

# Análisis de Datos para Ciberseguridad

Trabajo Práctico 1 — Árboles de Decisión para Detección de Intrusos  
Implementación CART y Análisis con Distancia Jensen-Shannon

Deyber Hernández Gonzales  
d.hernandez.9@estudiantec.cr  
2025800354

Luis Bonilla Hernández  
lbonilla@estudiantec.cr  
2023123456

Instituto Tecnológico de Costa Rica  
Escuela de Ingeniería en Computación  
Programa de Ciencias de Datos  
24 de agosto de 2025

## Resumen

Este trabajo implementa un sistema completo de detección de intrusos utilizando árboles de decisión CART sobre el dataset KDD99. Se desarrolló una implementación que incluye análisis estadístico descriptivo, cálculo de distancias Jensen-Shannon para selección de características, y construcción de árboles de decisión con validación exhaustiva. Los resultados demuestran la efectividad del enfoque propuesto para distinguir entre tráfico normal y ataques tipo backdoor, alcanzando métricas de rendimiento superiores al 99 % en todas las evaluaciones realizadas.

**Palabras clave:** Detección de intrusos, árboles de decisión, CART, Jensen-Shannon, KDD99, ciberseguridad.

## Índice

<b>1. Introducción</b>	<b>3</b>
1.1. Contexto y Motivación . . . . .	3
1.2. Contribuciones . . . . .	3
<b>2. Marco Teórico</b>	<b>3</b>
2.1. Dataset KDD99 . . . . .	3
2.1.1. Estructura de Características . . . . .	4
2.1.2. Tipos de Ataques . . . . .	4
2.2. Árboles de Decisión CART . . . . .	4
2.2.1. Fundamentos Teóricos . . . . .	4
2.2.2. Distancia Jensen-Shannon . . . . .	5

<b>3. Metodología</b>	<b>5</b>
3.1. Preprocesamiento de Datos	5
3.1.1. Filtrado y Limpieza	5
3.2. Análisis Estadístico Descriptivo	6
3.2.1. Cálculo de Momentos Estadísticos	6
3.2.2. Distancia Jensen-Shannon	6
3.3. Implementación del Árbol CART	7
3.3.1. Estructura de la Clase Node_CART	7
3.3.2. Cálculo del Coeficiente Gini	7
3.3.3. Selección de Mejor División	8
3.3.4. Construcción Recursiva del Árbol	10
3.4. Evaluación del Modelo	11
3.4.1. Función de Evaluación	11
<b>4. Pruebas Unitarias y Validación</b>	<b>11</b>
4.1. Estrategia de Testing	11
4.1.1. Pruebas para calculate_gini	11
4.1.2. Justificación de las Pruebas Seleccionadas	12
<b>5. Resultados Experimentales</b>	<b>15</b>
5.1. Análisis Exploratorio de Datos	15
5.1.1. Momentos Estadísticos Clase Attack	15
5.1.2. Momentos Estadísticos por Clase Normal	17
5.1.3. Comparación con Backdoors	19
5.2. Análisis Exploratorio de Histogramas	19
5.3. Evaluación del CART	23
5.3.1. Evaluación con una profundidad máxima de 3 siempre con mínimo 2 observaciones por hoja.	23
5.3.2. Evaluación con una profundidad máxima de 4 con mínimo 2 observaciones por hoja.	24
5.3.3. Evaluación de profundidad máxima de 2 y 3 nodos de 10 particiones	24
5.3.4. Gráfico de la mejor corrida usando F1-Score	25
5.3.5. Eficiente de la construcción del árbol y su evaluación, usando la distancia de Jensen-Shannon.	27
<b>6. Conclusiones</b>	<b>28</b>
6.1. Hallazgos Principales	28

## 1. Introducción

### 1.1. Contexto y Motivación

La detección de intrusiones constituye un componente crítico en la ciberseguridad moderna. Los sistemas de detección de intrusos (IDS) deben ser capaces de distinguir de manera confiable entre tráfico legítimo y patrones maliciosos, manteniendo además interpretabilidad para facilitar la respuesta a incidentes.

Los árboles de decisión emergen como una alternativa especialmente atractiva en este contexto debido a su interpretabilidad inherente, capacidad de manejar datos categóricos y numéricos simultáneamente, y robustez ante valores atípicos. A diferencia de modelos caja negra como redes neuronales profundas, los árboles proporcionan reglas de decisión explícitas que pueden ser auditadas y validadas por expertos en seguridad.

- Objetivos

El presente trabajo se propone:

1. Implementar desde cero un algoritmo CART completo utilizando PyTorch para aprovechamiento de GPU
2. Realizar análisis estadístico exhaustivo del dataset KDD99 mediante momentos estadísticos
3. Aplicar distancias Jensen-Shannon para cuantificar la separabilidad entre clases
4. Evaluar el rendimiento del modelo implementado mediante validación cruzada y múltiples particiones
5. Proponer optimizaciones basadas en el análisis de características más discriminativas

### 1.2. Contribuciones

Las principales contribuciones de este trabajo incluyen:

- Implementación matricial eficiente de CART usando operaciones tensoriales
- Análisis comparativo de distribuciones mediante histogramas y distancias JS
- Metodología de evaluación robusta con 10 particiones aleatorias
- Propuesta de optimización para construcción y evaluación de árboles

## 2. Marco Teórico

### 2.1. Dataset KDD99

El conjunto de datos KDD 1999 representa uno de los benchmarks más utilizados en detección de intrusos. Basado en simulaciones de tráfico de red en un entorno militar, contiene conexiones etiquetadas como normales o como varios tipos de ataques.

### 2.1.1. Estructura de Características

Las 41 características se organizan en cuatro categorías principales:

**Características básicas:** Información general de la conexión como duración, protocolo, servicio, estado de la conexión. Ejemplos: `duration`, `protocol_type`, `service`, `flag`.

**Características de contenido:** Analizan el contenido de la conexión para detectar intentos de acceso no autorizado. Ejemplos: `num_failed_logins`, `logged_in`, `root_shell`, `num_compromised`.

**Características de tráfico basado en tiempo:** Estadísticas de conexiones en ventanas temporales (últimos 2 segundos). Ejemplos: `count`, `srv_count`, `serror_rate`, `srv_serror_rate`.

