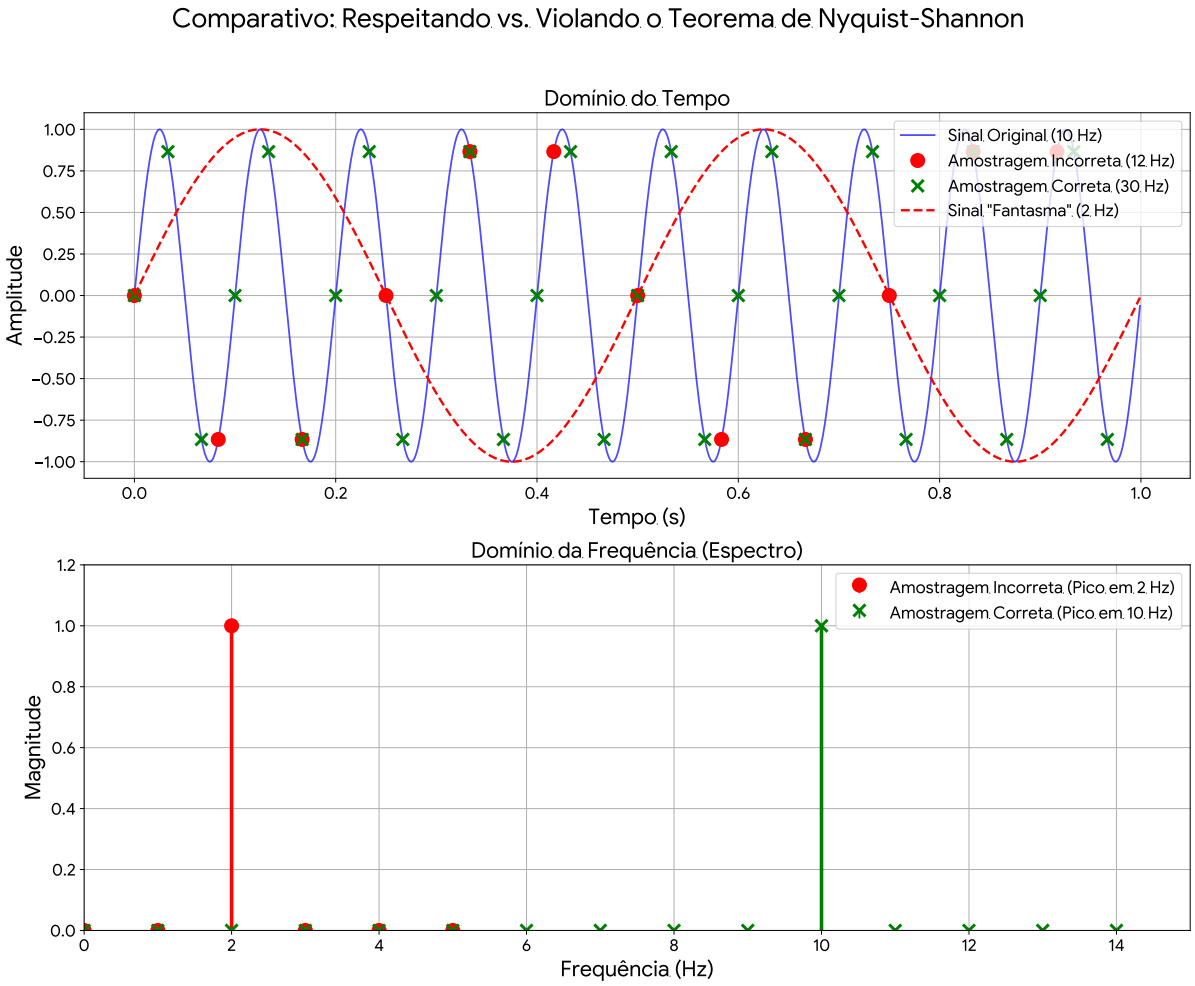

Figura 2: Ilustração do processo de *aliasing*, em que um sinal amostrado com uma frequência menor que duas vezes a frequência mais alta que compõe o sinal base gera no espectro de frequência um componente de baixa frequência.

2.1.2 Pré-processamento

Após a digitalização, o sinal geralmente contém ruído e outros componentes que não são de interesse para a análise da falha. O pré-processamento visa limpar o sinal para facilitar a subsequente extração de características [16]. As técnicas mais comuns nesta etapa são a filtragem e o janelamento.

A **filtragem digital** é aplicada para restringir o sinal à banda de frequência onde as informações relevantes da falha são esperadas. É comum que o sinal adquirido possua um desvio no nível de tensão médio, conhecido como *offset DC*, que pode ser removido com um filtro passa-alta. Da mesma forma, ruídos de alta frequência podem ser atenuados com um filtro passa-baixa. A combinação de ambos resulta em um filtro passa-banda, que isola a faixa de frequência de interesse [15].

O **janelamento** é uma operação essencial antes de se aplicar Transformada Rápida de Fourier (**FFT**, do inglês *Fast Fourier Transform*). Como a FFT processa blocos de dados de tamanho finito, ela assume que esse bloco se repete periodicamente. Essa suposição é quase sempre falsa, levando a descontinuidades nas bordas do bloco que causam o "vazamento espectral" (*spectral leakage*), um efeito

que espalha a energia de uma frequência para frequências vizinhas. Para minimizar esse problema, o bloco de dados é multiplicado por uma **função de janela** (como a de *Hann*), que força as amplitudes nas bordas a se aproximarem de zero suavemente, reduzindo as descontinuidades e melhorando a resolução da análise de frequência [15, 16]. Os elementos da janela de *Hann* podem ser calculados seguindo a equação 1.

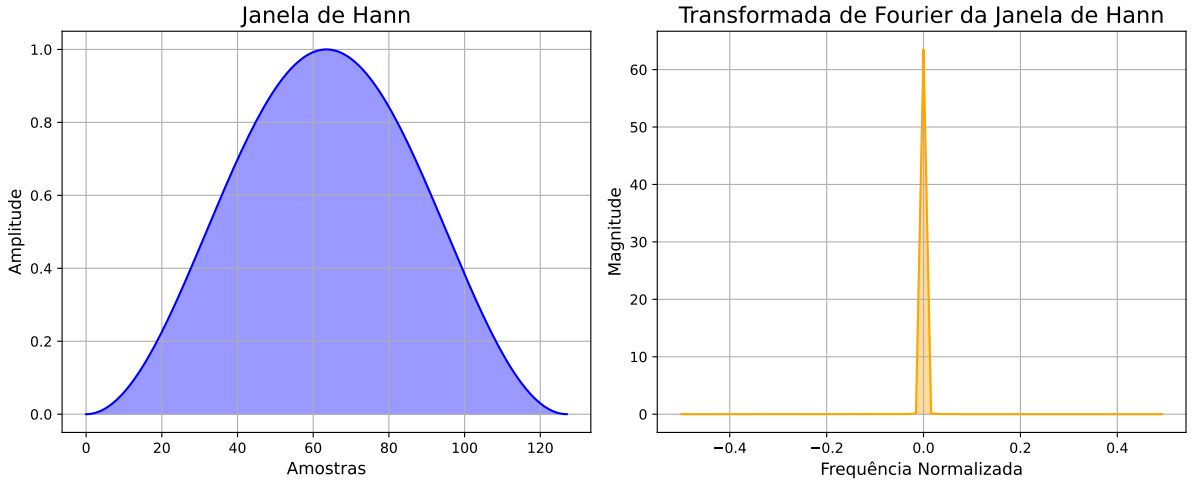

Figura 3: Ilustração da Janela de Hann. Sua função à direita e espectro de frequência à esquerda.

$$w(n) = 0.5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right) \tag{1}$$

onde:

- $w(n)$ é o valor da janela na amostra `n`;
- n é o indice da amostra atual (que vai de `0` até `N-1`);
- N é o número total de amostras (tamanho da janela).

2.1.3 *Features* no Domínio do Tempo

As *features* (características) no domínio do tempo são métricas estatísticas calculadas a partir das amostras de amplitude do sinal pré-processado [17]. Elas têm como objetivo resumir a forma de onda em um conjunto reduzido de valores numéricos, facilitando o trabalho do classificador de aprendizado de máquina. Essas métricas podem descrever a energia do sinal, a distribuição de suas amplitudes e a presença de impulsos [18]. As seções seguintes detalham as *features* no domínio do tempo que foram extraídas neste trabalho.

2.1.3.1 *(Arithmetic) Mean*

Em estatística, a Média Aritmética, ou *Arithmetic Mean*, é definida como a soma do valor de todos elementos de um conjunto de dados dividido pela quantidade total de elementos[19]. Pode ser calculado da seguinte forma:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{2}$$

onde:

- μ é o valor da média;

- N é total de elementos do conjunto;
- $x_{(i)}$ é i-ésimo valor do conjunto.

2.1.3.2 *Median*

Dado um conjunto de elementos ordenados em ordem crescente, a Mediana, ou *Median*, é definida como o elemento central desse conjunto [19]. Quando o numero de elementos do conjunto for par, utiliza-se a média aritmética dos dois elementos centrais do conjunto como mediana. A Mediana é definida como:

$$\text{Mediana} = \begin{cases} x_{((n+1)/2)} & \text{se } n \text{ é ímpar} \\ \dfrac{x_{(n/2)} + x_{(n/2+1)}}{2} & \text{se } n \text{ é par} \end{cases} \tag{3}$$

onde:

- n é o número total de elementos do conjunto;
- $x_{(k)}$ é o valor do k-ésimo elemento do conjunto ordenado.

2.1.3.3 *Root mean square* (RMS)

O valor RMS (*Root mean square*), raiz do valor quadrático médio (ou valor eficaz), é uma medida estatística que mede a raiz quadrada da media aritmética dos quadrados dos valores de um conjunto de dados[20]. Dessa forma, o valor RMS é definido como:

$$\text{RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2} \tag{4}$$

onde:

- x_i é o i-ésimo elemento do conjunto;
- N é o número total de elementos do conjunto.

2.1.3.4 *Variance*

A *Variance*, ou variância, mede a dispersão dos valores do sinal em torno da sua média. Pode ser calculada a partir da formula a seguir:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2 \tag{5}$$

onde:

- σ^2 é o valor da variância;
- N é o número total de elementos do conjunto;
- x_i é o i-ésimo elemento do conjunto;
- μ é o valor da média aritmética.

2.1.3.5 *Shape factor* (SF)

O *Shape factor*, ou fator de forma, é um valor adimensional que descreve a forma da onda de um sinal. É definido como a razão entre o valor RMS e a média dos valores absolutos de um sinal [9, 18]. É definido como:

$$\mathrm{SF} = \frac{X_{\mathrm{RMS}}}{\mu_{\mathrm{abs}}}, \quad \mu_{\mathrm{abs}} = \frac{1}{N}\sum_{i=1}^{N}|x_i| \tag{6}$$

onde:

- X_{RMS} é valor RMS do sinal;
- μ_{abs} é valor médio absoluto do sinal;
- x_i -ésima amostra do sinal.

2.1.3.6 *Impulse factor* (IF)

O *Impulse factor*, ou fator de impulso, é uma medida que relaciona o valor de pico do sinal com a média de seus valores absolutos [17]. É definido como:

$$\mathrm{IF} = \frac{X_{\mathrm{peak}}}{\mu_{\mathrm{abs}}} \tag{7}$$

onde:

- X_{peak} é o valor de pico do sinal;
- μ_{abs} é o valor médio absoluto do sinal.

2.1.3.7 *Crest factor* (CF)

O *Crest factor*, ou fator de crista, é a razão entre o valor de pico do sinal e seu valor RMS [16]. Ele quantifica a intensidade dos picos em relação à energia total do sinal. É definido como:

$$\mathrm{CF} = \frac{X_{\mathrm{peak}}}{X_{RMS}} \tag{8}$$

onde:

- X_{peak} é o valor de pico do sinal;
- X_{RMS} é o valor RMS do sinal.

2.1.3.8 *Margin factor* (MF)

O *Margin factor*, também conhecido como *Clearance Factor*, o fator de margem relaciona o valor de pico com a média dos quadrados das raízes dos valores absolutos [17]. É definido como:

$$\mathrm{MF} = \frac{X_{\mathrm{peak}}}{\left(\frac{1}{N}\sum_{i=1}^{N}\sqrt{|x_i|}\right)^2} \tag{9}$$

onde:

- X_{peak} é o valor de pico do sinal;
- N é o número total de amostras do sinal;
- x_i é a i-ésima amostra do sinal.

2.1.3.9 *Kurtosis*

A *Kurtosis*, ou curtose, mede o quão "achatada" ou "pontuda" é a distribuição de probabilidade das amplitudes do sinal [17, 18]. O valor da curtose é grande para sinais "mpulsivos"[16]. É definida como:

$$\mathrm{K} = \frac{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^4}{\sigma^4} \tag{10}$$

onde:

- N é o número total de amostras do sinal;
- x_i é a i-ésima amostra do sinal;
- μ é o valor médio do sinal;
- σ é o valor do desvio padrão do sinal.

2.1.3.10 *Skewness*

O *Skewness*, ou assimetria, mede a simetria da distribuição de probabilidade das amplitudes do sinal em torno da sua média [17, 18]. O valor de assimetria é zero para sinais simétricos e grande para sinais assimétricos [16]. É definida como:

$$\mathrm{S} = \frac{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^3}{\sigma^3} \tag{11}$$

onde:

- N é o número total de amostras do sinal;
- x_i é a i-ésima amostra do sinal;
- μ é o valor médio do sinal;
- σ é o valor do desvio padrão do sinal.

2.1.4 *Features* no Domínio da Frequência

A extração de *features* no domínio da frequência é realizada a partir do espectro do sinal, que é obtido aplicando-se a Transformada Rápida de Fourier (**FFT**, do inglês *Fast Fourier Transform*) a janela de dados pré-processados. A FFT é um algoritmo computacionalmente eficiente para o cálculo da Transformada Discreta de Fourier (**DFT**, do inglês *Discrete Fourier Transform*) e de sua inversa [15, 21].

De forma simplificada, FFT converte um sinal do domínio do tempo — como a vibração de uma estrutura mecânica ao longo dos segundos — em seu espectro de frequências, que revela a intensidade (amplitude) de cada componente de frequência presente no sinal. O processo é ilustrado visualmente na Figura 4.

A análise deste espectro permite a identificação de características distintas do sinal. Dentre as *features* mais diretas e informativas estão os picos de maior energia. Neste trabalho, foram extraídas 6 *features* relacionadas ao domínio da frequência, os três picos de maior proeminência (amplitude e frequência correspondente), definidos como:

- **Peak1, Peak2, Peak3:** Correspondem aos três maiores valores de amplitude (magnitude) encontrados no espectro de frequência. Esses picos representam as frequências dominantes que mais contribuem para a energia total do sinal.
- **PeakLoc1, PeakLoc2, PeakLoc3:** Indicam as frequências exatas (em Hz) onde ocorrem os picos de amplitude *Peak1*, *Peak2* e *Peak3*, respectivamente.

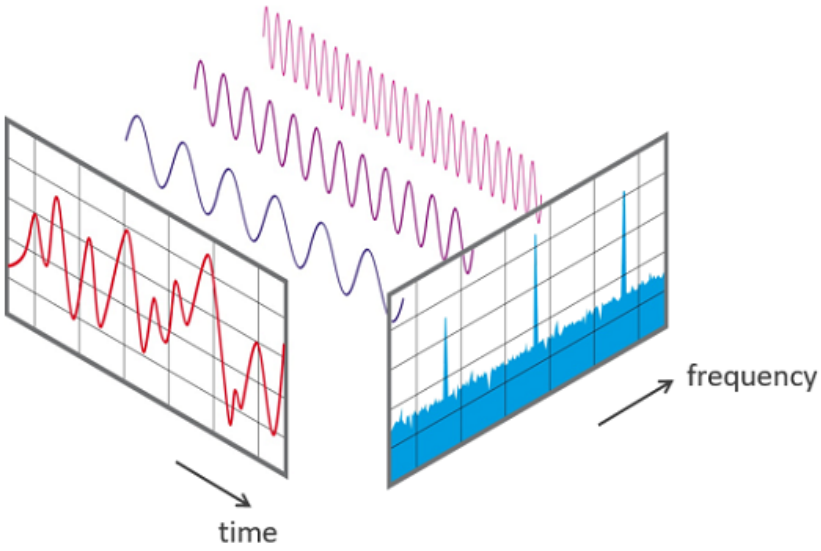

Figura 4: Ilustração do conceito da Transformada de Fourier, que decompõe um sinal no domínio do tempo (esquerda) em suas componentes de frequência constituintes (direita) - Fonte: [22].

Na implementação computacional, a FFT opera sobre um ***buffer*** (ou vetor) de N amostras do sinal de entrada (no caso desse trabalho, o sinal de entrada é o vetor resultante após a fase de pré-processamento). O resultado da FFT é um vetor de N números complexos, onde cada posição representa um *"bin"* de frequência. Para determinar a frequência exata de um pico ($PeakLocX$), é necessário converter o **índice (k)** desse pico no vetor de saída para um valor físico em Hertz (Hz). A conversão depende de dois parâmetros da aquisição: a **frequência de amostragem (f_s)** e o **tamanho do buffer (N)**, que é o número de pontos da FFT. A fórmula que mapeia o índice k para a frequência f_k é [15]:

$$f_k = k \cdot \frac{f_s}{N} \tag{12}$$

2.1.5 Classificação dos sinais

Após a extração e seleção de *features*, o próximo passo na *pipeline* de detecção de falhas é utilizar um modelo de Aprendizagem de Máquina (*Machine Learning*) para classificar o estado da máquina. A classificação é uma tarefa de aprendizagem supervisionada na qual o objetivo é treinar um algoritmo para atribuir um rótulo de classe (neste caso, "saudável", "falha de rolamento", etc.) a um novo conjunto de dados de entrada (as *features* extraídas) [23, 24].