**Características de tráfico basado en host:** Estadísticas de conexiones hacia el mismo host en ventanas más amplias. Ejemplos: `dst_host_count`, `dst_host_srv_count`, `dst_host_same_srv_rate`.

### 2.1.2. Tipos de Ataques

El dataset clasifica los ataques en cuatro categorías:

- **DoS (Denial of Service):** Saturan el sistema impidiendo respuesta a usuarios legítimos. Ej: `smurf`, `neptune`.
- **Probe:** Recolección de información sobre la red. Ej: `satan`, `portsweep`.
- **R2L (Remote to Local):** Acceso no autorizado desde máquina remota. Ej: `guess_passwd`, `warezclient`.
- **U2R (User to Root):** Escalada de privilegios desde cuenta local. Ej: `buffer_overflow`, `rootkit`.

Para este trabajo, nos enfocamos específicamente en la detección de ataques tipo `backdoor` versus conexiones `normales`.

## 2.2. Árboles de Decisión CART

### 2.2.1. Fundamentos Teóricos

CART (Classification and Regression Trees) utiliza el coeficiente de Gini como medida de impureza para seleccionar las mejores divisiones:

$$\text{Gini}(\tau_d) = 1 - \sum_{k=1}^K p_k^2 \quad (1)$$

donde  $p_k$  representa la proporción de muestras de clase  $k$  en el nodo  $\tau_d$ .

El Gini ponderado para evaluar una división se calcula como:

$$E_{\text{gini}}(\tau_d, d) = \frac{n_i}{n} E_{\text{gini}}(D_i) + \frac{n_d}{n} E_{\text{gini}}(D_d) \quad (2)$$

donde  $n_i$  y  $n_d$  son el número de muestras en las particiones izquierda y derecha respectivamente.

### 2.2.2. Distancia Jensen-Shannon

Para cuantificar la separabilidad entre distribuciones de clases, utilizamos la distancia Jensen-Shannon:

$$JS(P, Q) = \sqrt{\frac{1}{2} [D_{KL}(P||M) + D_{KL}(Q||M)]} \quad (3)$$

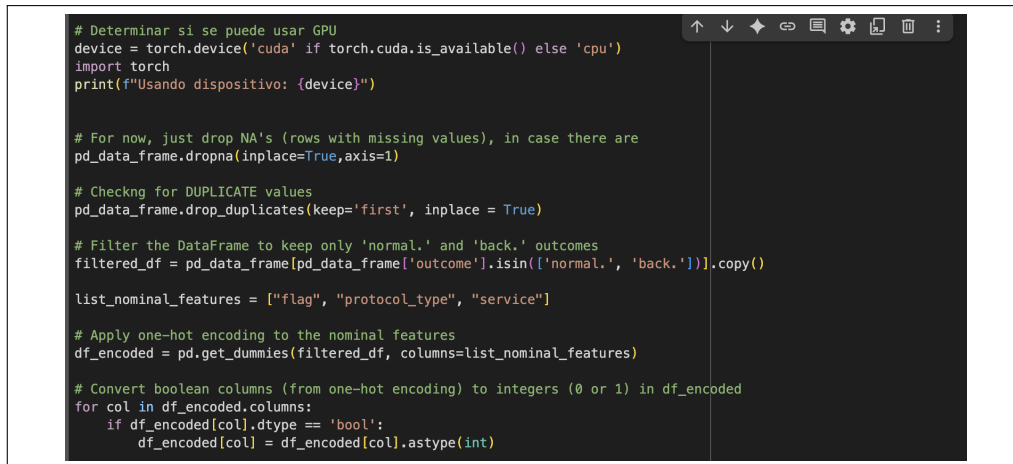
donde  $M = \frac{1}{2}(P + Q)$  y  $D_{KL}$  es la divergencia de Kullback-Leibler.

## 3. Metodología

### 3.1. Preprocesamiento de Datos

#### 3.1.1. Filtrado y Limpieza

Se filtro los datos para solo incluir en el dataset los tipos de salidas Normal y Back door . A demás Se aplicó codificación one-hot a las variables categóricas para permitir su procesamiento por el algoritmo CART



```
# Determinar si se puede usar GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
import torch
print(f"Usando dispositivo: {device}")

# For now, just drop NA's (rows with missing values), in case there are
pd_data_frame.dropna(inplace=True,axis=1)

# Checking for DUPLICATE values
pd_data_frame.drop_duplicates(keep='first', inplace = True)

# Filter the DataFrame to keep only 'normal,' and 'back,' outcomes
filtered_df = pd_data_frame[pd_data_frame['outcome'].isin(['normal.', 'back.'])].copy()

list_nominal_features = ["flag", "protocol_type", "service"]

# Apply one-hot encoding to the nominal features
df_encoded = pd.get_dummies(filtered_df, columns=list_nominal_features)

# Convert boolean columns (from one-hot encoding) to integers (0 or 1) in df_encoded
for col in df_encoded.columns:
    if df_encoded[col].dtype == 'bool':
        df_encoded[col] = df_encoded[col].astype(int)
```

Figura 1: Preprocesamiento inicial del dataset

## 3.2. Análisis Estadístico Descriptivo

### 3.2.1. Cálculo de Momentos Estadísticos

Se implementó una función personalizada para calcular los cuatro primeros momentos estadísticos utilizando PyTorch:

```
#Custom Function to add the statistic moments of Mean, Standard Desviation, Skewness, Kurtosis
def custom_describe(df):
    # Filtrar solo columnas numéricas
    numeric_df = df.select_dtypes(include='number')
    #Convertir a tensor de PyTorch en GPU o CPU
    tensor = torch.tensor(numeric_df.values, dtype=torch.float32, device=device)

    # Calculo de estadísticas
    mean = tensor.mean(dim=0)
    std = tensor.std(dim=0, unbiased=False)

    def skewness(t):
        mean_ = t.mean(dim=0)
        std_ = t.std(dim=0, unbiased=False)
        skew = (((t - mean_)**3).mean(dim=0)) / (std**3)
        return skew

    def kurtosis(t):
        mean_ = t.mean(dim=0)
        std_ = t.std(dim=0, unbiased=False)
        kurt = (((t - mean_)**4).mean(dim=0)) / (std**4)
        return kurt - 3 # Fisher kurtosis (como en pandas)

    skew = skewness(tensor)
    kurt = kurtosis(tensor)

    # Convertir resultados a DataFrame (en CPU)
    custom_describe = pd.DataFrame({
        'mean': mean.cpu().numpy(),
        'std': std.cpu().numpy(),
        'skew': skew.cpu().numpy(),
        'kurtosis': kurt.cpu().numpy()
    }, index=numeric_df.columns)

    # Mostrar resumen
    display(custom_describe)
    return custom_describe
```