De maneira simplificada, todo modelo/algoritmo de aprendizado de máquina para classificação tem como princípio fundamental encontrar uma função que mapeie/correlacione os dados de entrada (as *features*) com uma saída esperada (o rótulo/classe correto(a)) [25]. A partir dessa simplificação, pode-se representar os modelos de maneira genérica a partir da seguinte equação:

$$Y = f(\mathbf{X}) \tag{13}$$

onde:

- Y é o resultado, o rótulo/classe;
- X é o vetor de *features* consideradas pelo modelo.

Com base nessa equação, os modelos de aprendizagem de máquina podem ser categorizados de acordo com a suposição e feita com relação à função f que mapeia as *features* (entrada) e à classe (saída) [26, 27]. Desse modo, e possível categorizar os modelos em dois tipos: **paramétricos** e **não-paramétricos**.

- **Modelos Paramétricos**: como o Naïve Bayes, fazem suposições fortes sobre a forma da função que mapeia as *features* para o resultado. Eles aprendem um número fixo de parâmetros, independentemente do tamanho do conjunto de dados. São modelos geralmente mais rápidos e necessitam de menos dados para treinar, mas seu desempenho pode ser limitado se as suposições sobre os dados estiverem incorretas [24].
- **Modelos Não Paramétricos**: como as Árvores de Decisão e o K-Vizinhos Próximos (**KNN**, do inglês *K-Nearest Neighbors*), fazem poucas ou nenhuma suposição sobre a forma da função que mapeia as *features* para o resultado. Esses modelos se ajustam aos dados, de modo que conseguem mapear relações mais complexas. Por conta disso, são mais flexíveis e podem alcançar maior desempenho, mas geralmente exigem mais dados para treinar e podem ser mais lentos para fazer previsões [23].

O artigo base desse trabalho [9] avalia diversos classificadores, os quais seus princípios básicos funcionamentos serão explicados nas próximas seções. Para ilustrar visualmente o funcionamento e as diferenças entre os classificadores, as figuras subsequentes foram geradas utilizando o clássico *dataset Iris* [10]. Por fim, para simplificar a visualização, foram utilizadas apenas duas *features* do conjunto de dados (comprimento e largura da sépala) e duas das três classes (Setosa e Versicolor). Cada gráfico exibe a **fronteira de decisão** aprendida pelo modelo, que representa a superfície que separa as diferentes classes no espaço de *features*.

2.1.5.1 *Decision Tree*

A Árvore de Decisão (*Decision Tree*) é um modelo de aprendizado de máquina não paramétrico bastante flexível, sendo capaz de aprender relações complexas a partir dos dados [23, 28]. O seu funcionamento baseia-se na criação de uma hierarquia de regras de decisão simples (sim/não) sobre os valores das *features*. Como cada regra divide os dados de forma perpendicular a um dos eixos, o resultado final é uma fronteira de decisão composta por uma série de retângulos. A Figura 5 ilustra exatamente este comportamento, onde as regiões retangulares separam as duas classes do exemplo utilizando apenas duas *features*.

O processo de construção da Árvore de Decisão é comumente realizado pelo algoritmo *Classification and Regression Trees* (**CART**) [29]. Este algoritmo opera através de um processo de particionamento recursivo binário, que se inicia com todo o conjunto de dados no nó raiz. A cada passo, o algoritmo procura a melhor divisão possível, ou seja, a *feature* e o limiar de valor que separe os dados em dois subconjuntos (os nós filhos) da forma mais "pura"possível.

Para encontrar esta divisão ótima, o CART otimiza uma função de custo, como o Índice de Gini (o mais comum) ou o Ganho de Informação (Entropia), com o objetivo de criar nós filhos que sejam os mais homogêneos possível em relação às classes [23, 28]. Este processo de divisão é repetido para cada novo nó, criando a estrutura hierárquica da árvore.

No entanto, para evitar que a árvore cresça indefinidamente e se ajuste em excesso aos dados de treino (*overfitting*), é necessária a definição de um critério de parada [30]. Os critérios mais comuns, que funcionam como hiperparâmetros de regularização, incluem a definição de uma profundidade máxima para a árvore, a interrupção quando um nó atinge a pureza total (Índice de Gini igual a zero), ou a exigência de um número mínimo de amostras para que um nó possa ser dividido. A partir da Figura 6, é

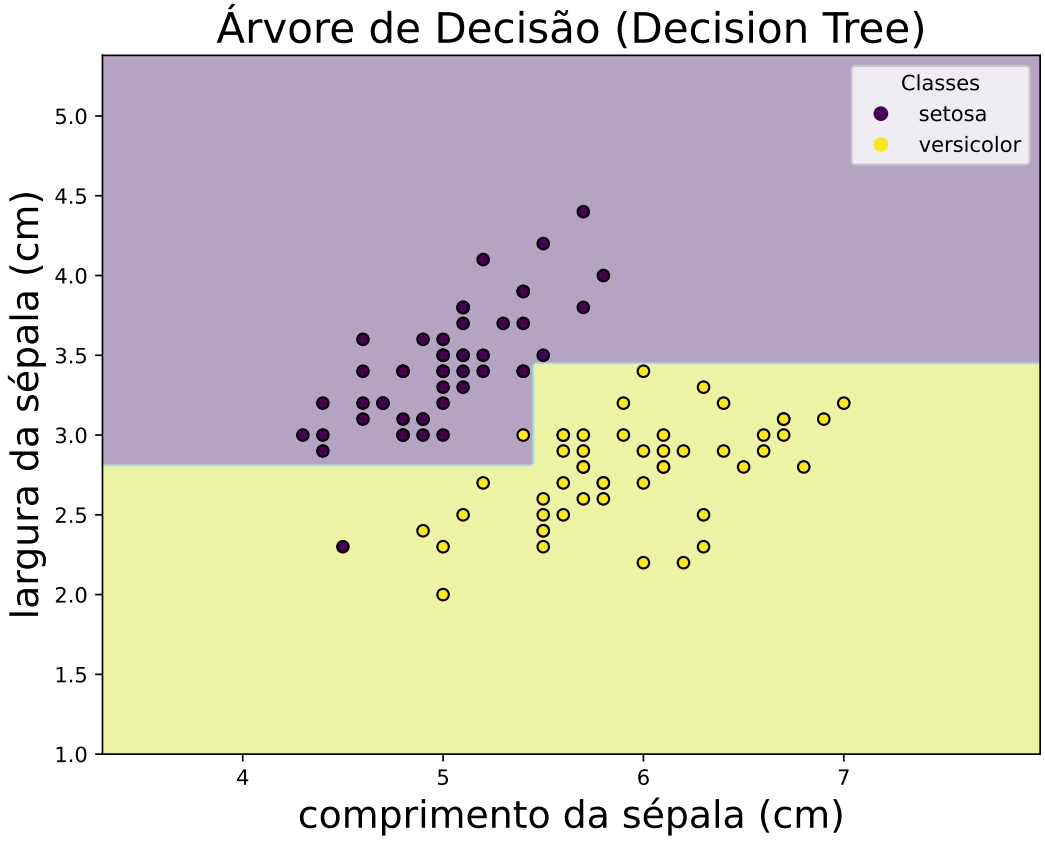

Figura 5: Fronteira de decisão de uma Árvore de Decisão treinada utilizando duas *features* (comprimento e largura da sépala) e duas classes (Setosa e Versicolor) do *dataset* Iris.

possível verificar a estrutura lógica resultante do treinamento utilizando-se o Índice de Gini como função de custo e configurando a profundidade máxima da arvore como 3.

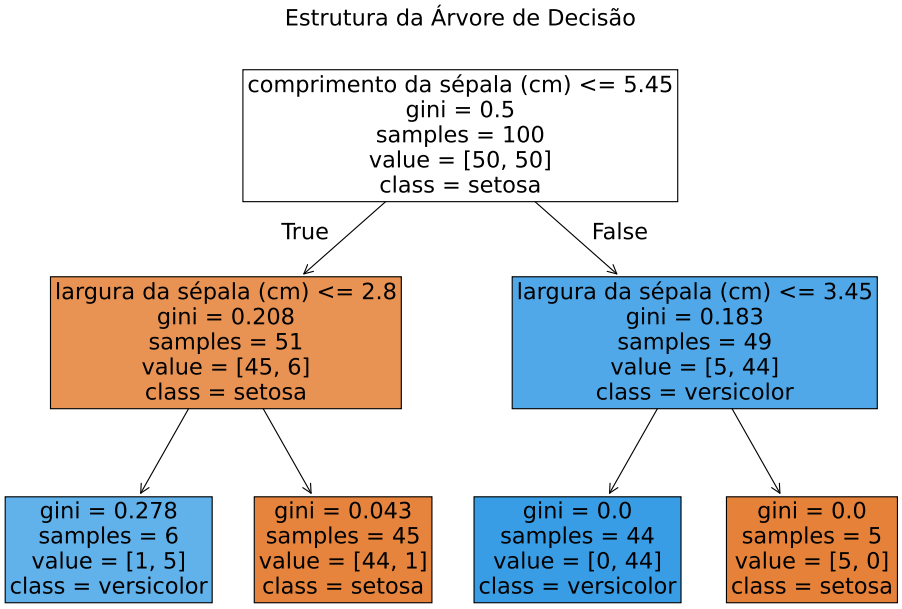

Figura 6: Estrutura de uma Árvore de Decisão treinada com o *dataset* Iris (profundidade máxima de 3) utilizando-se duas *features* (comprimento e largura da sépala) e duas classes (Setosa e Versicolor).

2.1.5.2 *Bagged Trees Ensemble*

Árvores de decisão são sensíveis a mudanças nos dados (alta variância) [28], porém ao agruparmos (método de *ensemble*) diversos modelos de árvores de decisão conseguimos reduzir essa variância [23]. A técnica de *Bootstrap Aggregating* (**Bagging**) é um método de *ensemble* (agrupamento) projetado especificamente para reduzir essa variância. O *Bagging* treina múltiplas árvores em diferentes subconjuntos de dados (criados por amostragem com reposição) e agrega as previsões através de uma votação majoritária [23, 28].

Ao fazer a média das previsões de muitas árvores, a variância do modelo final é reduzida, levando a maiores robustez e desempenho de classificação. O resultado, como visto na Figura 7, é uma fronteira de decisão muito mais suave e complexa do que as "caixas"de uma única árvore de decisão. Em contrapartida, uma desvantagem seria um aumento no uso de armazenamento (múltiplas árvores) e no tempo de inferência.

2.1.5.3 *K-Nearest Neighbors* (KNN)

O algoritmo KNN (*k-Nearest Neighbors*) é um método não paramétrico e baseado em instâncias, o que significa que ele não aprende um modelo explícito a partir dos dados de treinamento, em vez disso, armazena todo o conjunto de dados [31]. Para classificar uma nova amostra, o algoritmo calcula a distância entre este novo ponto e todos os pontos do conjunto de treinamento. A métrica de distância utilizada é a **distância Euclidiana**, que é definida para dois pontos **p** e **q** num espaço de n dimensões como:

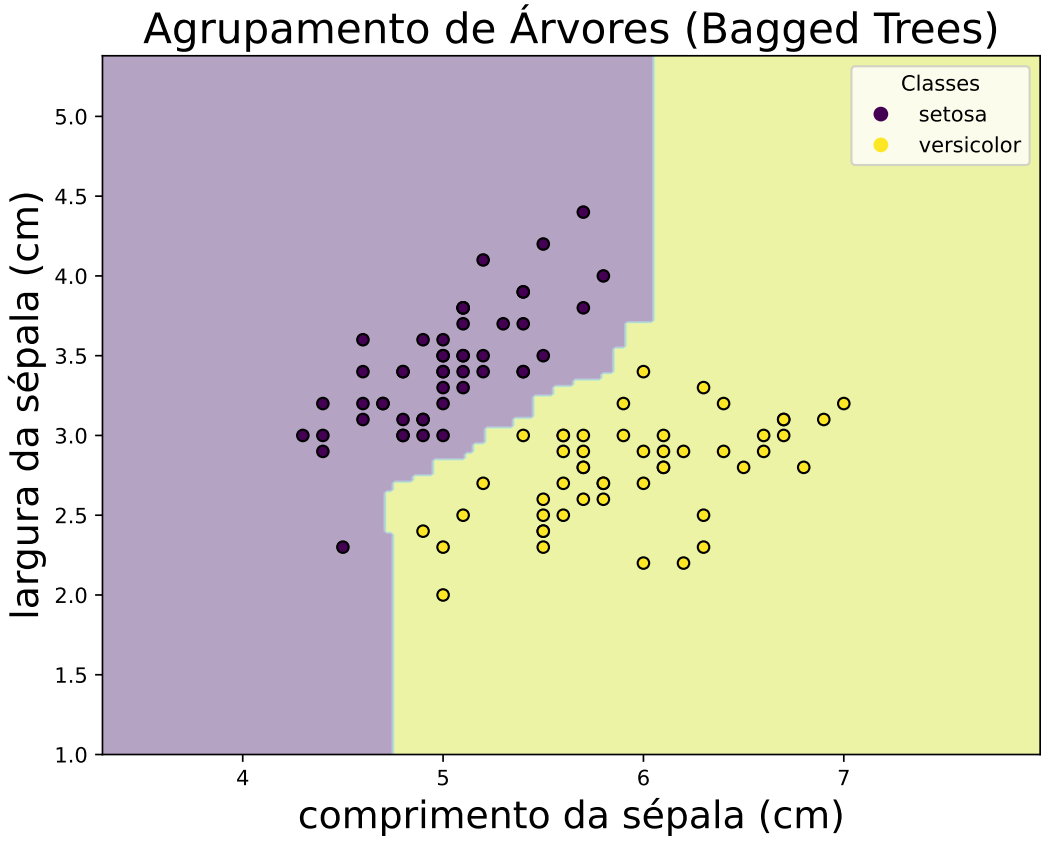

Figura 7: Fronteira de decisão de um *ensemble* de árvores. A média das previsões de múltiplas árvores resulta numa fronteira mais complexa e generalista.

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2} \tag{14}$$

Após o cálculo das distâncias, o algoritmo identifica os K vizinhos mais próximos e atribui à nova amostra a classe que for majoritária entre eles. A escolha do hiperparâmetro K correto é de grande importância, já que um valor pequeno torna o modelo sensível a ruído, enquanto um valor grande aumenta o custo computacional e pode suavizar excessivamente a fronteira de decisão. Na Figura 8 pode-se ver a fronteira de decisão de um modelo KNN treinado com K=7.

A simplicidade do KNN é uma vantagem, mas a necessidade de armazenar todos os dados e de calcular as distâncias a cada inferência representa um grande problema para plataformas com recursos limitados [30].

2.1.5.4 *Naïve Bayes*

O Naïve Bayes é um classificador probabilístico *paramétrico* cujo objetivo é calcular a probabilidade de uma amostra pertencer a uma determinada classe, dadas as suas *features*. O seu funcionamento é fundamentado no **Teorema de Bayes** (equação 15), que permite inverter a probabilidade condicional: em vez de calcular diretamente $P(C_k|\mathbf{x})$ (probabilidade de C_k dado x) , o modelo calcula a probabilidade da classe, $P(C_k)$, e as probabilidades das *features* dada a classe, $P(\mathbf{x}|C_k)$ [24, 28].

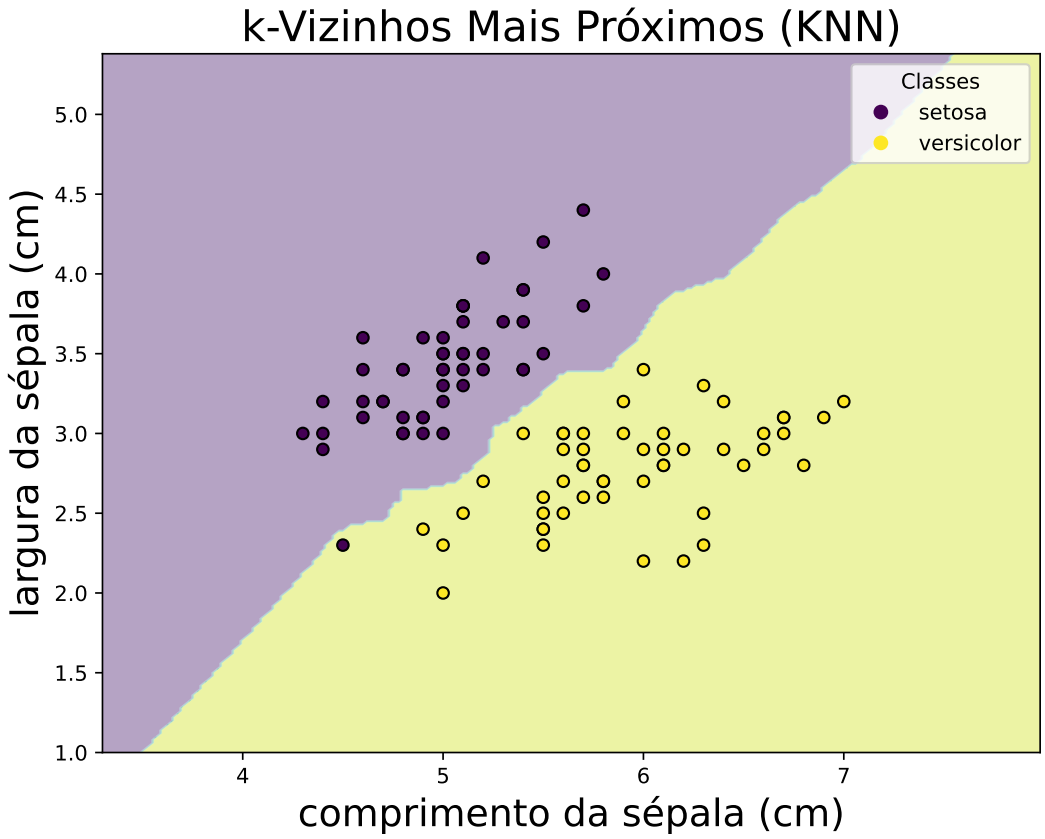

Figura 8: Fronteira de decisão um modelo KNN com k=7 para o exemplo de duas *features* e duas classes do *dataset* Iris

$$P(\mathbf{A}|\mathbf{B}) = \frac{P(\mathbf{B}|\mathbf{A}) \cdot P(\mathbf{A})}{P(\mathbf{B})} \tag{15}$$

onde:

- $P(\mathbf{A}|\mathbf{B})$ é a probabilidade condicional do evento **A** acontecer dado o evento **B** ocorreu;
- $P(\mathbf{B}|\mathbf{A})$ é a probabilidade condicional do evento **B** acontecer dado o evento **A** ocorreu;
- $P(\mathbf{A})$ é a probabilidade do evento **A**;
- $P(\mathbf{B})$ é a probabilidade do evento **B**.

O termo *"Naïve"* (ingênuo) vem da principal suposição do algoritmo: a de que todas as *features* são independentes entre si [28]. Esta suposição, embora dificilmente seja verdadeira na prática, simplifica drasticamente o problema, permitindo que a probabilidade conjunta seja calculada como o produto das probabilidades individuais: $P(\mathbf{x}|C_k) = \prod_{i=1}^{n} P(x_i|C_k)$.

Dentre os tipos de classificadores *"Naïve" Bayes*, o modelo ***Naïve Bayes Gaussiano***, é modelo utilizado para dados contínuos e assume-se que a probabilidade de cada *feature* segue uma distribuição Gaussiana [31]. O modelo então "aprende" os parâmetros desta distribuição (média e variância) para cada classe a partir dos dados de treinamento. O resultado, como se pode ver na Figura 9, é uma fronteira de decisão suave e quadrática, com vantagem de ser um modelo extremamente rápido e eficiente em termos de memória.

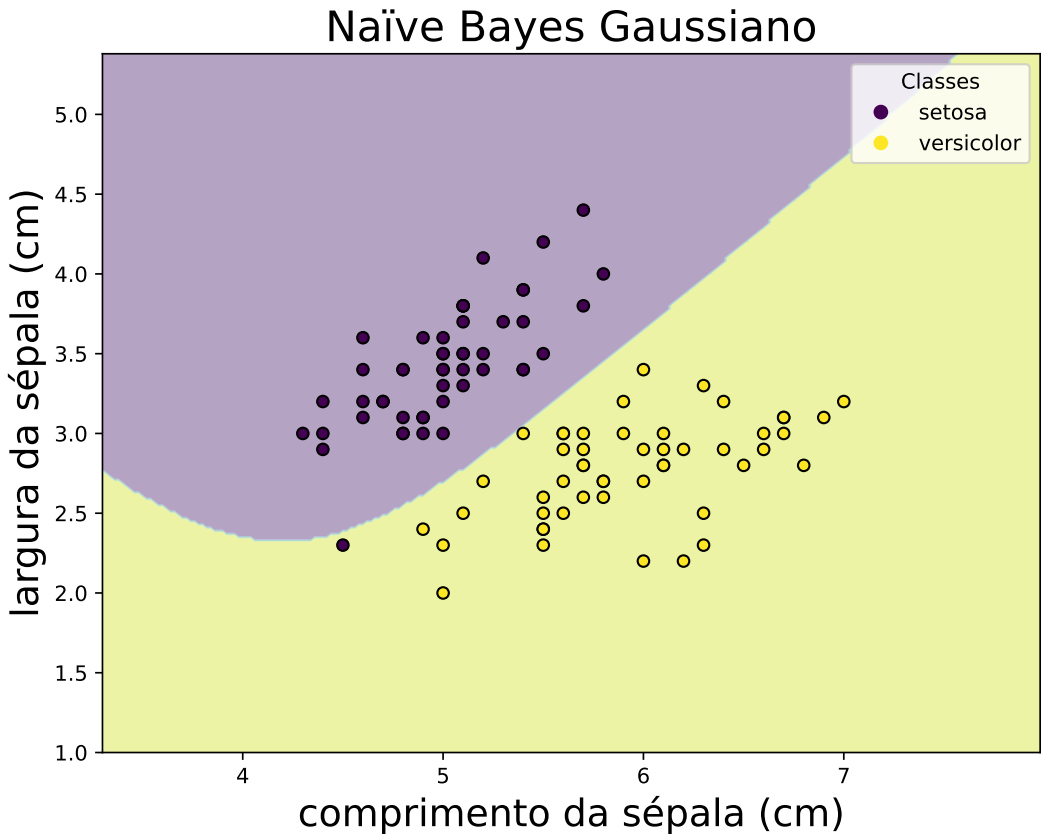

Figura 9: Fronteira de decisão do Naïve Bayes Gaussiano para o exemplo de duas *features* e duas classes do *dataset* Iris.

2.1.5.5 *Support Vector Machines* (SVM)

As Máquinas de Vetores de Suporte (**SVN**, do inglês *Support Vector Machines*) são um conjunto de modelos **paramétricos** que usam uma abordagem de "máxima margem". Em vez de simplesmente encontrar uma linha que separe as classes, o SVM busca o hiperplano que cria a maior margem possível, isto é, tenta maximizar a distância entre o hiperplano de decisão e os pontos de dados mais próximos de cada classe [23].

Em muitos problemas reais, os dados não podem ser separadas por um linha reta (não lineares). Para lidar com esses tipos de dados, a SVM usa uma técnica conhecida como "truque do *kernel*" *(kernel trick)* [32]. Uma **função de *kernel*** é, em essência, uma função que mede a "similaridade" entre dois pontos. O truque consiste em utilizar esta função para calcular a relação entre os pontos como se estivessem num espaço de *features* de dimensão superior, mas sem nunca ter de realizar explicitamente a computação custosa nesse espaço [33, 34]. Uma **SVM Quadrático** é um caso específico que utiliza um *kernel* polinomial de grau 2, resultando numa fronteira de decisão não-linear (uma cônica), como se pode ver na Figura 10.

Por fim, a SVM lida com pontos que não são perfeitamente separáveis através do hiperparâmetro de regularização **C** [30]. Este parâmetro controla o compromisso entre maximizar a margem e minimizar o erro de classificação nos dados de treino [30, 31]. Um valor de **C baixo** permite uma margem mais larga, tolerando que alguns pontos fiquem do lado errado da margem (um modelo mais "suave"). Um valor de **C alto** força o modelo a classificar corretamente o máximo de pontos de treino possível, resultando numa margem mais estreita e num modelo que pode ser mais propenso a sobreajuste (*overfitting)* [30].

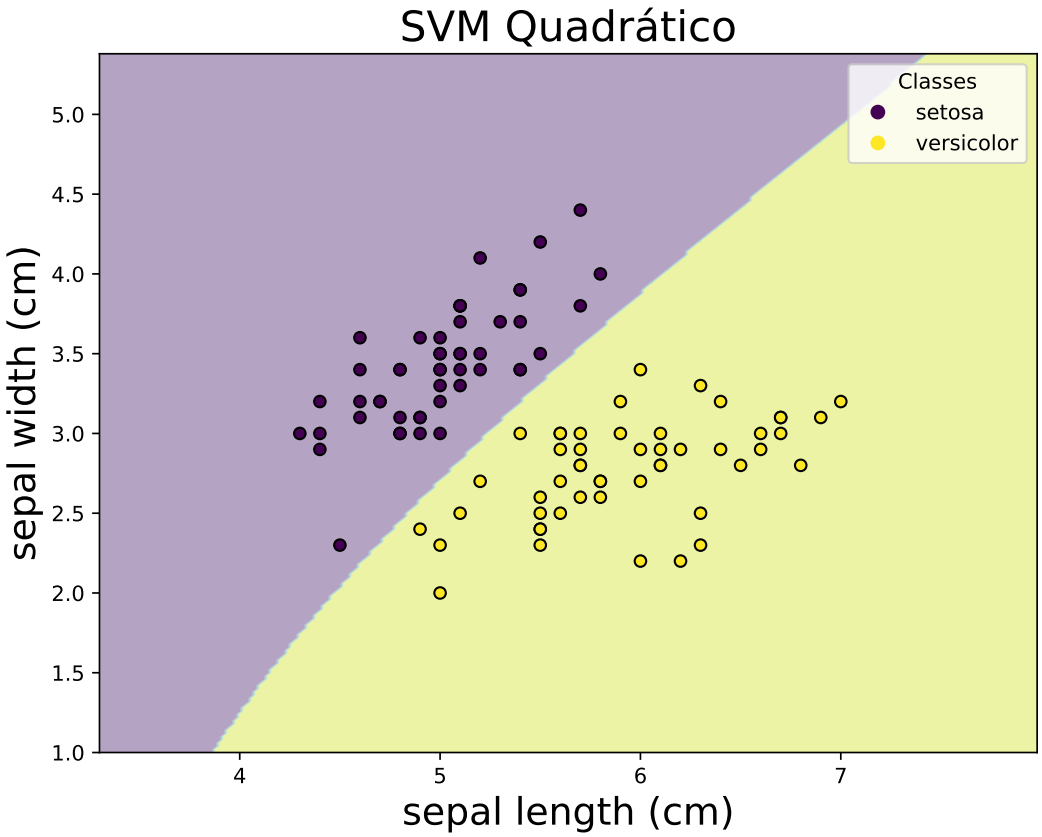

Figura 10: Fronteira de decisão de um SVM Quadrático para o exemplo de duas *features* e duas classes do *dataset* Iris.

2.1.6 Avaliação dos Modelos

Para avaliar e comparar o desempenho destes modelos, utiliza-se uma **matriz de confusão**. Esta matriz é uma tabela que resume os resultados da classificação, comparando as classes previstas pelo modelo com as classes reais (Tabela 1). A partir dela, é possível calcular métricas de desempenho importantes, como a acurácia [9, 30].

Tabela 1: Exemplo de Matriz de Confusão para um problema de duas classes.

		Classe Prevista	
		Positivo	Negativo
Classe Real	Positivo	Verdadeiro Positivo (TP)	Falso Negativo (FN)
	Negativo	Falso Positivo (FP)	Verdadeiro Negativo (TN)

A **acurácia** mede a proporção de previsões corretas (a soma dos Verdadeiros Positivos e Verdadeiros Negativos) sobre o total de previsões realizadas. É a métrica utilizada para comparação neste trabalho [9, 30] e pode ser calculada utilizando a Equação 16. A acurácia foi adotada como métrica para manter a comparabilidade com o trabalho de referência [9].

$$\text{Acurácia} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{16}$$

2.2 *Machine Learning* Embarcado (*Edge ML*)

A abordagem tradicional para a implementação de modelos de Aprendizagem de Máquina envolve a sua execução em servidores na nuvem (*cloud*), que oferecem alto poder computacional. No entanto, esta arquitetura centralizada exige a transmissão contínua de dados brutos, o que gera desvantagens para aplicações industriais, como alta latência, consumo elevado de banda, e dependência de uma conexão de internet estável e segura [35].

Como solução para estes problemas, surge o paradigma do *Machine Learning* Embarcado (aprendizado de maquina embarcado), também conhecido como *Edge ML* [36]. Esta abordagem consiste em executar a etapa de inferência do modelo diretamente no dispositivo de "borda" (*edge*), ou seja, no próprio microcontrolador que está próximo ao sensor. Esta mudança de paradigma traz vários benefícios relevantes:

- **Baixa latência**: processamento em tempo real, essencial para detecção imediata de falhas;
- **Economia de banda**: transmissão apenas de resultados ou alertas, não dos dados brutos;
- **Maior privacidade**: dados sensíveis não precisam deixar o ambiente local;
- **Autonomia**: operação offline em locais sem cobertura de rede.

Contudo, a implementação de ML em microcontroladores apresenta restrições de recursos, principalmente em termos de memória (RAM e Flash) e capacidade de processamento [35, 36]. A superação destes desafios exige uma seleção cuidadosa tanto do hardware quanto do software. Neste trabalho, a plataforma de hardware escolhida é a **STM32 Nucleo-F446RE**, equipada com um microcontrolador **ARM Cortex-M4**. A sua Unidade de Ponto Flutuante (**FPU**, do inglês *Float Point Unit*) e as instruções de Processamento Digital de Sinais (**DSP**, do inglês *Digital Signal Processing*) são particularmente adequado para os cálculos matemáticos exigidos pela extração de *features* e pela inferência do modelo [37]. Como detalhado na Tabela 2, as suas especificações são comparáveis às da placa utilizada no artigo de referência [9], tornado-a uma plataforma robusta para este projeto.

Existem diversas ferramentas que facilitam a transição do ambiente de desenvolvimento para a plataforma embarcada:

- **Emlearn**: converte modelos do Scikit-learn para código C puro, adequado a Microcontroladores (**MCU**) com recursos limitados [11].
- **TensorFlow Lite for Microcontrollers**: otimizado para redes neurais em hardware com pouca memória [38].
- **microTVM**: permite compilação e otimização de modelos para diferentes arquiteturas de MCU [39].

Neste trabalho, optou-se pelo *Emlearn* por sua compatibilidade com modelos baseados em árvores de decisão e seu baixo *overhead* de execução, além da simplicidade de integração com o firmware em C. O *Emlearn* converte o modelo treinado e gera arquivos ('.c' e '.h') contendo uma função de predição, permitindo que o modelo seja facilmente integrado ao *firmware* do microcontrolador STM32.

Tabela 2: Tabela Comparativa de Especificações de Hardware

Especificação	**STM32 Nucleo-F446RE**	**STM32F407VG Discovery**
Microcontrolador	STM32F446RET6	STM32F407VGT6
Core da CPU	ARM Cortex-M4	ARM Cortex-M4
FPU (Ponto Flutuante)	Sim (precisão simples)	Sim (precisão simples)
Frequência Máx. CPU	180 MHz	168 MHz
Memória Flash	512 KB	1024 KB (1 MB)
Memória SRAM	128 KB	192 KB
Tensão de Operação	1.7V – 3.6V	1.8V – 3.6V
Conversores A/D	3x 12-bit (16 canais)	3x 12-bit (16 canais)
Conversores D/A	2x 12-bit	2x 12-bit

3 Metodologia

3.1 Visão Geral

Como descrito na sessão 1.1, o objetivo deste trabalho está na replicação da metodologia proposta por Shubita et al. [9], avaliando a possibilidade e a eficácia da utilização de um ecossistema de ferramentas de código aberto baseado em Python em vez da plataforma proprietária MATLAB, utilizada no estudo original. Para alcançar este objetivo, foi estabelecido um fluxo de trabalho que abrange desde o processamento dos dados em Python até a implementação e a validação em um sistema embarcado. Todo o processo, desde a extração de características até a implementação no sistema embarcado, que foi desenvolvido como parte deste trabalho, e o código-fonte completo estão disponíveis para consulta Online [40].

O fluxograma apresentado na Figura 11 ilustra as etapas sequenciais do projeto. Inicialmente, os dados de áudio são processados em em um *notebook* Python, onde as características relevantes (*features*) são extraídas. Em seguida, modelos de aprendizado de máquina são treinados e avaliados com base nessas *features*. Feito isso, os modelos são então convertidos para código C e integrados ao microcontrolador STM32. Finalmente, a implementação embarcada é validada, e seus resultados são comparados com os obtidos no ambiente Python e com os resultados do artigo base. Vale notar que, para garantir a fidelidade ao artigo original, utilizou-se dos arquivos de áudio disponibilizados no repositório do artigo base.

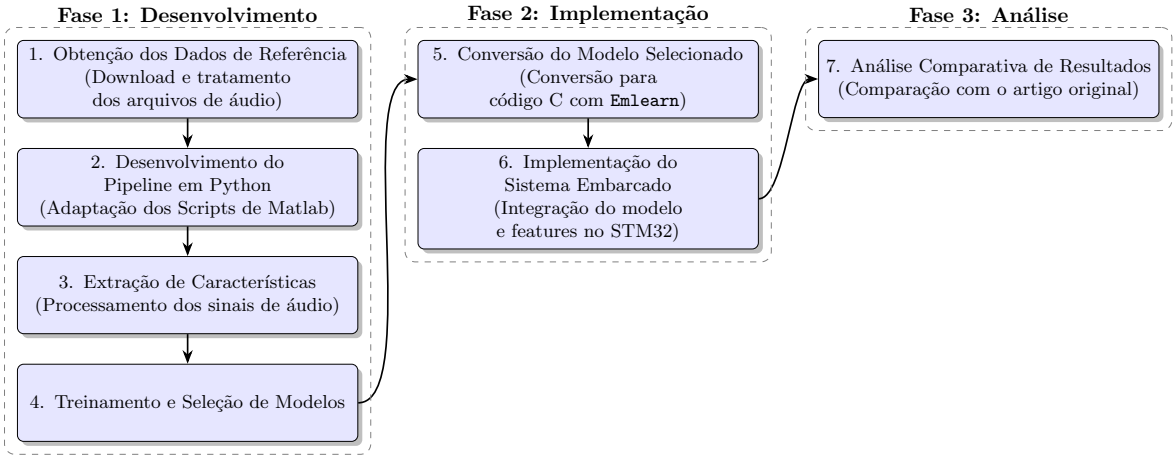

Figura 11: Fluxograma da visão geral das etapas do trabalho, mostrando as fases de desenvolvimento em Python, implementação na plataforma STM32 e análise de resultados.