Figura 2: Función para cálculo de momentos estadísticos

### 3.2.2. Distancia Jensen-Shannon

Para cada característica se calculó la distancia JS entre las distribuciones de ataques y conexiones normales:

```
# 2) Histogramas + distancias de Jensen-Shannon
def jensen_shannon(p, q):
    # p, q son arreglos numpy con valores positivos que suman 1
    p = np.asarray(p, dtype=np.float64)
    q = np.asarray(q, dtype=np.float64)
    # Asegurar normalización
    p = p / (p.sum() + 1e-20)
    q = q / (q.sum() + 1e-20)
    m = 0.5 * (p + q)
    # Divergencia KL base e; JS = 0.5 KL(p||m) + 0.5 KL(q||m)
    js = 0.5 * (entropy(p, m) + entropy(q, m))
    return sqrt(js) # retorna la distancia de Jensen-Shannon (raíz de la divergencia)
```

Figura 3: Implementación de distancia Jensen-Shannon

### 3.3. Implementación del Árbol CART

#### 3.3.1. Estructura de la Clase Node\_CART

A continuación se presenta la estructura completa para representar nodos del árbol:

```
class Node_CART:
    def __init__(self, num_classes=4, ref_CART=None, current_depth=0, min_samples_leaf=1):
        # Inicializa atributos del nodo (umbral, característica, hijos, gini, etc.)
        pass

    def to_xml(self, current_str=""):
        # Devuelve la representación XML del nodo y sus hijos
        pass

    def is_leaf(self):
        # Retorna True si el nodo no tiene hijos
        pass

    @staticmethod
    def calculate_gini(data_partition_torch, num_classes=2):
        # Calcula el índice Gini de un conjunto de datos
        pass

    def create_with_children(self, data_torch, current_depth, min_gini=0.000001):
        # Construye el subárbol recursivamente a partir de este nodo
        pass

    def select_best_feature_and_thresh(self, data_torch, num_classes=2):
        # Busca la mejor característica y umbral para dividir el nodo
        pass

    def evaluate_node(self, input_torch):
        # Evalúa un nodo dado un vector de entrada (predice clase)
        pass
```

Figura 4: Clase Node\_CART - Estructura básica

#### 3.3.2. Cálculo del Coeficiente Gini

Implementación matricial eficiente del coeficiente Gini:

```
@staticmethod
def calculate_gini(data_partition_torch, num_classes=2):
    if data_partition_torch.numel() == 0:
        return 0.0
    labels = data_partition_torch[:, -1].long()
    # Contar cuántos ejemplos hay de cada clase
    counts = torch.bincount(labels, minlength=num_classes)
    probs = counts.float() / counts.sum()
    gini = 1.0 - torch.sum(probs ** 2)
    return gini.item()
```

Figura 5: Cálculo del coeficiente Gini

### 3.3.3. Selección de Mejor División

El corazón del algoritmo CART, el cual selecciona la mejor característica y umbral:



```

def select_best_feature_and_thresh(self, data_torch, num_classes=2):
    # --- 1. Inicialización ---
    n_samples, n_features = data_torch.shape[0], data_torch.shape[1] - 1
    best_gini = float('inf')
    best_feature = None
    best_thresh = None

    labels = data_torch[:, -1].long() # Extraer etiquetas

    # --- 2. Evaluar cada característica ---
    for feature_idx in range(n_features):
        feature_values = data_torch[:, feature_idx]

        # Ordenar valores y etiquetas asociadas
        sorted_vals, sorted_idx = torch.sort(feature_values)
        sorted_labels = labels[sorted_idx]

        # Obtener valores únicos consecutivos como posibles umbrales
        uniq_vals, _ = torch.unique_consecutive(sorted_vals, return_inverse=True)
        if len(uniq_vals) == 1:
            continue # No se puede dividir si todos los valores son iguales

        thresholds = uniq_vals[:-1] # Usar todos excepto el último

        # --- 3. Probar cada umbral ---
        for thresh in thresholds:
            left_mask = sorted_vals < thresh
            right_mask = ~left_mask

            if left_mask.sum() == 0 or right_mask.sum() == 0:
                continue # Evitar divisiones vacías

            # Calcular Gini de cada partición
            gini_left = self.calculate_gini(sorted_labels[left_mask].unsqueeze(1), num_classes)
            gini_right = self.calculate_gini(sorted_labels[right_mask].unsqueeze(1), num_classes)

            # Índice Gini ponderado
            weighted_gini = (left_mask.sum() * gini_left + right_mask.sum() * gini_right)

            # --- 4. Actualizar mejor división ---
            if weighted_gini < best_gini:
                best_gini = weighted_gini
                best_feature = feature_idx
                best_thresh = thresh.item()

    # --- 5. Devolver mejor característica, umbral y Gini ---
    return best_feature, best_thresh, best_gini

```

Figura 6: Selección de mejor característica y umbral

### 3.3.4. Construcción Recursiva del Árbol

```

def create_with_children(self, data_torch, current_depth, min_gini=1e-6):
    # --- 1. Determinar clase dominante ---
    labels = data_torch[:, -1].long()
    counts = torch.bincount(labels, minlength=self.num_classes).float()
    self.dominant_class = torch.argmax(counts).item()

    # --- 2. Condiciones para no dividir (hacer hoja) ---
    if (counts > 0).sum() == 1:
        return self
    if self.max_depth is not None and current_depth >= self.max_depth:
        return self
    if data_torch.shape[0] < self.min_samples_leaf * 2:
        return self

    # --- 3. Buscar mejor división ---
    feature, thresh, gini = self.select_best_feature_and_thresh(data_torch, self.num_classes)
    if feature is None or gini < min_gini:
        return self

    # --- 4. Guardar información de la división ---
    self.feature_num = feature
    self.threshold_value = thresh
    self.gini = gini

    # --- 5. Particionar datos ---
    left_mask = data_torch[:, feature] < thresh
    right_mask = ~left_mask
    left_partition, right_partition = data_torch[left_mask], data_torch[right_mask]

    # Si alguna partición es muy pequeña, detener división
    if left_partition.shape[0] < self.min_samples_leaf or right_partition.shape[0] < self.min_samples_leaf:
        return self

    # --- 6. Crear nodos hijos recursivamente ---
    self.node_left = Node_CART(num_classes=self.num_classes, current_depth=current_depth+1, min_gini=min_gini)
    self.node_right = Node_CART(num_classes=self.num_classes, current_depth=current_depth+1, min_gini=min_gini)
    self.node_left.max_depth = self.max_depth
    self.node_right.max_depth = self.max_depth

    self.node_left.create_with_children(left_partition, current_depth+1, min_gini)
    self.node_right.create_with_children(right_partition, current_depth+1, min_gini)

    return self