3.2 Ambiente de Desenvolvimento e Conjunto de Dados

O desenvolvimento deste trabalho foi executado em duas frentes principais. A prototipagem, extração de *features* e treinamento dos modelos de aprendizado de máquina foram conduzidos em **Python**, utilizando o editor de código *Visual Studio Code* [41]. O gerenciamento das bibliotecas e do ambiente foi realizado com o auxílio da ferramenta *Poetry*, garantindo a reprodutibilidade do projeto. As principais bibliotecas utilizadas foram `Pandas` para manipulação de dados, `Scikit-learn` para o treinamento dos classificadores, `Soundfile` para leitura de arquivos de áudio e `emlearn` para a conversão dos modelos para serem utilizados na plataforma embarcada. A implementação no sistema embarcado foi desenvolvida em **linguagem C**,utilizando o ambiente de desenvolvimento integrado (**IDE**, do inglês *Integrated Development Environment*) `STM32CubeIDE` [42], fornecido pela STMicroelectronics e ilustrado na Figura 12.

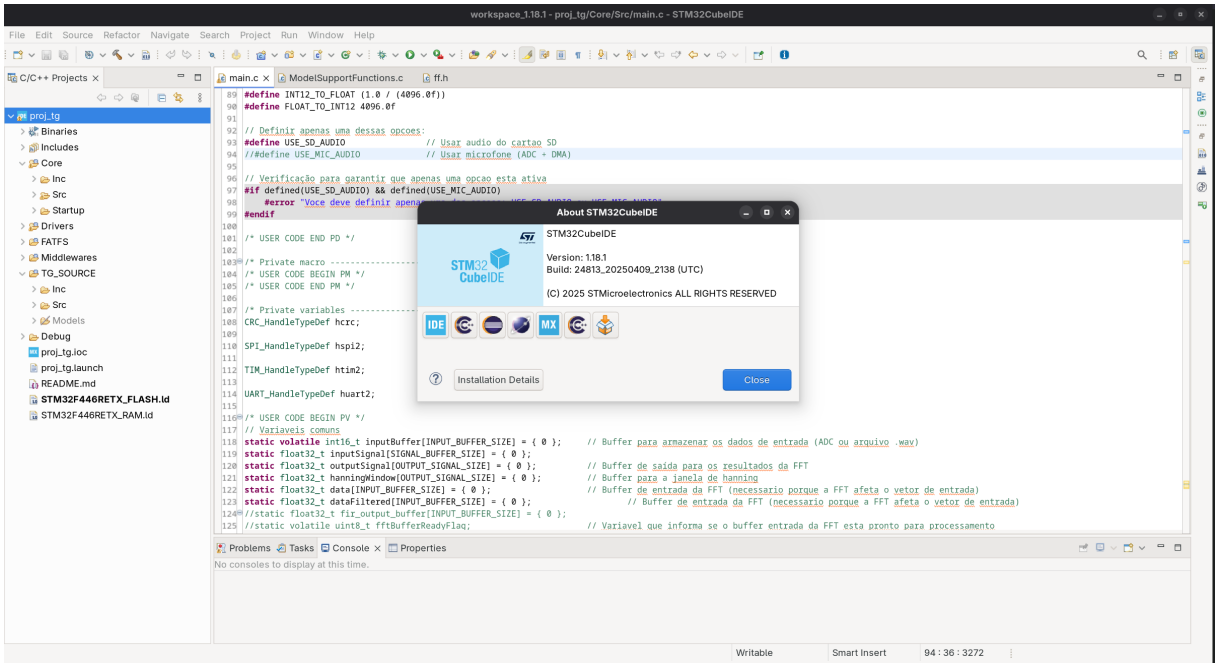

Figura 12: Imagem da ferramenta de desenvolvimento *STM32CubeIDE*

O conjunto de dados, proveniente do estudo de Shubita et al. [9], foram disponibilizados no repositório do projeto [43] junto alguns arquivos fonte MATLAB e as imagens utilizadas no artigo. Os arquivos de áudio disponibilizados foram capturados a partir de uma furadeira comercial (CROWN CT10128) [9] como fonte sonora, gravando o sinal de áudio com um smartphone a 10 cm de distância, conforme mostrado na Figura 13. O áudios capturados foram disponibilizados no formato FLAC, com dois canais (estéreo), uma taxa de amostragem de 48 kHz e 24 bits de quantização por amostra. Foram capturados sinais em cinco condições [9]:

- Saudável: condição na qual o equipamento não apresenta defeito algum;
- Desligado: apenas o som ambiente é registrado, sem interferência do som produzido pelo equipamento;
- Falha no rolamento: uma pequena partícula foi inserida no interior do rolamento para induzir um problema real;
- Falha na ventoinha: a falha foi produzida danificando duas das pás da ventoinha, causando desbalanceamento mecânico;
- Falha na engrenagem: dois dentes da engrenagem foram danificados para produzir o som da falha.

Pelo fato do formato FLAC ser comprimido, isso exigiria que tempo de processamento e memória fossem gastos para descompactar os dados para que pudessem ser utilizados para a extração das *features*. Por conta disso, a conversão dos arquivos de áudio de FLAC para WAV foi necessária. O formato WAV armazena o áudio de forma não comprimida e possui um cabeçalho simples e de estrutura definida [44], conforme ilustrado na Figura 14. Essa estrutura permite a leitura dos metadados essenciais, como a taxa de amostragem, e acesso aos dados de áudio brutos de forma direta e com baixo custo computacional.

De modo a avaliar a capacidade de generalização do modelo, foram feitos testes com dados que não haviam sido utilizados previamente para o treinamento. Uma prática comum é a divisão dos dados de áudio em conjuntos de treinamento e de teste (validação) [30, 31]. Assim, para cada arquivo de áudio, os primeiros 80% de cada gravação foram separados para compor o conjunto de dados de treinamento

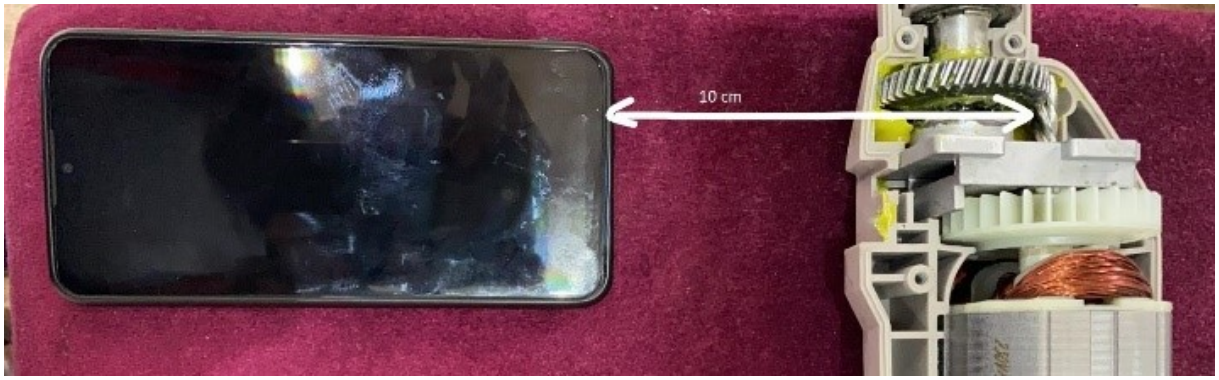

Figura 13: Configuração dos equipamentos de aquisição dos dados utilizado no trabalho de referência para a gravação dos sinais de áudio da furadeira [9, 43].

Estrutura do Cabeçalho de um Arquivo WAV (44 bytes)

Offset (Bytes)	Tamanho (Bytes)	Nome do Campo
0	4	ChunkID ("RIFF")
4	4	ChunkSize
8	4	Format ("WAVE")
12	4	Subchunk1ID ("fmt ")
16	4	Subchunk1Size
20	2	AudioFormat
22	2	NumChannels
24	4	SampleRate
28	4	ByteRate
32	2	BlockAlign
34	2	BitsPerSample
36	4	Subchunk2ID ("data")
40	4	Subchunk2Size
44	...	Dados de Áudio (PCM)...

Figura 14: Estrutura do cabeçalho padrão de 44 bytes de um arquivo WAV.

dos modelos. Os 20% restantes foram utilizados como o conjunto de teste. Esses arquivos de teste foram utilizados tanto para testar a acurácia dos modelos em Python no *Desktop* quanto para a validação final da implementação no sistema embarcado.

3.3 Extração das *Features* com Python

Nesta etapa, realizou-se a extração de características (*features*) dos dados brutos de áudio dos conjuntos de treinamento e validação, armazenando-as em um conjunto estruturado de dados para servirem de entrada no treinamento dos modelos de aprendizado de máquina. Com esse objetivo, todo o processo de tratamento dos sinais e extração das *features* foi implementado no notebook Python (`FeatureExtractor.ipynb`). A implementação seguiu a metodologia descrita no artigo de referência [9], sendo adaptada a partir dos Scripts originais em MATLAB disponibilizados pelos autores [43]. O processo foi estruturado da seguinte maneira:

1. **Leitura e Preparação do Sinal:** Os arquivos de áudio de cada condição foram lidos e as primeiras 40980 amostras de cada sinal foram descartados para remover o silêncio inicial da gravação.

2. **Segmentação com Sobreposição:** O sinal foi então segmentado utilizando uma técnica de janela deslizante com 75% de sobreposição (*overlap*). A cada iteração, uma nova janela de 2048 amostras (`BUFFER_LEN`) foi extraída, avançando apenas 512 amostras (25% do tamanho da janela) em relação à anterior, seguindo o procedimento descrito em [9]. Esta técnica aumenta a quantidade de dados para treinamento (cada janela de sinal origina quatro janelas de análise) e auxilia na recuperação das informações das bordas, que são atenuadas pela aplicação da janela de Hann na etapa seguinte.

3. **Pré-processamento por Janela:** Cada janela de dados extraída passou por uma sequência de pré-processamento para garantir a qualidade do sinal antes da extração das *features*.

    - **Filtragem Passa-Faixa:** Primeiramente, foi aplicado um filtro digital para isolar as frequenci as de interesse e atenuar ruídos. Foi utilizado um filtro de Impulso de Duração Infinita (IIR, do inglês *Infinite Impulse Response*) *Butterworth* passa-faixa de 8ª ordem, projetado para operar na mesma frequência de amostragem dos dados (48 kHz) e com frequências de corte em 20 Hz e 20.000 Hz. A sua arquitetura, implementada no arquivo `ExtractFeatures.py`, consiste em uma cascata de 4 seções de segunda ordem (SOS, do inglês *Second-Order Sections*), uma estrutura que garante maior estabilidade numérica.

    - **Janelamento de Hann:** Após a filtragem, uma janela de Hann foi aplicada, multiplicando-se ponto a ponto os dados da janela filtrada. O objetivo do janelamento é suavizar as bordas do sinal, forçando as amplitudes no início e no fim do segmento a se aproximarem de zero [45]. Este procedimento é importante para diminuir o efeito de "vazamento espectral" (*spectral leakage*) que ocorre durante o cálculo da FFT [46], garantindo assim uma representação mais precisa das componentes de frequência do sinal [15, 46].

4. **Cálculo das *Features*:** Após o pré-processamento, o conjunto completo de 16 *features* foi calculado a partir de cada janela de dados tratada, abrangendo tanto o domínio do tempo quanto o da frequência conforme o artigo base [9].

    - **Domínio do Tempo (10 *features*):** Foram calculadas/extraídas as métricas de Média, Mediana, RMS, Variância, *Shape Factor*, *Impulse Factor*, *Crest Factor*, *Margin Factor*, *Kurtosis* e *Skewness*.

    - **Domínio da Frequência (6 *features*):** A partir da FFT de cada segmento, foram identificadas a amplitude e a localização dos três picos espectrais de maior proeminência (`Peak1`, `PeakLocs1`, `Peak2` etc.).

5. **Estruturação e Armazenamento:** As *features* extraídas, juntamente com a classe identificando o tipo da falha (`FaultID`), foram armazenadas em um *DataFrame* da biblioteca `pandas` e exportadas para um arquivo `.csv`, para serem utilizadas como entrada para a etapa de treinamento.

3.4 Treinamento e Conversão dos Modelos

O treinamento, a avaliação (utilizando os dados de teste) e a exportação dos modelos de aprendizado de máquina foram realizados no notebook `TrainClassifier.ipynb`. Utilizando os arquivos `.csv` gerados na etapa anterior, os diferentes modelos (algoritmos) de classificação foram explorados para identificar o mais adequado para a tarefa.

A partir do conjunto de 16 *features* extraídas na etapa de extração das *features*, apenas 14 foram utilizadas para o treinamento dos modelos. As *features* de Média e Mediana foram removidas para manter

comparabilidade com o trabalho de referência [9]. No estudo original, os autores utilizaram a metodologia de Análise de Variância (**ANOVA**, do inglês *Analysis of Variance*) para ordenar a importância das *features*. A ANOVA é um teste estatístico que avalia a relevância de uma *feature* ao medir se seus valores médios diferem de maneira significativa entre as diferentes classes de falha; *features* com baixa variação entre as classes são consideradas menos informativas [19, 28]. Utilizando a ANOVA, os autores do artigo base notaram que a Média e a Mediana eram as *features* de menor relevância e a remoção não resultou em uma mudança significativa na acurácia do modelo final [9]. Por isso as *features* de Média e Mediana foram excluídas antes da etapa de treinamento.

O processo de treinamento foi dividido em duas fases: treinamento de modelos para *"desktop"* e treinamento de modelos compatíveis com o sistema embarcado. Foram utilizados os seguintes classificadores da biblioteca `scikit-learn` (hiperparâmetros não mencionados foram mantidos seus valores padrão da biblioteca):

- ***Decision Tree*** com hiperparâmetros:
  - `criterion='gini'`
  - `max_depth=None`
  - `random_state=42`
- ***Bagged Trees Ensemble*** com hiperparâmetros:
  - `estimator=DecisionTreeClassifier(max_depth=None, criterion='gini')`
  - `n_estimators=3`
  - `random_state=42`
- ***Random Forest*** com hiperparâmetros:
  - `n_estimators=3`
  - `max_depth=None`
  - `random_state=42`
- ***Gaussian Naive Bayes*** com hiperparâmetros:
  - `var_smoothing=1e-9`
- ***K-Nearest Neighbors (KNN)*** com hiperparâmetros:
  - `n_neighbors=3`
  - `weights='uniform'`
- ***Support Vector Machine (SVM)*** com hiperparâmetros:
  - `kernel='poly'`
  - `degree=2`
  - `C=2`

Para a implementação no microcontrolador, a biblioteca `emlearn` foi utilizada. Ela permite converter modelos treinados no ambiente Python para um código na linguagem C portátil. O processo de exportação, realizado pela função `emlearn.convert()`, gera um arquivo de cabeçalho (`.h`) que contém o modelo serializado e as funções necessárias para realizar a inferência, como `model_predict()`.

Uma restrição importante foi identificada durante a fase de exportação dos modelos: a versão da biblioteca `emlearn` utilizada neste trabalho não oferece suporte para a conversão de todos os classificadores disponíveis no `scikit-learn`. Em especial, os modelos *Quadratic SVM* e *Bagged Trees Ensemble*, que foram treinados com no ambiente Python para estabelecer uma base comparação com o artigo de referência [9], não puderam ser convertidos para serem testados no sistema embarcado.

Diante dessa limitação, a análise desses modelos ficou restrita ao desempenho em *desktop*. Para substituir o modelo *Bagged Trees Ensemble* na etapa de validação embarcada, foi selecionado o classificador *Random Forest*. Essa escolha foi feita pelo fato desse algoritmo ser da mesma família dos métodos de agrupamento (*ensemble*), que frequentemente apresenta desempenho superior ao *Bagged Trees* por introduzir uma camada adicional de aleatoriedade na seleção das características em cada nó da árvore, o que contribui para reduzir a correlação entre as árvores e, consequentemente, a variância do modelo final [28, 30].

3.5 Implementação no Sistema Embarcado

A etapa final consistiu na implementação e validação de todo o fluxo de processamento e inferência no microcontrolador da placa de desenvolvimento STM32 Nucleo-F446RE [37]. O código-fonte, desenvolvido no ambiente STM32CubeIDE [42], foi estruturado de forma modular para criar um sistema capaz de replicar em hardware as etapas realizadas em Python.