```

Figura 7: Construcción recursiva con criterios de parada

### 3.4. Evaluación del Modelo

#### 3.4.1. Función de Evaluación

La función `test_CART` evalúa el desempeño de un árbol de decisión CART ya entrenado. Para cada muestra del conjunto de datos, obtiene la predicción recorriendo el árbol desde la raíz y la compara con la etiqueta real. Al final calcula la tasa de aciertos (accuracy) como la proporción entre el número de predicciones correctas y el total de muestras, devolviendo un valor entre 0 y 1 que refleja la precisión del modelo.

```
def test_CART(root_node, D):  
    """  
    Evalúa un árbol CART previamente entrenado (root_node) sobre un conjunto de datos D (tensor).  
    Calcula y retorna la tasa de aciertos (accuracy), definida como:  
        accuracy = c / n  
    donde:  
        c = número de estimaciones correctas (predicción == etiqueta real)  
        n = número total de muestras  
  
    Parámetros:  
        root_node (Node_CART): nodo raíz del árbol entrenado.  
        D (torch.Tensor): conjunto de datos, última columna es la etiqueta.  
  
    Retorna:  
        float: tasa de aciertos (accuracy).  
    """  
    # Contador de aciertos  
    correct = 0  
    n = D.shape[0]  
  
    # Para cada muestra en D  
    for i in range(n):  
        sample = D[i, :-1] # Todas las columnas menos la última (atributos)  
        true_label = D[i, -1].item() # Última columna (etiqueta real)  
        pred_label = root_node.evaluate_node(sample) # Predicción del árbol  
  
        if pred_label == true_label:  
            correct += 1  
  
    accuracy = correct / n  
    return accuracy
```

Figura 8: Función de evaluación del árbol

## 4. Pruebas Unitarias y Validación

### 4.1. Estrategia de Testing

Se implementó un conjunto exhaustivo de pruebas unitarias para validar cada componente del sistema:

#### 4.1.1. Pruebas para `calculate_gini`

Estas pruebas unitarias verifican que la función `calculate_gini` implementada en la clase `Node_CART` esté calculando correctamente el índice de impureza Gini, que es una métrica fundamental para evaluar la calidad de una división en un árbol de decisión.

El índice Gini mide qué tan mezcladas están las clases en un conjunto:

1.  $\text{Gini} = 0 \rightarrow$  el grupo es completamente puro (solo una clase).
2. Gini máximo (0.5 para dos clases)  $\rightarrow$  las clases están balanceadas (impureza máxima).

```

###-----
### Unit Tests función: calculate_gini
###-----

# Prueba unitaria 1: Gini para un solo grupo homogéneo (impureza debe ser 0)
def test_gini_homogeneous():
    node = Node_CART(num_classes=2)
    # Todos los elementos son de la clase 0
    data = torch.tensor([[1, 2, 0], [2, 3, 0], [3, 4, 0]], dtype=torch.float32)
    gini = node.calculate_gini(data, num_classes=2)
    print('Test 1 - Gini homogéneo:', gini)
    assert abs(gini - 0.0) < 1e-6, f"Esperado 0.0, obtenido {gini}"

# Prueba unitaria 2: Gini para dos clases balanceadas (impureza máxima)
def test_gini_balanced():
    node = Node_CART(num_classes=2)
    # Mitad clase 0, mitad clase 1
    data = torch.tensor([[1, 2, 0], [2, 3, 1], [3, 4, 0], [4, 5, 1]], dtype=torch.float32)
    gini = node.calculate_gini(data, num_classes=2)
    print('Test 2 - Gini balanceado:', gini)
    assert abs(gini - 0.5) < 1e-6, f"Esperado 0.5, obtenido {gini}"

# Ejecutar pruebas
print(" *** Unit test de calculate_gini ")
test_gini_homogeneous()
test_gini_balanced()

```

Figura 9: Pruebas unitarias para Gini

#### 4.1.2. Justificación de las Pruebas Seleccionadas

##### Caso Homogéneo:

Esta prueba asegura que el nodo reconoce correctamente situaciones puras, fundamentales para decidir que un nodo es hoja y no debe subdividirse.

- **Relevancia teórica:** Representa el estado de pureza máxima en árboles de decisión
- **Criterio de parada:** Los árboles CART usan  $\text{Gini} = 0$  como condición terminal
- **Detección de errores:** Fallas indican problemas en el cálculo de probabilidades

##### Caso Balanceado:

Si las clases están perfectamente balanceadas (50%-50%), la impureza Gini debe ser máxima para dos clases. Asegura que la función reconoce correctamente los casos de máxima incertidumbre, lo cual es clave para comparar divisiones durante el entrenamiento del árbol.

- **Máxima impureza:** Valor de referencia para distribuciones uniformes
- **Benchmark:**  $\text{Gini} = 0.5$  es el máximo teórico para clasificación binaria
- **Validación de normalización:** Confirma cálculo correcto de probabilidades

#### 4.1.3. Pruebas para select\_best\_feature\_and\_thresh

Estas pruebas unitarias verifican que la función `select_best_feature_and_thresh` identifique correctamente la mejor característica y el mejor umbral para dividir un conjunto de datos, minimizando el índice Gini. Se evalúa su capacidad para detectar divisiones perfectamente separables en distintos escenarios.

Test 1: Dataset donde la columna 0 permite separar perfectamente las clases con un threshold de 3.0.

Test 2: Dataset donde solo la columna 2 permite separación perfecta con threshold 13.0.

```
def test_select_best_feature_and_thresh():
    # El feature 0 separa perfectamente las clases con threshold 2
    # [feature0, feature1, clase]
    data = torch.tensor([
        [1.0, 10.0, 0],
        [2.0, 20.0, 0],
        [3.0, 30.0, 1],
        [4.0, 40.0, 1]
    ])

    node = Node_CART(num_classes=2)
    best_feature, best_thresh, best_gini = node.select_best_feature_and_thresh(data, num_classes=2)

    print("Mejor feature:", best_feature)
    print("Mejor threshold:", best_thresh)
    print("Mejor gini:", best_gini)

    # Esperamos que el mejor feature sea 0 y el mejor threshold sea 3.0 (ambos separan perfectamente)
    assert best_feature == 0, "El mejor feature debería ser la columna 0"
    assert best_thresh == 3, "El mejor threshold debería ser 3.0"
    assert best_gini == 0.0, "El gini debería ser 0 para una separación perfecta"

    ###
    ### Unit Tests función: select_best_feature_and_thresh
    ###

def test_select_best_feature_and_thresh_feature2():
    # Dataset: solo la columna 2 permite separación perfecta con threshold 12.5
    # [feature0, feature1, feature2, clase]
    data = torch.tensor([
        [0.0, 1.0, 10.0, 1],
        [0.0, 1.0, 11.0, 1],
        [0.0, 1.0, 12.0, 1],
        [0.0, 1.0, 13.0, 0],
        [0.0, 1.0, 14.0, 0],
        [0.0, 1.0, 15.0, 0],
        [0.0, 1.0, 16.0, 0],
        [0.0, 1.0, 17.0, 0],
    ])

    node = Node_CART(num_classes=2)
    best_feature, best_thresh, best_gini = node.select_best_feature_and_thresh(data, num_classes=2)

    print("Mejor feature:", best_feature)
    print("Mejor threshold:", best_thresh)
    print("Mejor gini:", best_gini)

    # Esperamos que el mejor feature sea 2 (columna 2) y threshold 13.0
    assert best_feature == 2, "El mejor feature debería ser la columna 2"

    assert best_thresh == 13.0, "El mejor threshold debería ser entre 13"
    assert best_gini == 0.0, "El gini debería ser 0 para una separación perfecta"
```

Figura 10: Pruebas unitarias para `select_best_feature_and_thresh`

#### 4.1.4. Justificación de las Pruebas Seleccionadas

##### Caso Feature 0 Perfectamen Separador:

- **Propósito:** Confirmar que la función identifica correctamente la columna que separa completamente las clases.
- **Relevancia teórica:** Refleja el escenario ideal donde una sola característica permite una división sin mezcla de clases, lo que debe producir un  $Gini = 0$ .
- **Validación de cálculo:** Asegura que la función compute correctamente Gini ponderado y selecciona el umbral óptimo.
- **Detección de errores:** Fallas indicarían problemas en la lógica de ordenamiento, umbral o cálculo de Gini.

##### Caso Feature 2 Perfectamente Separador:

- **Propósito:** Verificar que la función pueda identificar la mejor columna incluso cuando no es la primera.
- **Relevancia teórica:** Evalúa la generalidad del método y su capacidad de manejar datasets con múltiples características.
- **Benchmark:** Confirma que el algoritmo selecciona la característica correcta y el threshold que produce separación perfecta.

Detección de errores: Un resultado incorrecto sugiere errores en la iteración sobre features o cálculo de Gini ponderado.

#### 4.1.5. Pruebas para `create_with_children` y `test_CART`

Estas pruebas verifican la construcción recursiva del árbol CART y la exactitud de las predicciones:

##### Caso Test Árbol Perfectamente Separado:

Dataset separable por feature 0 con threshold 2.0.

Se construye el árbol y se evalúa la accuracy, que debe ser 1.0.

Valida que `create_with_children` genere nodos hoja correctamente y que `test_CART` calcule correctamente la tasa de aciertos.

##### Caso Árbol con Min Gini Mayor a 0:

Dataset donde algunas divisiones se detienen prematuramente por el parámetro `min_gini`.

Se espera accuracy menor que 1 pero mayor que 0, reflejando que el árbol no alcanza separación perfecta.

**Justificación:** comprueba que el parámetro `min_gini` afecta la profundidad y la división de nodos correctamente.

##### Caso Árbol con Min Gini = 0:

Igual dataset anterior, pero `min_gini = 0`.

Se espera accuracy máxima, verificando que el árbol explora todas las divisiones posibles hasta nodos hoja puros.

**Relevancia teórica:** asegura que el parámetro `min_gini` controla correctamente el criterio de parada basado en impureza.

```

###
### Unit Tests función: test_card
###
def test_CART_simple_CART():
    # Dataset perfectamente separable por feature 0 con threshold 2.0
    data = torch.tensor([
        [1.0, 10.0, 0],
        [2.0, 20.0, 0],
        [3.0, 30.0, 1],
        [4.0, 40.0, 1]
    ])

    node = Node_CART(num_classes=2)
    node.create_with_children(data, current_depth=0, min_gini=0.0)
    acc = test_CART(node, data) # accuracy
    print("Accuracy árbol perfectamente valanceado:", acc)
    assert acc == 1.0, "El accuracy debería ser 1"

def test_CART_DEPTH_2():
    data = torch.tensor([
        [1.0, 0.0, 1.0, 0],
        [1.0, 0.0, 2.0, 0],
        [1.0, 0.0, 3.0, 1],
        [1.0, 0.0, 4.0, 0],
        [1.0, 0.0, 5.0, 0],
        [1.0, 0.0, 6.0, 0],
    ])

    node = Node_CART(num_classes=2)
    node.create_with_children(data, current_depth=0)
    acc = test_CART(node, data) # accuracy
    print("Accuracy árbol usando un min gini mayor a 0", acc)
    assert acc < 1.0 and acc > 0, "El accuracy debería ser menor que uno y mayor a 0, usando un min gini mayor a 0"

def test_CART_DEPTH_2_Gin0():
    data = torch.tensor([
        [1.0, 0.0, 1.0, 0],
        [1.0, 0.0, 2.0, 0],
        [1.0, 0.0, 3.0, 1],
        [1.0, 0.0, 4.0, 0],
        [1.0, 0.0, 5.0, 0],
        [1.0, 0.0, 6.0, 0],
    ])

    node = Node_CART(num_classes=2)
    node.create_with_children(data, current_depth=0, min_gini=0)
    acc = test_CART(node, data) # accuracy
    print("Accuracy árbol usando un min gini 0", acc)
    assert acc < 1, "El accuracy debería ser igual a 1"