3.5.1 Configuração Física e Conexões

O hardware utilizado consiste na placa Nucleo-F446RE conectada a um módulo de leitor de cartão MicroSD genérico. A comunicação entre o microcontrolador STM32 e o módulo de cartão SD foi estabelecida através da Interface Periférica Serial (**SPI**, do inglês *Serial Peripheral Interface*). O SPI é um protocolo de comunicação serial síncrono que opera em modo *full-duplex* com uma arquitetura mestre-escravo (*master-slave*) [47]. A comunicação é gerenciada por quatro linhas de sinal principais:

- **SCK** (Serial Clock), que sincroniza a transferência de dados;
- **MOSI** (Master-Out, Slave-In) para o envio de dados do mestre ao escravo;
- **MISO** (Master-In, Slave-Out) para o envio de dados do escravo de volta ao mestre;
- **CS** (Chip Select), um sinal ativo baixo que o mestre utiliza para selecionar com qual dispositivo escravo deseja se comunicar.

Neste trabalho, o microcontrolador STM32 atua como mestre, enquanto o módulo de cartão SD opera como o dispositivo escravo. A Figura 15 ilustra a montagem do protótipo e a Figura 16, o seu diagrama de conexão.

3.5.2 Estrutura do *Firmware* e Módulos de Software

O *firmware* foi desenvolvido em linguagem C e organizado em módulos distintos para facilitar a manutenção e a clareza do código, conforme detalhado nos arquivos do repositório do projeto [40] dentro da pasta `stm32_code/`.

3.5.2.1 Gerenciamento de Arquivos com FatFs

Para a leitura dos arquivos de áudio `.wav` e a escrita dos resultados em `.csv`, foi utilizada a biblioteca de sistema de arquivos *middleware* `FatFs`. A partir dela, foi criada uma camada de abstração

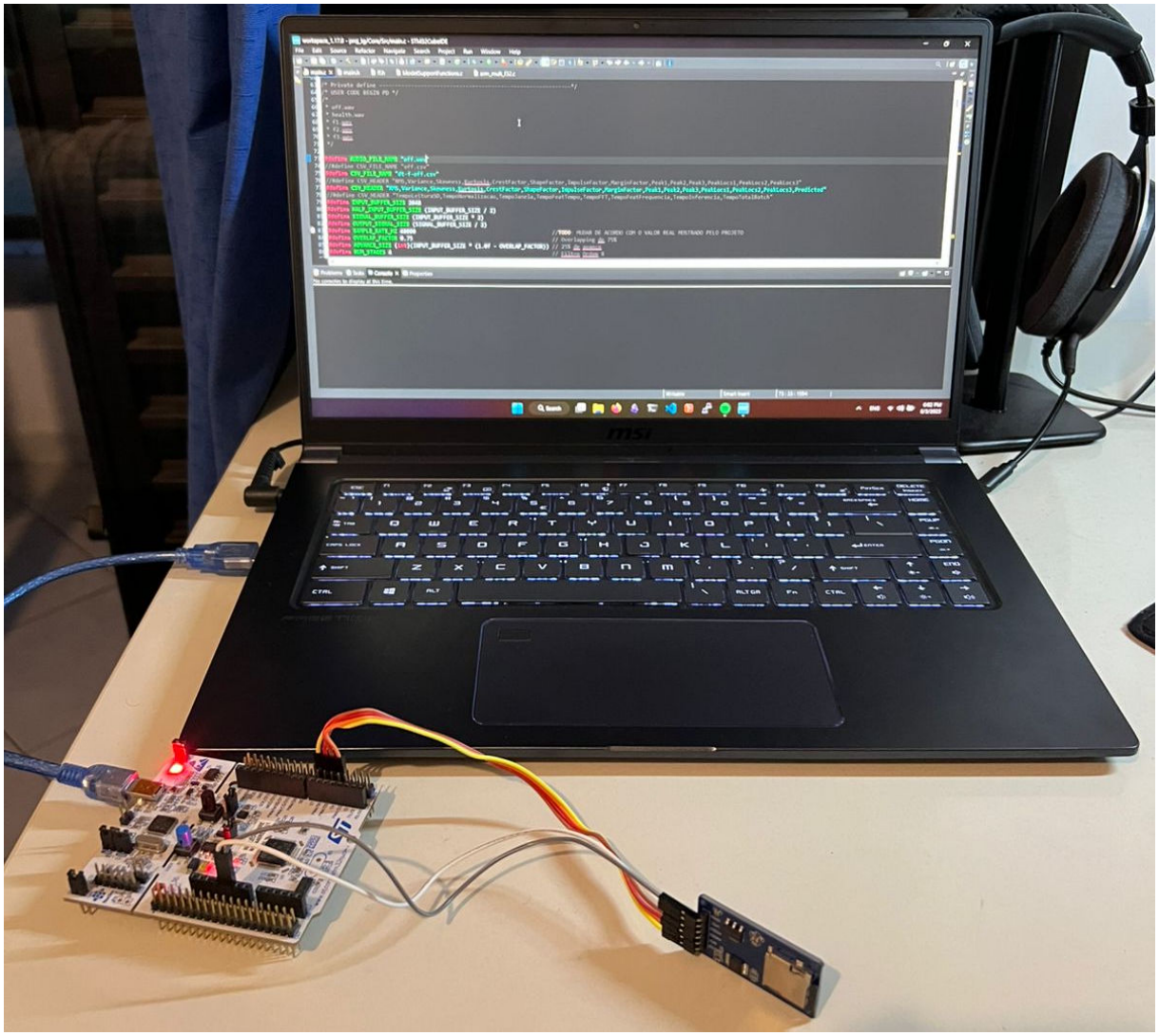

Figura 15: Configuração do sistema embarcado.

no arquivo `SDCard.c` para simplificar as operações de leitura e escrita. Funções como `SDCard_Mount()`, `SDCard_OpenFile()` e `SDCard_WriteLine()` foram criadas para esconder/encapsular as chamadas de baixo nível da feitos na biblioteca `FatFs`, deixando o código principal no arquivo `main.c` mais simples e legível.

3.5.2.2 Extração de *Features* no Domínio do Tempo

A extração das 10 *features* no domínio do tempo foi implementada na função `extractTimeDomainFeatures()`, localizada no arquivo `FeatureExtraction.c`. Para melhorar o desempenho da implementação, a implementação fez uso extensivo da biblioteca ARM CMSIS-DSP [48]. Funções como `arm_rms_f32()`, `arm_var_f32()` e `arm_std_f32()` foram utilizadas para os cálculos de RMS, variância e desvio padrão, entre outras métricas. A biblioteca CMSIS-DSP é uma coleção de funções de processamento digital de sinais altamente otimizadas para processadores da linha *Cortex-M* desenvolvida pela própria ARM [37].

Para as *features* como *Kurtosis* e *Skewness*, que não possuem funções diretas implementadas na biblioteca CMSIS-DSP, foram implementadas as funções `calculateKurtosis()` e `calculateSkewness()`,

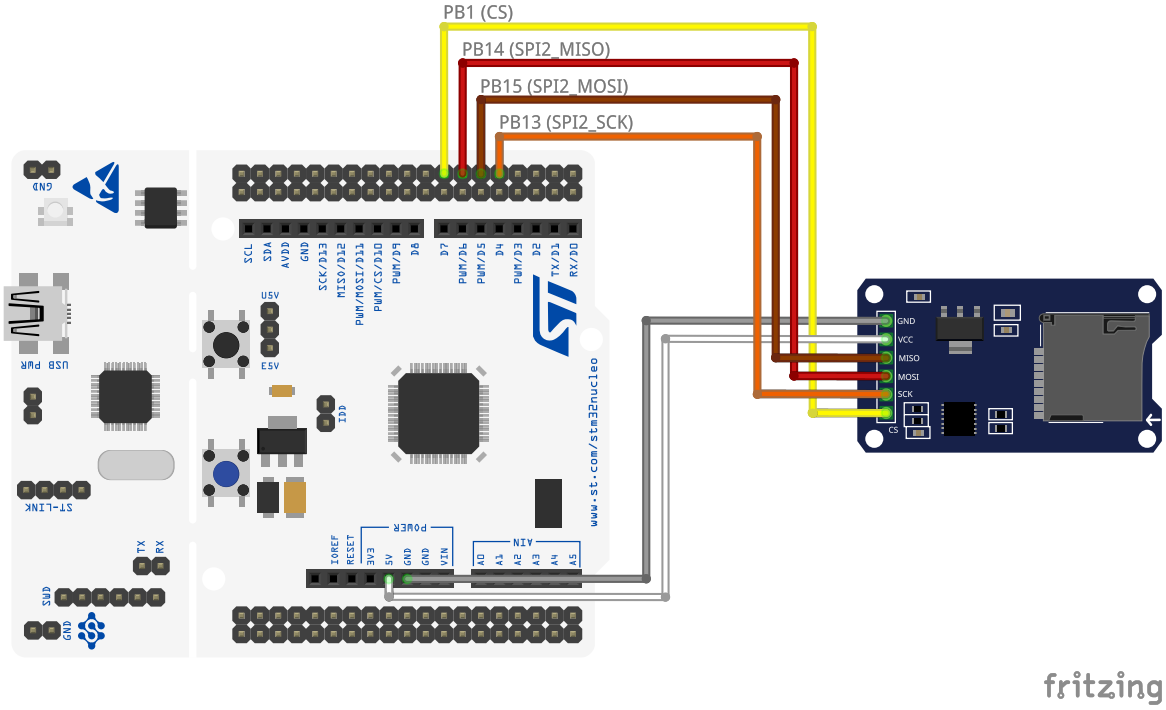

Figura 16: Diagrama de conexão do leitor de Cartão MicroSD a placa STM32

seguindo as definições matemáticas apresentadas na Seção 2 (equações (10) e (11)) e suas implementações podem ser verificadas nas Figuras 17 e 18.

3.5.2.3 Extração de *Features* no Domínio da Frequência

As 6 *features* do domínio da frequência foram calculadas utilizando a função extractFrequencyDomainFeatures(). O processo, implementado em `FeatureExtraction.c`, segue três passos principais para cada janela de dados:

1. **Cálculo da FFT:** O espectro de frequência do sinal é obtido através da função `arm_rfft_fast_f32()` da biblioteca CMSIS-DSP.
2. **Cálculo da Magnitude:** O resultado da FFT é um vetor de números complexos. A magnitude de cada componente de frequência é calculada com a função `arm_cmplx_mag_f32()`, que gera um vetor com os valores de amplitude do espectro.
3. **Identificação dos Picos:** Um laço de repetição e utilizado para percorrer o vetor de magnitudes para identificar os três maiores picos de amplitude (`PeakAmp1`, `PeakAmp2`, `PeakAmp3`) e suas frequências correspondentes (`PeakLocs1`, `PeakLocs2`, `PeakLocs3`), que são então armazenadas na estrutura de dados das *features*.

A implementação da lógica para o cálculo da magnitude e identificação dos picos de frequência pode ser verificado na Figura 19.

3.5.2.4 Execução da Inferência

A lógica de inferência do modelo de aprendizado de máquina também foi encapsulada, neste caso,

```c
float32_t calculateKurtosis(float32_t *signal, uint32_t length, float32_t mean,
    float32_t stdDev) {
    // Inicializa a kurtosis
    float32_t kurtosis = 0.0f;

    if (stdDev == 0 || length == 0) {
        return 0.0f;
    }

    // Calcula a kurtosis usando a formula
    for (uint32_t i = 0; i < length; i++) {
        float32_t diff = signal[i] - mean;
        kurtosis += (diff * diff * diff * diff);
    }

    kurtosis = kurtosis / (length * stdDev * stdDev * stdDev * stdDev);

    return kurtosis;
}

```

Figura 17: Implementação da função para cálculo da *Kurtosis* na Linguagem C.

```c
float32_t calculateSkewness(float32_t *signal, uint32_t length, float32_t mean,
    float32_t stdDev) {
    // Inicializa o skewness
    float32_t skewness = 0.0f;

    if (stdDev == 0 || length == 0) {
        return 0.0f;
    }

    // Calcula o skewness usando a formula
    for (uint32_t i = 0; i < length; i++) {
        float32_t diff = signal[i] - mean;
        skewness += (diff * diff * diff);
    }

    skewness = skewness / (length * stdDev * stdDev * stdDev);

    return skewness;
}

```

Figura 18: Implementação da função para cálculo da *Skewness* na Linguagem Cs.

no arquivo `ModelSupportFunctions.c`. Isso foi feito para facilitar a inferência utilizando os diferentes modelos. A função principal, `run_inference()`, recebe as estruturas de dados com as *features* de tempo e frequência e executa os seguintes passos:

1. **Montagem do Vetor de Entrada:** As 14 *features* são organizadas em um único vetor do tipo `float32_t`, na mesma ordem utilizada durante o treinamento do modelo em Python.
2. **Chamada da Predição:** Este vetor de entrada é então repassado para a função `model_predict()`, que foi gerada pela ferramenta `emlearn` e está contida no arquivo de cabeçalho do modelo (e.g., `decision_tree_model`).
3. **Retorno do Resultado:** A função retorna um valor inteiro representando a classe prevista pelo modelo (`FaultID`).

3.5.2.5 Formatação e Escrita dos Resultados

```c
void extractFrequencyDomainFeatures(FDFeatures *fdFeatures, float32_t *buffer,
    uint16_t bufferSize, uint32_t sampleRate) {

  // Atribui zero para todos os valores da estrutura
  memset(fdFeatures, 0, sizeof(FDFeatures));

  /* Calcula vetor de Magnitudes */
  float32_t fftMag[bufferSize/2];
  arm_cmplx_mag_f32(buffer, fftMag, (bufferSize/2));

  for (int index = 0; index < bufferSize / 2; index += 1) {
    float curVal = fftMag[index];

    if (curVal > fdFeatures->PeakAmp1) {
      fdFeatures->PeakAmp3 = fdFeatures->PeakAmp2;
      fdFeatures->PeakAmp2 = fdFeatures->PeakAmp1;
      fdFeatures->PeakAmp1 = curVal;
      fdFeatures->PeakLocs3 = fdFeatures->PeakLocs2;
      fdFeatures->PeakLocs2 = fdFeatures->PeakLocs1;
      fdFeatures->PeakLocs1 =  (index * (sampleRate / ((float) bufferSize)));
    } else if (curVal > fdFeatures->PeakAmp2) {
      fdFeatures->PeakAmp3 = fdFeatures->PeakAmp2;
      fdFeatures->PeakAmp2 = curVal;
      fdFeatures->PeakLocs3 = fdFeatures->PeakLocs2;
      fdFeatures->PeakLocs2 = (index * (sampleRate / ((float) bufferSize)));
    } else if (curVal > fdFeatures->PeakAmp3) {
      fdFeatures->PeakAmp3 = curVal;
      fdFeatures->PeakLocs3 = (index * (sampleRate / ((float) bufferSize)));
    }
  }
}

```

Figura 19: Implementação da função para extração das features de frequência na Linguagem C.

Para facilitar a validação do sistema, foi criado o módulo `StringFormatter.c`, que contém funções para converter os dados numéricos em *strings* formatadas. A função formatFeaturesAndResultToString() recebe as *features* extraídas e o resultado da inferência, e gera uma única *string* no formato CSV. Essa *string* é então gravada no cartão SD através da função `SDCard_WriteLine()`, permitindo uma análise posterior e a comparação direta com os resultados obtidos no ambiente Python executado no *desktop*.

3.6 Medição de Desempenho e Metodologia de Análise

Para avaliar a viabilidade e a eficiência do *pipeline* de código aberto implementado no *hardware* embarcado, o código implementado na plataforma foi instrumentalizado para medir a latência de cada etapa do processamento. O objetivo desta análise é identificar potenciais gargalos computacionais e validar se o desempenho do sistema é compatível com os requisitos de aplicações de monitoramento em tempo real. Para todos os efeitos, neste trabalho, considera-se que a execução em tempo real ocorre quando o processamento de cada janela de dados é concluído antes que a próxima janela esteja pronta para ser analisada.

3.6.1 Instrumentação do Código e Coleta de Dados

A medição de tempo foi implementada diretamente no *firmware* em linguagem C, utilizando um dos *timers* de *hardware* do microcontrolador STM32 (TIM2). O *timer* de *hardware* foi configurado para realizar a contagem em microssegundos, garantindo uma medição de alta precisão e de baixo custo de processamento, sem interferir significativamente no desempenho do próprio sistema que está sendo medido.

Para facilitar a ativação e a desativação das medições, foi utilizada compilação condicional com

a diretiva de pré-processador `#ifdef ENABLE_TIME_MEASUREMENT` no arquivo `main.c`. Essa abordagem permite gerar uma versão do *firmware* focada em análise de desempenho e outra, otimizada para a funcionalidade final (extração da *features*, sem o código de medição).

A instrumentação foi realizada por meio de macros em C (`TIME_START` e `TIME_STOP`), que capturam o valor do contador do *timer* no início e no fim de blocos de código específicos. A diferença entre esses valores representa a latência da etapa em questão. Um exemplo simplificado da aplicação dessas macros pode ser visto na Figura 20:

```
/*-------------------------------------------------------*/
/* Extracao das Features no Dominio da Frequencia -----*/
/*-------------------------------------------------------*/

// Calcula fft usando a biblioteca da ARM
TIME_START(startTick);
arm_rfft_fast_f32(&fftHandler, &dataFiltered[0], &outputSignal[0], 0);
TIME_STOP(4, startTick); // Mede o tempo para calcular a fft


TIME_START(startTick);
extractFrequencyDomainFeatures(&fdFeatures, outputSignal, OUTPUT_SIGNAL_SIZE,
    sampleRate);
TIME_STOP(5, startTick); // Mede o tempo para extrair as features do dominio da
    Frequencias

```

Figura 20: Ilustração da utilização dos macros `TIME_START` e `TIME_STOP` para medição da latência das diferentes etapas do *pipeline*. Nesse exemplo, demonstra a utilização dos macros para a medição dos tempo gasto no cálculo da FFT e extração das *features* do domínio da frequência.

3.6.2 Etapas Mensuradas

O *pipeline* de processamento foi segmentado para permitir a medição individual das seguintes etapas, para cada janela de dados processada:

1. **Leitura do Cartão SD:** Tempo necessário para ler um bloco de dados de áudio do cartão MicroSD via interface SPI.
2. **Pré-processamento:** Latência combinada da aplicação do filtro digital passa-faixa e da janela de Hann sobre os dados brutos.
3. **Extração de Features (Domínio do Tempo):** Custo computacional para calcular as 10 métricas no domínio do tempo.
4. **Cálculo da FFT:** Tempo para executar a Transformada Rápida de Fourier sobre a janela de dados.
5. **Extração de Features (Domínio da Frequência):** Custo para identificar os picos e suas respectivas localizações no espectro.
6. **Execução da Inferência:** Latência da execução da função `model_predict()`, que foi gerada pela biblioteca *emlearn*, para classificar a janela de dados.
7. **Ciclo Total de Processamento:** Tempo total gasto desde a leitura do bloco de dados até a obtenção do resultado final da classificação para uma janela.

3.6.3 Armazenamento e Análise dos Resultados

O *firmware* foi desenvolvido com dois modos de operação distintos, controlados por compilação condicional. No modo padrão, o sistema executa o *pipeline* e salva a predição bruta de cada janela

de análise em um arquivo CSV. No modo de análise de desempenho, o sistema gera um arquivo CSV contendo exclusivamente a latência de cada etapa do *pipeline*. Esse procedimento de salvar os dados brutos no cartão MicroSD permite uma análise *offline*, utilizando ferramentas como a biblioteca Pandas em Python para calcular estatísticas e gerar os gráficos discutidos na Seção 4.

Diferentemente do trabalho de referência [9], que aplica uma Função de Distribuição Cumulativa Empírica (**ECDF**, do inglês *Empirical Cumulative Distribution Function*) diretamente no *firmware*, neste trabalho optou-se por mover essa etapa de pós-processamento para um *script* em Python executado no *desktop*. Essa decisão de projeto teve como objetivo aumentar a flexibilidade da análise, permitindo testar diferentes estratégias de estabilização sem a necessidade de recompilar o *firmware* a cada teste.

Esta decisão de projeto foi tomada para aumentar a flexibilidade da análise. Ao separar a geração de dados (no embarcado) da sua interpretação (no *desktop*), isso permite testar e comparar o impacto de diferentes estratégias de pós-processamento de forma mais ágil, sem a necessidade de recompilar o *firmware* a cada teste. Para manter a maior comparabilidade possível com o estudo de referência, optou-se por uma técnica de estabilização comparável: uma janela de votação majoritária (cálculo da moda) com o tamanho de 20 amostras consecutivas, conforme implementado na função `apply_ecdf` do *notebook* de análise `DataAnalyzer.ipynb`.

4 Resultados e Discussão

Esta seção apresenta os resultados experimentais obtidos, divididos em três análises principais:

1. O desempenho dos classificadores no ambiente *desktop* em comparação com o trabalho de referência;
2. O desempenho dos classificadores no ambiente da solução embarcada com o foco na acurácia final do sistema;
3. A análise do desempenho de latência da solução embarcada, avaliando sua viabilidade para aplicações em tempo real.

4.1 Desempenho dos Classificadores no Ambiente Desktop

A primeira etapa da validação consistiu em replicar os experimentos do estudo de referência utilizando o *pipeline* de código aberto baseado em Python e Scikit-learn. A Tabela 3 compara a acurácia média obtida com 5-fold *cross-validation* com os resultados reportados por Shubita et al. [9].

Tabela 3: Comparativo de acurácia média (cross-validation) entre os modelos implementados e os do estudo de referência.

Classificador	**Acurácia (Este Trabalho)**	**Acurácia (Shubita et al., 2023)**
Bagged Trees Ensemble	97,48%	97,4%
Random Forest	97,31%	–
Decision Tree	97,28%	96,8%
Quadratic SVM	96,81%	97,2%
K-Nearest Neighbors (KNN)	92,37%	93,5%
Gaussian Naive Bayes	89,31%	94,0%

Os resultados indicam que o *pipeline* baseado nas ferramentas de código aberto em Python foi capaz de atingir um desempenho de classificação comparável e, em alguns casos, marginalmente superior ao da abordagem com ferramentas proprietárias, validando a premissa inicial do trabalho.

4.1.1 Análise da Qualidade da Conversão com *Emlearn*

A Tabela 4 detalha a acurácia dos modelos antes e depois da conversão para código C com a biblioteca *Emlearn*, utilizando uma divisão fixa de treino e teste sendo executado no *desktop*.

Tabela 4: Comparativo de acurácia entre os modelos em Python e suas versões convertidas em C.

Classificador	**Acurácia (Python)**	**Acurácia (C,Emlearn)**
Decision Tree	98,27%	98,27%
Random Forest	97,86%	97,86%
Extra Trees	98,27%	98,27%
K-Nearest Neighbors	93,17%	93,14%
Gaussian Naïve Bayes	89,34%	79,52%

A conversão demonstrou não gerar grandes perdas para os modelos baseados em árvore e KNN. Porem, é visível a queda de desempenho no modelo *Gaussian Naïve Bayes*, o que sugere que suas hipóteses fundamentais — em especial a suposição de independência condicional entre as variáveis de entrada — não são plenamente válidas para o conjunto de características utilizado. Essa limitação estrutural pode explicar a maior sensibilidade do modelo e a consequente degradação de desempenho após a conversão.

4.2 Análise de Desempenho da Solução Embarcada

A segunda fase da análise avaliou o desempenho com base nos dados gerados pela execução dos modelos de classificação, convertidos para código C pela ferramenta `Emlearn`[11], diretamente na placa STM32 Nucleo-F446RE. O objetivo é era verificar a qualidade da conversão e a performance do sistema na plataforma embarcada que possui recursos computacionais limitados. Esta seção apresenta a acurácia global do sistema e também uma análise por meio das matrizes de confusão.

4.2.1 Análise da Acurácia Global com Pós-Processamento

Para avaliar o impacto do pós-processamento, a acurácia do sistema foi calculada em duas condições: utilizando as predições brutas (saída direta do modelo para cada janela de áudio) e utilizando as predições após a decisão obtida pela técnica de votação majoritária (ECDF). A Tabela 5 consolida a acurácia geral para cada modelo.

Tabela 5: Acurácia geral dos modelos na plataforma embarcada, com e sem a aplicação da decisão baseada na ECDF.

Modelo	**Acurácia Bruta**	**Acurácia baseada na ECDF**	**Melhora**
DecisionTree	96.00%	+99.37%	+3.37 p.p.
GaussianNaiveBayes	78.63%	+91.47%	+12.84 p.p.
RandomForest	92.30%	+99.30%	+7.00 p.p.

A aplicação da decisão baseada na ECDF (com janela de valor igual a 20) resultou em um aumento significativo da acurácia para todos os modelos, conforme os dados da Tabela 5. O destaque é o modelo *Gaussian Naïve Bayes*, que, apesar de apresentar a menor acurácia bruta (78,63%), obteve o maior ganho com a filtragem, um aumento de 12.84 pontos percentuais, indo para um valor de 91.47% e assim ficando mais próximo dos resultados encontrados no trabalho de referência (94,0%) [9]. Os modelos baseados em árvores, *Decision Tree* e *Random Forest*, alcançaram uma ótima performance, com acurácia superior a 99% após o pós-processamento.

Para ajudar no entendimento do impacto causando pela aplicação da ECDF, foram geradas as matrizes de confusão para cada modelo antes e depois do pós-processamento. A Figura 21 apresenta o desempenho bruto do modelo *Decision Tree*, enquanto a Figura 22 mostra o desempenho após implementar a decisão baseada na ECDF. As matrizes de confusão para os modelos *Random Forest* e *Gaussian Naives Bayes* foram colocadas no Anexo B.

A comparação visual entre a Figura 21 e a Figura 22 ilustra o efeito da ECDF. Nas matrizes sem a aplicação da ECDF, é possível observar uma maior quantidade de erros (valores fora da diagonal principal), representando classificações esporádicas e potencialmente incorretas. Após sua aplicação, as matrizes se tornam visivelmente mais "limpas", com os valores concentrando-se quase que exclusivamente na diagonal principal. Isso indica que a técnica de votação majoritária foi eficaz corrigir erros isolados das predições, aumentando a robustez e a acurácia geral do sistema.

Por fim, para permitir uma comparação direta entre as plataformas e com o trabalho de referência, a Tabela 6 apresenta um sumário do desempenho do modelo *Decision Tree*.

Os dados da Tabela 6 confirmam a conclusão central deste trabalho: o pipeline de código aberto não apenas replicou, mas em alguns casos superou ligeiramente, o desempenho da abordagem baseada em ferramentas proprietárias. Um ponto importante a se destacar é que a acurácia final (sem aplicação da filtragem) no sistema embarcado foi quase igual à do estudo original, validando a eficácia da metodologia.

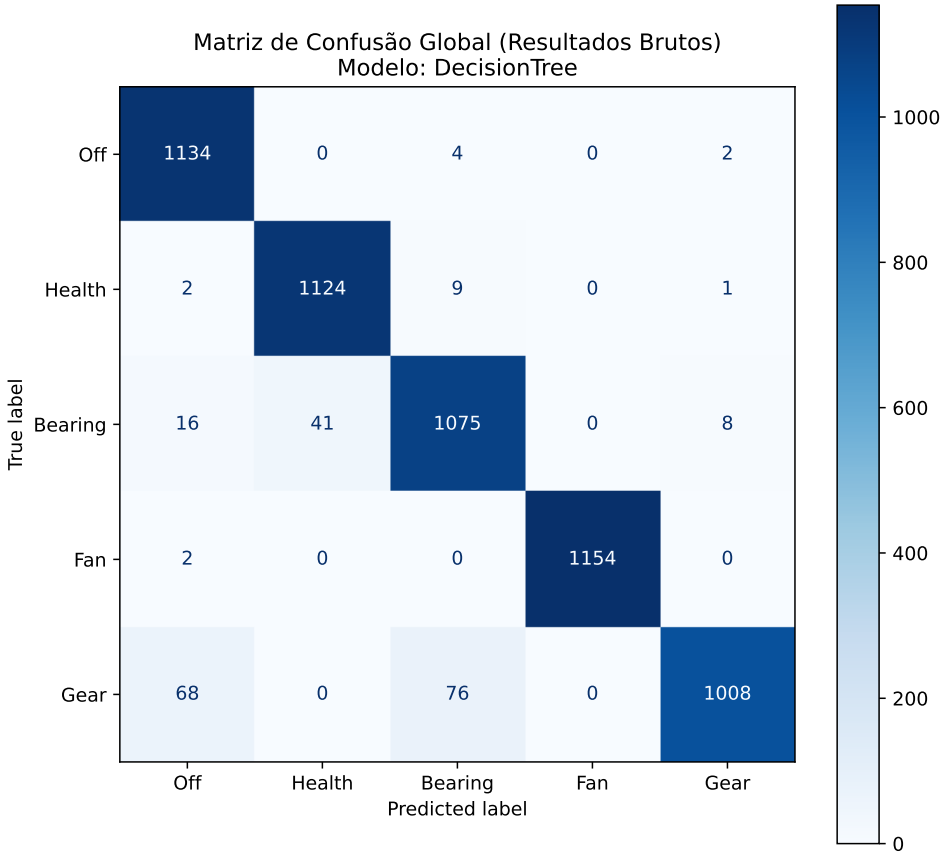

Figura 21: Matriz de confusão para o modelo Decision Tree (Resultados Brutos).

Tabela 6: Comparativo de acurácia do modelo *Decision Tree*.

Ambiente	**Plataforma/Ferramenta**	**Acurácia (%)**
Artigo de Referência [9]	MATLAB (Desktop)	96,8
Este Trabalho	Python / Scikit-learn (Desktop)	97,28
Artigo de Referência [9]	STM32F407VG (Embarcado)	96,1
Este Trabalho	STM32F446RE / Emlearn (Embarcado)	99.37

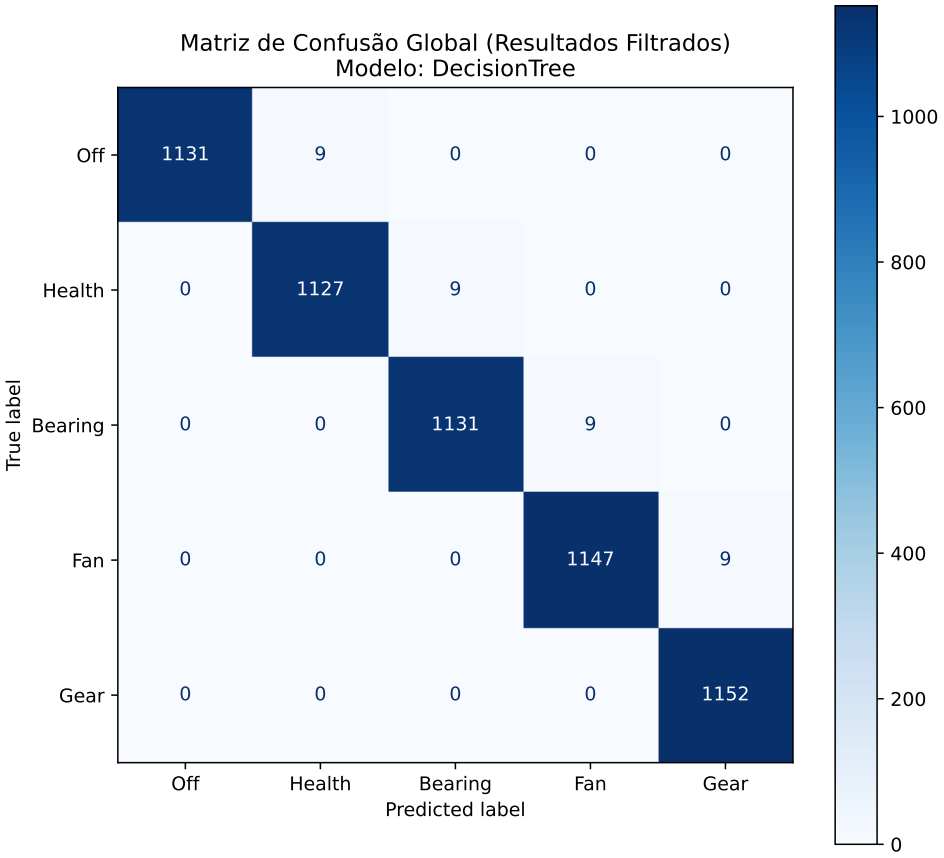

Figura 22: Matriz de confusão para o modelo Decision Tree (Resultados aplicação da ECDF com janela de tamanho igual a 20).

4.3 Análise de Latência e Viabilidade em Tempo Real

A análise de desempenho computacional revelou um ponto de grande importância do projeto. Conforme sumarizado na Tabela 7, nenhum dos modelos testados atendeu ao requisito de tempo real de aproximadamente 42,667 ms (42.667 µs, i.e., o tempo correspondente à aquisição de uma nova janela de dados) por *batch* com o microcontrolador rodando à frequência de *clock* de 84 MHz (frequencência padrão quando o projeto foi criado na IDE STM32CubeIDE).

Tabela 7: Sumário de desempenho de latência no microcontrolador.

Modelo	**Tempo Médio por Batch (µs)**
Decision Tree	341.590,18
Random Forest	341.545,36
Gaussian Naive Bayes	342.268,97

Para investigar a causa deste resultado, a Tabela 8 detalha o tempo gasto de processamento médio nas diferentes etapas do *pipeline*.

Tabela 8: Tempo de processamento médio por etapa do *pipeline* (em µs).

Etapa	**Decision Tree**	**Random Forest**	**Gaussian NB**
TempoNormalizacao	903,64	903,63	903,68
TempoJanela	8.719,00	8.719,00	8.719,00
TempoFeatTempo	12.901,08	12.884,63	12.884,62
TempoFFT	10.147,08	10.143,00	10.143,01
TempoFeatFrequencia	2.208,07	2.199,18	2.199,19
TempoInferencia	**6,95**	**10,22**	**376,91**

A análise dos intervalos de tempo gastos em cada estágio mostra que o tempo gasto na inferência do modelo (`TempoInferencia`) é extremamente curto para os modelos de árvore *(Decision Tree)*, representando uma fração mínima do tempo gasto total. O principal "gargalo" está na etapa de **extração de features**, especificamente no cálculo das *features* no domínio do tempo (`TempoFeatTempo`) e na FFT. Este resultado é importante, pois reforça a ideia de que, para esta aplicação, a otimização do desempenho não deve focar no modelo de ML, mas sim nas etapas de pré-processamento e extração de *features*.

Para complementar essa análise, foi realizado um segundo experimento para verificar o impacto da frequência de operação do processador no tempo de execução do sistema embarcado. O *firmware* foi configurado novamente no modo de medição do tempo e o teste foi refeito na placa STM32 Nucleo-F446RE configurada com de frequência de *clock* máxima - **180 MHz**.