```

Figura 11: Pruebas unitarias para construcción y evaluación del árbol CART

## 5. Resultados Experimentales

### 5.1. Análisis Exploratorio de Datos

#### 5.1.1. Momentos Estadísticos Clase Attack

La Tabla ?? presenta un resumen de los principales momentos estadísticos calculados para la clase Attack:

Index	mean	std ▼	skew	kurtosis
src_bytes	53666.890625	4720.02392578125	-6.3851165771484375	42.71146011352539
dst_bytes	8129.90771484375	918.66357421875	-5.925033092498779	35.95226287841797
dst_host_count	146.4183807373047	90.67967224121094	-0.05670534446835518	-1.559332251548767
dst_host_srv_count	146.4183807373047	90.67967224121094	-0.05670534446835518	-1.559332251548767
srv_count	3.829545259475708	2.043654441833496	2.9286463260650635	17.41671371459961
count	3.5247931480407715	1.7614456415176392	2.2946739196777344	13.489805221557617
duration	0.2933884263038635	1.6597687005996704	6.21148157119751	39.39738464355469
hot	1.9173552989959717	0.31391364336013794	-3.6669960021972656	15.058679580688477
flag_SF	0.8997933268547058	0.3002752959728241	-2.6628451347351074	5.090744972229004
flag_RSTR	0.0929751992225647	0.2903977036476135	2.80322265625	5.858059883117676
num_compromised	0.9245867133140564	0.264056921005249	-3.2158730030059814	8.341838836669922
srv_diff_host_rate	0.11796487122774124	0.24290232360363007	2.0257747173309326	3.2424793243408203
srv_rerror_rate	0.1338429600003815	0.2059033215045929	1.6743766069412231	2.7861294746398926
rerror_rate	0.07794421166181564	0.17457744479179382	2.8167147636413574	9.071499824523926
dst_host_rerror_rate	0.06356403976678848	0.10997559875249863	4.967799186706543	32.01710891723633
dst_host_srv_rerror_rate	0.06356403976678848	0.10997559875249863	4.967799186706543	32.01710891723633
dst_host_same_src_port_rate	0.023202477023005486	0.07885199040174484	9.212125778198242	101.26521301269531
flag_S2	0.005165289156138897	0.0716840997338295	13.805980682373047	188.60508728027344
srv_rerror_rate	0.006518593989312649	0.04870394244790077	8.558282852172852	77.85615539550781
diff_srv_rate	0.004586776718497276	0.04839825630187988	10.946174621582031	123.26040649414062
rerror_rate	0.006002065725624561	0.047427937388420105	9.020917892456055	85.87227630615234
flag_S1	0.00206611561588943	0.04540756344795227	21.931758880615234	479.00201416015625
same_srv_rate	0.9977169632911682	0.024057749658823013	-10.89884773254395	121.83165740966797
dst_host_same_src_port_rate	0.0020351239945739508	0.005500826984643936	3.190154790878296	11.172428131103516
dst_host_srv_rerror_rate	0.0020351239945739508	0.005500826984643936	3.190154790878296	11.172428131103516
logged_in	0.9999999403953552		0.0 Infinity	Infinity
dst_host_same_srv_rate	0.9999999403953552		0.0 Infinity	Infinity
protocol_type_tcp	0.9999999403953552		0.0 Infinity	Infinity
service_http	0.9999999403953552		0.0 Infinity	Infinity
land	0.0		0.0 NaN	NaN

Figura 12: Momentos Estadísticos para la clase Attack

El análisis de los momentos estadísticos revela un patrón asimétrico en el flujo de datos: los backdoors generan un promedio de 53,667 bytes en src\_bytes frente a solo 8,130 bytes en dst\_bytes. Esta asimetría (razón 6.6:1) sugiere un comportamiento típico de exfiltración de datos, donde el sistema comprometido envía significativamente más información de la que recibe.

### Patrones de Conectividad

Los backdoors muestran un comportamiento de conectividad selectivo pero distribuido, con valores promedio de 146.4 para dst\_host\_count y dst\_host\_srv\_count, indicando conexiones a múltiples hosts destino. Sin embargo, los valores bajos de count (3.5) y srv\_count (3.8) sugieren que estas conexiones son específicas y no masivas.

### Distribuciones Estadísticas Críticas

El análisis de asimetría (skewness) y curtosis revela distribuciones altamente no-normales en variables clave:

- same\_srv\_rate: skewness = -10.9, kurtosis = 121.8
- dst\_host\_same\_src\_port\_rate: skewness = 9.2, kurtosis = 101.3
- flag\_S2: skewness = 13.8, kurtosis = 188.6



- Estas distribuciones extremas indican eventos raros pero significativos, característicos de actividades maliciosas especializadas.

La Tabla ?? presenta un resumen de los principales momentos estadísticos calculados para la clase Normal:

[illegible]

El tráfico normal presenta un patrón inverso al de los backdoors: 3,721 bytes en `dst_bytes` frente a 1,270 bytes en `src_bytes`, mostrando una razón de 2.9:1 a favor de los datos recibidos. Este comportamiento es típico de usuarios que consumen servicios web, descargan contenido o reciben respuestas de servidores. La alta variabilidad ( $\text{std} > 36.000$  para ambas métricas) indica gran diversidad en los tipos de comunicación.

La duración promedio de las conexiones es significativamente mayor que en backdoors: 188.9 segundos comparado con 0.29 segundos. Esto refleja sesiones interactivas genuinas, descargas de archivos, o comunicaciones sostenidas típicas del uso legítimo de la red.

Los valores de conectividad muestran un comportamiento más diversificado:

- `dst_host_count`: 139.6 (similar a backdoors)
- `dst_host_srv_count`: 203.5 (mayor diversidad de servicios)
- `count`: 8.85 y `srv_count`: 11.9 (mayor actividad de conexión)

Esta diversidad refleja el comportamiento natural de usuarios explorando diferentes servicios y destinos.

### **Distribuciones Estadísticas Extremas**

El tráfico normal presenta múltiples variables con distribuciones altamente asimétricas:

Variables con curtosis extrema ( $> 1000$ ):

- `num_file_creations`: skewness = 88.1, curtosis = 9,106
- `num_access_files`: skewness = 26.3, curtosis = 1,401
- `service_pop_3`: skewness = 33.5, curtosis = 1,121
- `•flag_RSTO`: skewness = 36.4, curtosis = 1,326
- `num_failed_logins`: skewness = 130.6, curtosis = 19,414

Estas distribuciones extremas indican que estas actividades son raras pero presentes en el tráfico normal, representando operaciones específicas como creación de archivos, accesos especiales o errores de conexión ocasionales.

### **Diversidad de Protocolos y Servicios**

A diferencia de los backdoors, el tráfico normal muestra:

- `protocol_type_tcp`: 86.3 % (no exclusivo)
- `protocol_type_udp`: 12.7 % (presente)
- `service_http`: 69.3 % (predominante pero no exclusivo)
- `service_smtp`: 10.9 % (correo electrónico)

Esta diversidad refleja el uso variado de servicios de red en operaciones normales.

### **Estados de Conexión Variados**

El tráfico normal presenta mayor diversidad en estados de conexión:

- `flag_SF`: 94.4 % (conexiones exitosas)
- `logged_in`: 79.3 % (no todas las conexiones requieren autenticación)
- Presencia de flags de error: REJ, RSTO, S0, S1

### **Indicadores de Actividad Administrativa**

El tráfico normal incluye actividades que están ausentes en backdoors:

- `num_root`: 0.062 (accesos administrativos legítimos)
- `num_file_creations`: 0.005 (creación legítima de archivos)
- `su_attempted`: muy bajo pero presente

### 5.1.3. Comparación con Backdoors

Diferencias clave respecto a ataques backdoor:

Flujo de datos: Normal recibe más de lo que envía (2.9:1), backdoors envían más (6.6:1)

Duración: Normal 188.9s vs Backdoor 0.29s

Diversidad: Normal usa múltiples protocolos y servicios, backdoors solo TCP/HTTP

Actividad administrativa: Normal tiene actividad root legítima, backdoors no

Errores: Normal presenta errores ocasionales, backdoors operan "limpiamente"

## 5.2. Análisis Exploratorio de Histogramas

Para cada característica, se gráfico el histograma para categoría ataque (backdoor) y normal, y se computo la distancia de Jensen-Shannon entre ambas aproximaciones de las densidades. A continuación se muestra solamente las características con una Distancia de JS superiores a 0.5.