A Tabela 9 apresenta os resultados de tempo de execução médio para o modelo *Decision Tree*, comparando a frequência de operação padrão (84 MHz) e a frequência de operação máxima (180 MHz). O incremento de velocidade (*speedup*) foi calculado como a razão entre o tempo de execução em 84 MHz e em 180 MHz.

O aumento da frequência do *clock* de 84 MHz para 180 MHz representa um ganho teórico de 2.14x. Os resultados práticos mostram um *speedup* médio de 2.14x, indicando um alinhamento entre a teoria e a prática. Isso indica que todas as etapas de processamento do algoritmo são fortemente dependentes da capacidade de processamento da CPU (*CPU-bound*).

A avaliação da viabilidade do sistema para uma aplicação em tempo real exige uma análise mais aprofundada. O tempo total para processar uma única janela de 2048 amostras na configuração de 180 MHz foi de aproximadamente 16.295 µs. No entanto, devido à implementação de um janelamento com 75% de sobreposição, o sistema precisa concluir este processamento no intervalo de tempo em que apenas

Tabela 9: Comparativo de tempo de execução por estágio para diferentes clocks do processador, utilizando o modelo *Decision Tree*.

Estágio do Pipeline	**Tempo (µs) @84 MHz**	**Tempo (µs) @180 MHz**	**Speedup (x)**
Tempo de Normalização	903,64	421,40	2,14
Tempo de Janelamento	8.719,00	4.068,00	2,14
Features de Tempo	12.901,08	6.011,44	2,15
Cálculo da FFT	10.147,08	4.762,58	2,13
Features de Frequência	2.208,07	1.026,31	2,15
Tempo de Inferência	11,20	5,21	2,15
Tempo Total por Janela	**34.890,07**	**16.294,94**	**2,14**

25% de novas amostras (512 amostras) são adquiridas. Com uma frequência de amostragem de 48 kHz, este de tempo necessário é de aproximadamente 10667 µs.

Como o tempo de processamento (16.295 µs) excede o tempo de aquisição de novos dados (10.667 µs), conclui-se que o sistema, na sua configuração atual com sobreposição de janelas, não é capaz de operar em tempo real. Para alcançar essa capacidade, seria necessário reduzir ou eliminar a sobreposição das janelas, ou utilizar um microcontrolador com maior capacidade de processamento. A distribuição dos intervalos de tempo de execução para cada estágio pode ser visualizada nas Figuras 23 e 24.

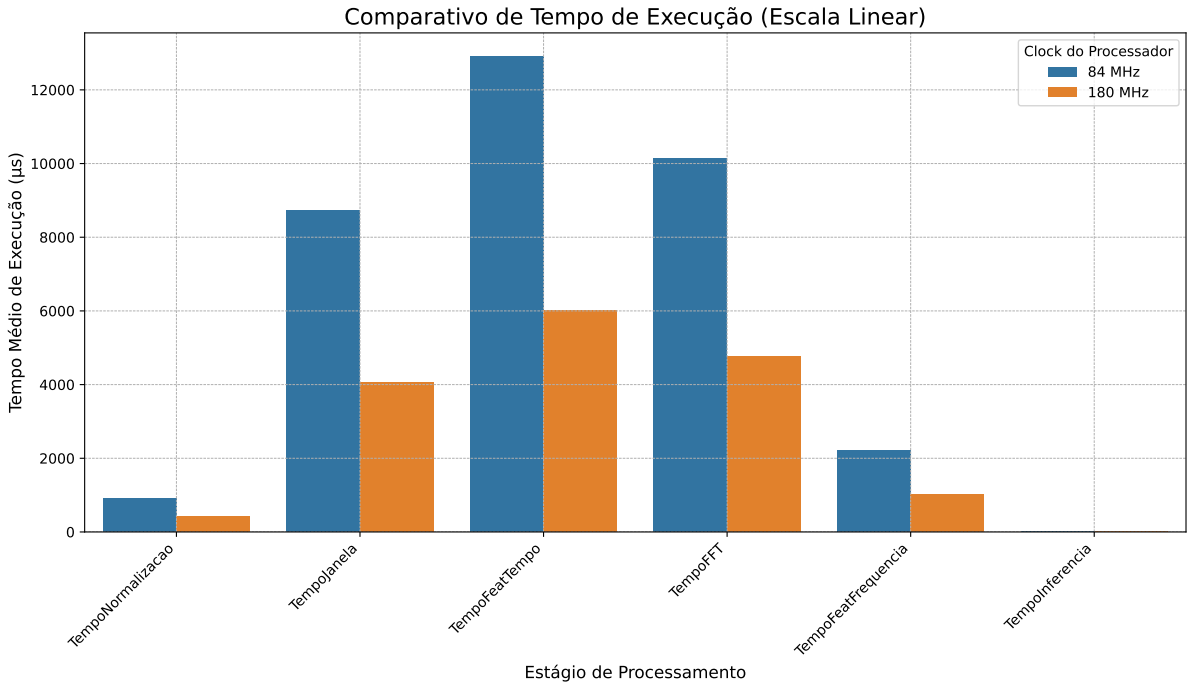

Figura 23: Comparativo gráfico do tempo de execução médio para cada estágio do *pipeline* em 84 MHz e 180 MHz.

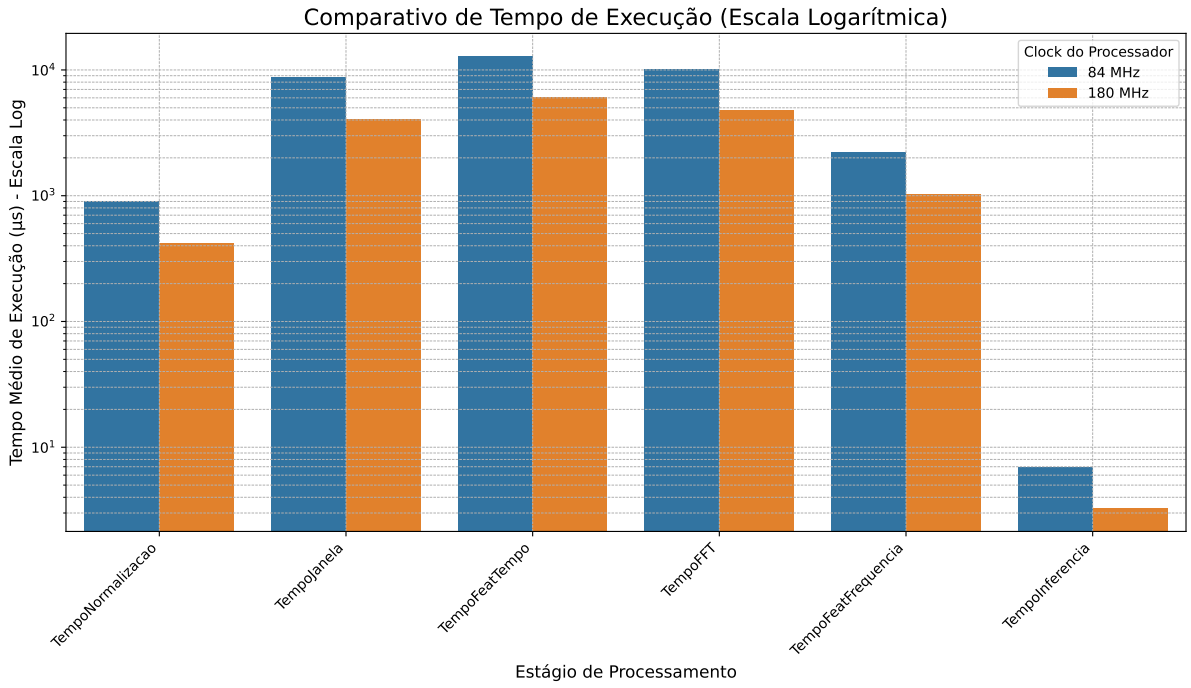

Figura 24: Comparativo gráfico do tempo de execução médio em escala logarítmica para cada estágio do pipeline em 84 MHz e 180 MHz.

5 Conclusão

Este trabalho teve como objetivo principal avaliar a viabilidade de replicar a metodologia de detecção de falhas em máquinas rotativas de Shubita et al. [9], substituindo ferramentas proprietárias (MATLAB) por um *pipeline* baseado em software de código aberto e hardware acessível. Conclui-se então que a abordagem foi bem-sucedida, demonstrando que o ecossistema de código aberto em Python foi capaz de reproduzir os resultados de acurácia do estudo original, validando a hipótese central do trabalho.

Essa conclusão é suportada pelos resultados encontrados. Primeiramente, confirmou-se que o *pipeline* desenvolvido em Python não apenas é possível, mas também atingiu um desempenho de classificação semelhante ao do artigo de referência, com o modelo de *Decision Tree* alcançando 97,28% de acurácia em validação cruzada (Tabela 3). Além disso, um nível semelhante de desempenho também foi observado pelos modelos convertidos executados na plataforma STM32 Nucleo-F446RE, alcançando uma acurácia final de 96,00% (sem aplicação da ECDF) e cerca de 99.37% com a aplicação da ECDF para o modelo *Decision Tree* (Tabela 5). Apresentando assim, valores de acurácia semelhantes à reportada no estudo original, validando a eficácia de todo o fluxo de trabalho, desde o treinamento até a inferência no sistema embarcado.

Além da acurácia, a análise de desempenho computacional, detalhada na Tabela 9, trouxe uma visão real sobre as capacidades do sistema. Um fato de grande importância foi a constatação de que a sobreposição de 75% entre as janelas impede a operação em tempo real na plataforma testada, mostrando um importante compromisso de projeto (*trade-off*) entre a densidade da análise e a latência do sistema.

A principal contribuição deste trabalho é a demonstração da viabilidade da biblioteca *Emlearn* [11] dentro de um fluxo de desenvolvimento para *Edge ML* totalmente baseado em software de código aberto, validando sua aplicação prática em cenários com restrições de hardware.

Por último, é importante reconhecer as limitações deste trabalho. A validação foi realizada utilizando o mesmo conjunto de dados do artigo de referência, porém, a implementação focou na validação da análise a partir de dados pré-gravados, não englobando a integração de um microfone para aquisição de dados em tempo real.

Por conta dessas limitações, uma oportunidade para trabalho futuro mais imediata seria a integração de um microfone ao sistema para a aquisição dos dados utilizando os conversores A/D presentes na plataforma e o processamento de dados em tempo real, validando a solução de ponta a ponta. Outra opção seria a otimização do código embarcado ou a avaliação de um microcontrolador mais potente para viabilizar a análise com alta sobreposição em tempo real.

6 Referências

[1] Hosameldin Ahmed e Asoke K Nandi. *Condition monitoring with vibration signals*. en. IEEE Press. New York, NY: Wiley-IEEE Press, jan. de 2020.

[2] Swapnil K. Gundewar e Prasad V. Kane. "Rolling element bearing fault diagnosis using supervised learning methods- artificial neural network and discriminant classifier". Em: *International Journal of System Assurance Engineering and Management* 13.6 (ago. de 2022), pp. 2876–2894. ISSN: 0976-4348. DOI: 10.1007/s13198-022-01757-4. URL: http://dx.doi.org/10.1007/s13198-022-01757-4.

[3] R. K. Mobley. *An Introduction to Predictive Maintenance*. Butterworth-Heinemann, 2002.

[4] K. Vernekar, H. Kumar e G. K. V. "Engine gearbox fault diagnosis using machine learning approach". Em: *Journal of Quality in Maintenance Engineering* 24 (2018), pp. 345–357.

[5] J. Taylor. *Machinery Oil Analysis: Methods, Automation & Benefits*. Industrial Press, 2014.

[6] T. Holroyd e N. Randall. "Use of acoustic emission for machine condition monitoring". Em: *British Journal of Non-Destructive Testing* 35 (1993), pp. 75–78.

[7] *MP34DT05-MEMS Microphone Datasheet*. InvenSense (TDK). 2019. URL: https://invensense.tdk.com/products/ics/mp34dt05/.

[8] T. Tran e J. Lundgren. "Drill fault diagnosis based on the scalogram and MEL spectrogram of sound signals using artificial intelligence". Em: *IEEE Access* 8 (2020), pp. 203655–203666.

[9] R. R. Shubita, A. S. Alsadeh e I. M. Khater. "Fault Detection in Rotating Machinery Based on Sound Signal Using Edge Machine Learning". Em: *IEEE Access* 11 (2023), pp. 6665–6671.

[10] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". Em: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[11] Jon Nordby, Mark Cooke e Adam Horvath. *emlearn: Machine Learning inference engine for Microcontrollers and Embedded Devices*. Mar. de 2019. DOI: 10.5281/zenodo.2589394. URL: https://doi.org/10.5281/zenodo.2589394.

[12] R. A. Collacott. *Mechanical Fault Diagnosis and Condition Monitoring*. Springer, 2012.

[13] Salah Al-Obaidi. "A Review of Acoustic Emission Technique for Machinery Condition Monitoring: Defects Detection & Diagnostic". Em: *Applied Mechanics and Materials* (jan. de 2012).

[14] M. et al. Altaf. "Automatic and efficient fault detection in rotating machinery using sound signals". Em: *Acoustics Australia* 47 (2019), pp. 125–139.

[15] Alan V. Oppenheim e Ronald W. Schafer. *Discrete-Time Signal Processing*. 3rd. Upper Saddle River, NJ: Prentice Hall, 2010. ISBN: 978-0131988422.

[16] Robert Bond Randall. *Vibration-based condition monitoring*. en. 2ª ed. Standards Information Network, jun. de 2021.

[17] Wahyu Caesarendra e Tegoeh Tjahjowidodo. "A Review of Feature Extraction Methods in Vibration-Based Condition Monitoring and Its Application for Degradation Trend Estimation of Low-Speed Slew Bearing". Em: *Machines* 5.4 (2017). ISSN: 2075-1702. DOI: 10.3390/machines5040021. URL: https://www.mdpi.com/2075-1702/5/4/21.

[18] MathWorks. *Signal Features*. The MathWorks, Inc. 2025. URL: https://www.mathworks.com/help/predmaint/ug/signal-features.html (acesso em 28/07/2025).

[19] W. O. Bussab e P. A. Morettin. *Estatística Básica*. 9ª ed. São Paulo: Saraiva, 2017. ISBN: 978-8547220228.

[20] "IEEE Standard for Transitions, Pulses, and Related Waveforms". Em: *IEEE Std 181-2011 (Revision of IEEE Std 181-2003)* (2011), pp. 1–71. DOI: 10.1109/IEEESTD.2011.6016198.

[21] James W. Cooley e John W. Tukey. “An algorithm for the machine calculation of complex Fourier series”. Em: *Mathematics of Computation* 19.90 (1965), pp. 297–301. DOI: 10.1090/S0025-5718-1965-0178586-1.

[22] NTI Audio. *Transformação Rápida de Fourier (FFT)*. Disponível em: https://www.nti-audio.com/pt/suporte/saber-como/transformacao-rapida-de-fourier-fft. Acesso em: 28 jul. 2025. 2023.

[23] Trevor Hastie. *The elements of statistical learning : data mining, inference, and prediction*. New York: Springer, 2009. ISBN: 978-0387848570.

[24] Kevin Murphy. *Machine learning : a probabilistic perspective*. Cambridge, Mass: MIT Press, 2012. ISBN: 9780262018029.

[25] Vinicius Nala. *Modelos ML Paramétricos x Não-paramétricos*. pt-br. 2024. URL: https://medium.com/@viniciusnala/modelos-ml-param%C3%A9tricos-x-n%C3%A3o-param%C3%A9tricos-0cc68e1a82aa (acesso em 30/07/2025).

[26] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. New York, NY: Springer, 2006. ISBN: 978-0-387-31073-2.

[27] Jason Brownlee. *Parametric and Nonparametric Machine Learning Algorithms*. Acesso em: 30 jul. 2025. [Em inglês]. 2020. URL: https://machinelearningmastery.com/parametric-and-nonparametric-machine-learning-algorithms/ (acesso em 30/07/2025).

[28] Gareth James et al. *An Introduction to Statistical Learning: with Applications in R*. New York: Springer, 2013. ISBN: 978-1-4614-7137-0.

[29] Leo Breiman et al. *Classification and Regression Trees*. Monterey, CA: Wadsworth & Brooks/Cole Advanced Books & Software, 1984. ISBN: 978-0-412-04841-8.

[30] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media, Inc, 2019. ISBN: 9781492032649.

[31] Andreas C. Müller e Sarah Guido. *Introduction to Machine Learning with Python: A Guide for Data Scientists*. Sebastopol, CA: O'Reilly Media, 2016. ISBN: 978-1449369415.

[32] Stuart Russell e Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th. Pearson, 2020. ISBN: 9781292401133.

[33] Fernando J. VON ZUBEN e Romis R. de F. ATTUX. “Uma introdução às support vector machines”. Português. Em: *Revista de Informática Teórica e Aplicada* 9.2 (2002). Acesso em: 30 jul. 2025, pp. 43–67. DOI: 10.22456/2175-2745.5690. URL: https://seer.ufrgs.br/index.php/rita/article/view/5690.

[34] Oliver Theobald. *Machine Learning for Absolute Beginners: A Plain English Introduction*. Inglês. Third Edition. Independently Published, 2020, p. 180. ISBN: 978-1913666521. URL: https://www.amazon.com/dp/B08S2K5P2H.

[35] Massimo Merenda, Carlo Porcaro e Demetrio Iero. “Edge Machine Learning for AI-Enabled IoT Devices: A Review”. Em: *Sensors* 20.9 (2020). ISSN: 1424-8220. DOI: 10.3390/s20092533. URL: https://www.mdpi.com/1424-8220/20/9/2533.

[36] Pete Warden e Daniel Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. Sebastopol, CA: O'Reilly Media, 2020. ISBN: 978-1492052043.

[37] *STM32 Nucleo-F446RE User Manual*. STMicroelectronics. 2023.

[38] Robert David et al. “TensorFlow Lite for Microcontrollers: A Case Study of an Open-Source, Production-Ready Microcontroller-Targeted ML Framework”. Em: *Proceedings of the 4th MLSys Conference, (MLSys '21)*. MLSys.org, 2021. URL: https://proceedings.mlsys.org/paper/2021/hash/31ca8355611433c7b88496152a10c90c-Abstract.html.

[39] Tianqi Chen et al. “µTVM: A Compiler-Based Framework for Neural Network Inference on Bare-Metal Microcontrollers”. Em: *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 881–897. ISBN: 9781450371025. DOI: 10.1145/3373376.3378512. URL: https://doi.org/10.1145/3373376.3378512.

[40] Ítalo Milhomem de Abreu Lanza. *Pipeline de Código Aberto para Detecção de Falhas em Máquinas Rotativas*. Código-fonte desenvolvido para o Trabalho de Graduação. 2025. URL: https://github.com/italolanza/projeto-tg/.

[41] Microsoft. *Visual Studio Code*. Versão 1.102.3. Acessado em: 02 Ago. 2025. 2025. URL: https://code.visualstudio.com/.

[42] STMicroelectronics. *STM32CubeIDE*. Versão 1.18.1. Acessado em: 02 Ago. 2025. 2025. URL: https://www.st.com/en/development-tools/stm32cubeide.html.

[43] Rashad Shubita. *Fault Detection using TinyML*. Inglês. Commit: d7c2f1b, License: MIT. 2023. URL: https://github.com/RashadShubita/Fault-Detection-using-TinyML (acesso em 02/08/2025).

[44] Wikipedia contributors. *WAV. Waveform Audio File Format*. Português. Última modificação: 16 de março de 2024. Wikimedia Foundation. 2024. URL: https://pt.wikipedia.org/wiki/WAV (acesso em 02/08/2025).

[45] Steven W. Smith. *The Scientist and Engineer’s Guide to Digital Signal Processing*. Second. San Diego, California: California Technical Publishing, 1999. ISBN: 0-9660176-3-3. URL: http://www.dspguide.com/.

[46] Allen B. Downey. *Think DSP: Digital Signal Processing in Python*. 1st. Sebastopol, CA: O’Reilly Media, 2016. ISBN: 978-1491938454. URL: https://greenteapress.com/thinkdsp/.

[47] Steven Barrett. *Microcontrollers fundamentals for engineers and scientists*. San Rafael, Calif: Morgan & Claypool Publishers, 2006. ISBN: 1598290584.

[48] ARM Limited. *ARM CMSIS: Cortex Microcontroller Software Interface Standard*. Versão 6.0.0. Acesso em: 17 jul. 2025. ARM Ltd. Cambridge, United Kingdom, mai. de 2023. URL: https://developer.arm.com/Architectures/CMSIS.

A Bibliotecas utilizadas no Trabalho

A.1 Bibliotecas Python

Veja na tabela abaixo as bibliotecas Python utilizadas no artigo e suas respectivas versões:

Tabela 10: Bibliotecas Python e suas versões.

Biblioteca	Versão
Jupyter	`1.1.1`
Pandas	`2.2.3`
SoundFile	`0.12.1`
Scikit-learn	`1.5.2`
NumPy	`2.1.2`
Notebook	`7.2.2`
IPykernel	`6.29.5`
LLVMLite	`0.43.0`
Matplotlib	`3.9.2`
Emlearn	`0.20.4`
Seaborn	`0.13.2`

A.2 Bibliotecas C

Tabela 11: Bibliotecas utilizadas no sistema embarcado e suas versões.

Biblioteca	Versão
FatFS	`R0.12c`
CMSIS-DSP	`1.10.0`
STM32F4xx HAL Driver	`1.8.4`

B Matrizes de Confusão Adicionais

Este apêndice contém as matrizes de confusão para os modelos *Random Forest* e *Gaussian Naive Bayes*, complementando a análise de resultados apresentada no Seção 4.

B.1 Resultados do Modelo Random Forest

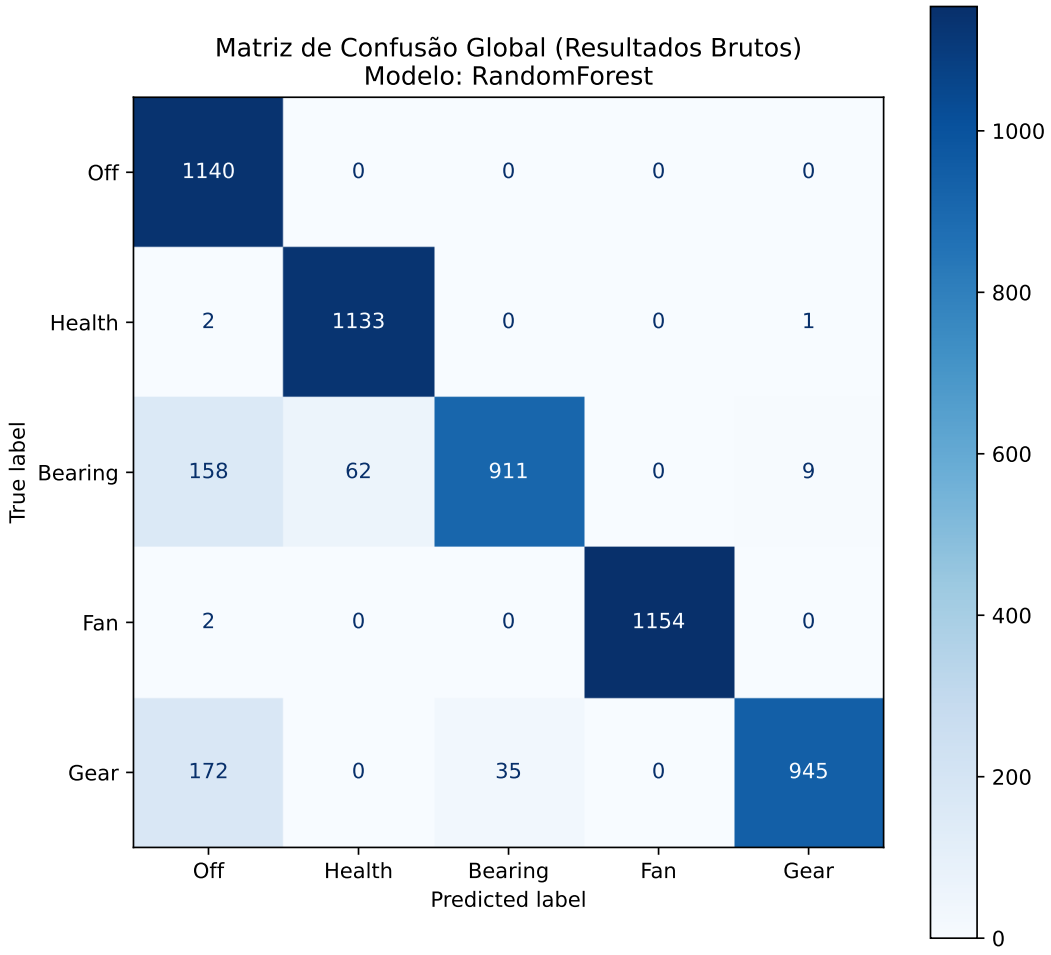

Figura 25: Matriz de confusão para o modelo Random Forest (Resultados Brutos).

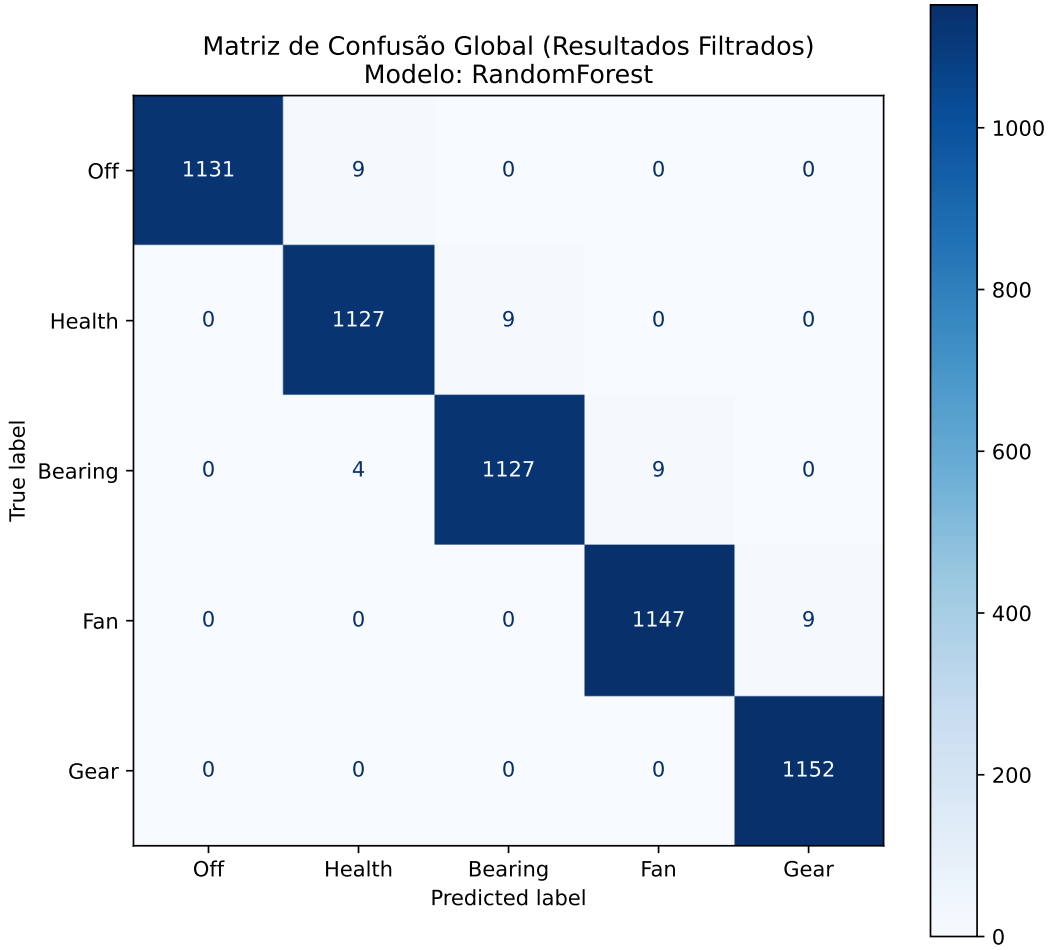

Figura 26: Matriz de confusão para o modelo Random Forest (Resultados Pós-Filtro ECDF com janela de tamanho igual a 20).

B.2 Resultados do Modelo *Gaussian Naive Bayes*

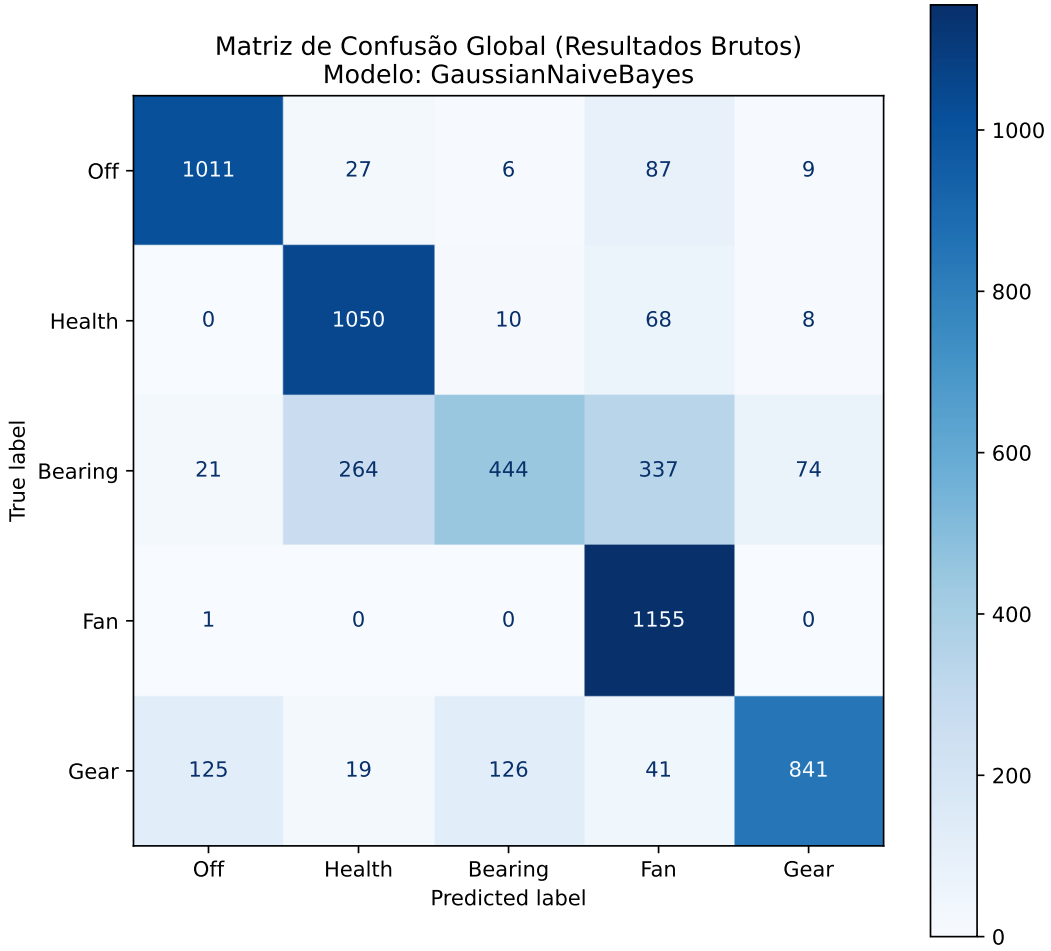

Figura 27: Matriz de confusão para o modelo Naive Bayes (Resultados Brutos).

.

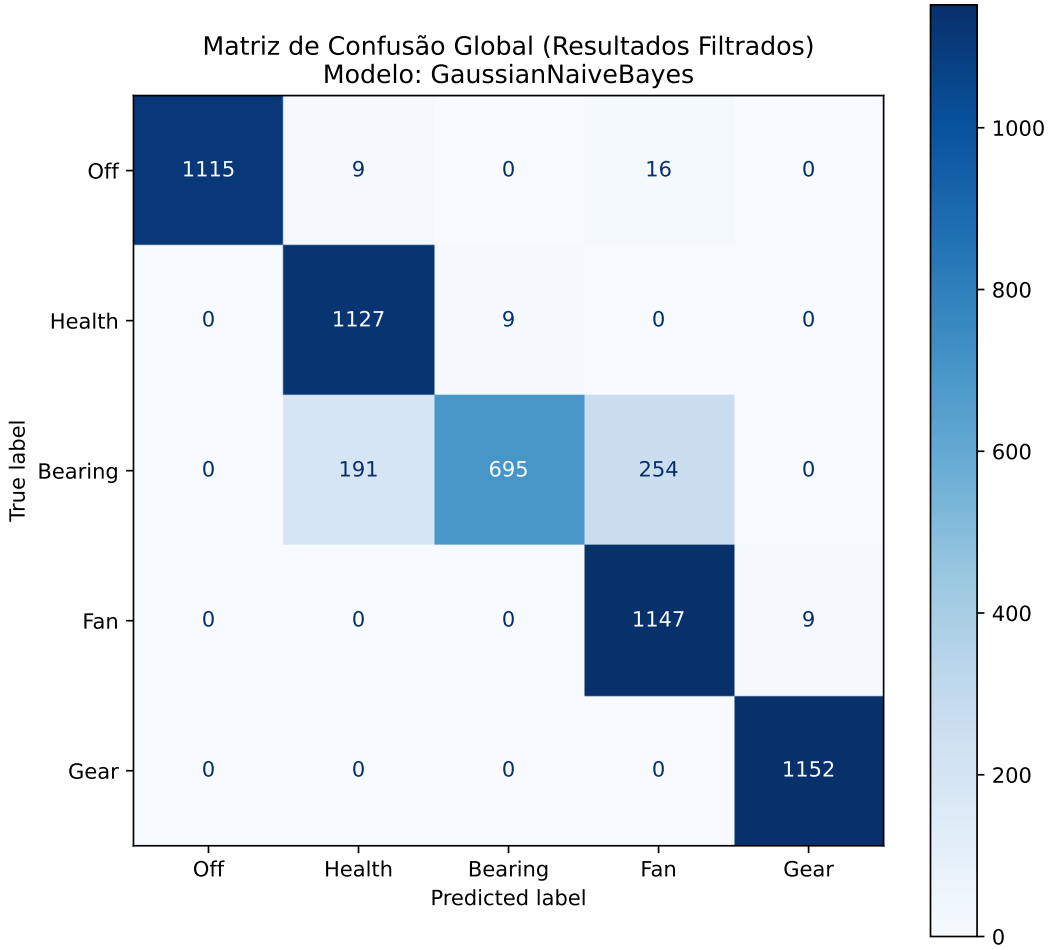

Figura 28: Matriz de confusão para o modelo Naive Bayes (Resultados Pós-Filtro ECDF com janela de tamanho igual a 20).