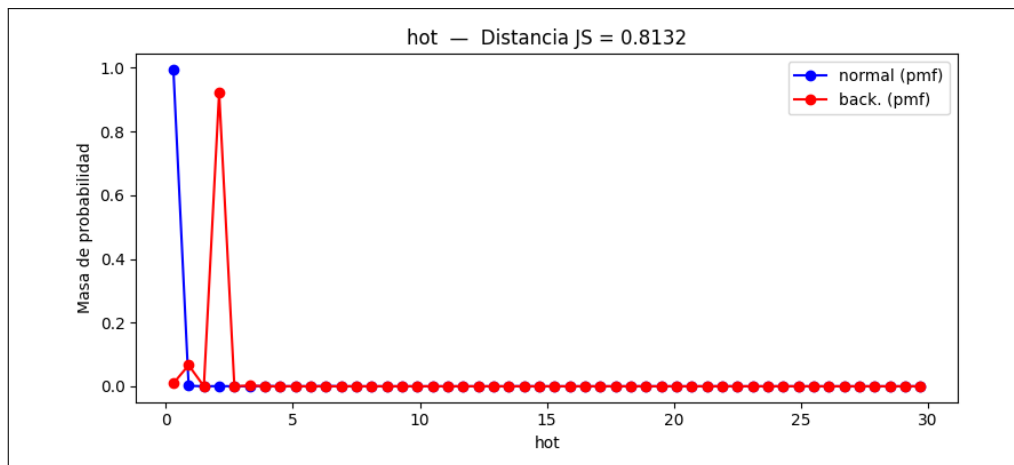


Figura 14: Histograma hot

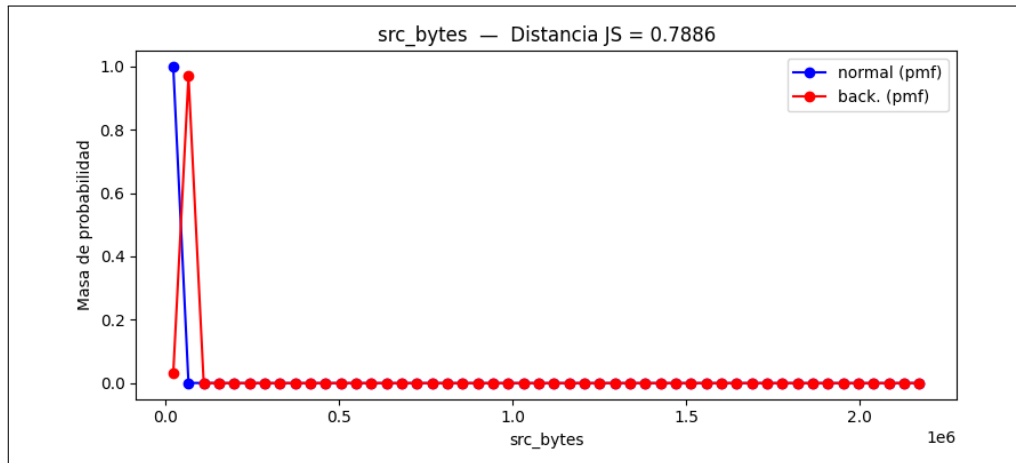


Figura 15: Histograma src\_byte

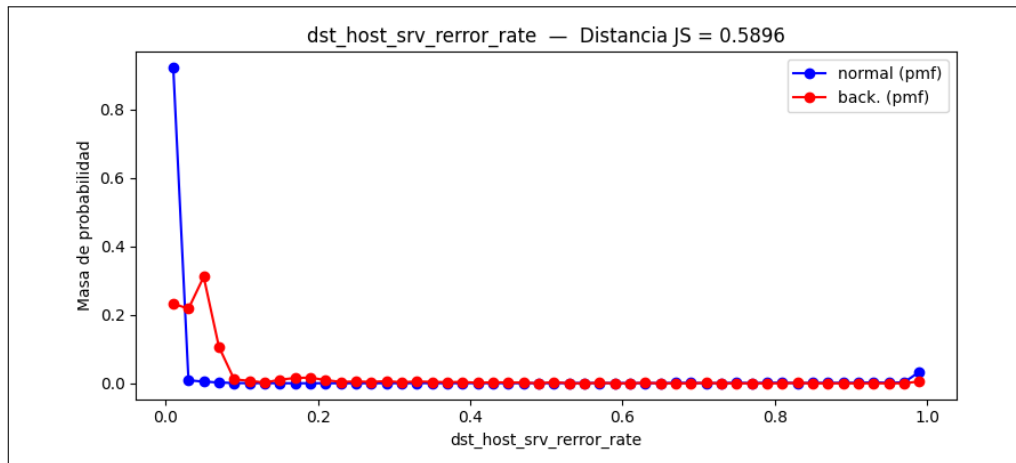


Figura 16: Histograma dst\_host\_srv\_error\_rate

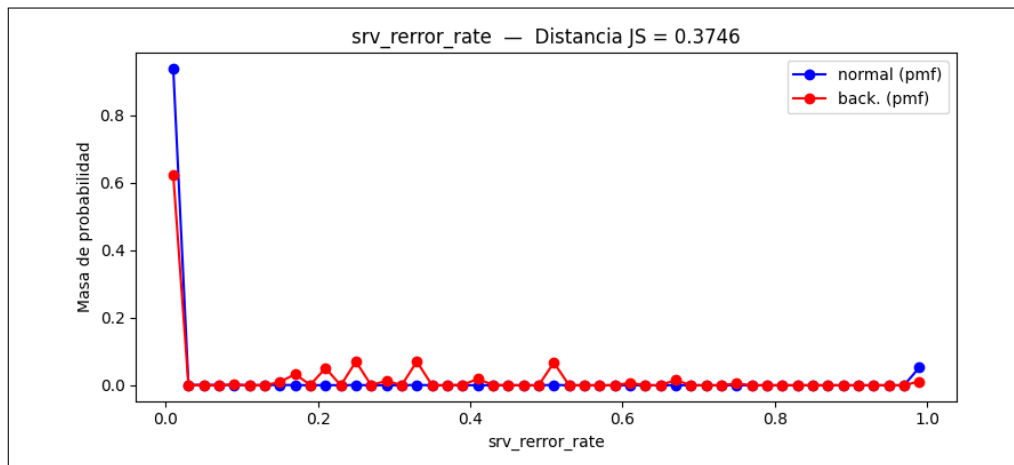


Figura 17: Histograma host\_error\_rate

### Análisis de Histogramas con Jensen-Shannon Divergence $> 0.5$

El análisis de los histogramas con alta divergencia Jensen-Shannon ( $JS > 0.5$ ) revela características distintivas clave para la detección de ataques backdoor en el dataset KDD'99.

Variables con Mayor Poder Discriminativo

**hot** ( $JS = 0.8132$ )

Esta variable muestra la diferencia más extrema entre tráfico normal y backdoors:

- Tráfico normal: Concentración casi total en valor 0 ( $>99\%$  de probabilidad)
- Ataques backdoor: Distribución bimodal con picos en valores 0 y 2, y presencia significativa en valores intermedios hasta 30
- Interpretación: Los backdoors generan más "hotlists" o accesos a recursos críticos del sistema

**src\_bytes** ( $JS = 0.7886$ )

Patrón de bytes enviados muestra clara diferenciación:

- Tráfico normal: Concentración extrema en valores cercanos a 0
- Ataques backdoor: Distribución más dispersa con cola larga hacia valores altos (hasta  $2 \times 10^6$  bytes)
- Interpretación: Los backdoors tienden a enviar volúmenes de datos significativamente mayores, confirmando el patrón de exfiltración dst\_host\_srv\_error\_rate ( $JS = 0.5896$ )

Tasa de errores de re-respuesta por servicio en host destino:

- Tráfico normal: Concentración masiva en 0 (~95 %) con pequeño pico secundario
- Ataques backdoor: Distribución más uniforme con picos múltiples en valores bajos (0.05-0.15)
- Interpretación: Los backdoors generan patrones de error más diversos, posiblemente por técnicas de evasión

**dst\_host\_error\_rate** (JS = 0.5809)

Tasa de errores de re-respuesta en host destino:

- Patrón muy similar al anterior
- Tráfico normal: Dominancia absoluta del valor 0
- Ataques backdoor: Distribución escalonada con múltiples picos pequeños
- Interpretación: Los backdoors provocan más errores de reconexión durante su operación

Variables con Divergencia Moderada pero Significativa

**srv\_error\_rate** (JS = 0.3746)

- Tráfico normal: Concentración extrema en 0
- Ataques backdoor: Mayor dispersión con valores distribuidos hasta 1.0
- Muestra que los backdoors generan más errores a nivel de servicio

**srv\_count** (JS = 0.3292)

- Tráfico normal: Decaimiento exponencial desde valores bajos
- Ataques backdoor: Pico muy alto en valor 0, seguido de dispersión más uniforme
- Los backdoors tienden a ser más específicos en cuanto a servicios utilizados

## Patrones de Discriminación Identificados

### Concentración vs Dispersión

El patrón más consistente es que el tráfico normal se concentra masivamente en valores mínimos (típicamente 0), mientras que los backdoors muestran distribuciones más dispersas o multimodales.

### Comportamiento de Errores

Los backdoors generan patrones de error más complejos y variados, lo que puede indicar:

- Técnicas de evasión que causan reconexiones
- Comportamiento automatizado que no maneja errores como el tráfico humano
- Operaciones que fuerzan estados no comunes en los servicios

## Volumen de Datos

La variable `src_bytes` confirma que los backdoors son fundamentalmente diferentes en su patrón de envío de datos, con una cola larga hacia valores altos que representa actividad de exfiltración.

Implicaciones para Detección

### Variables Críticas para Clasificación

Las variables con  $JS > 0.5$  son candidatos ideales para algoritmos de clasificación:

1. `hot`: Discriminador más potente ( $JS = 0.8132$ )
2. `src_bytes`: Indicador clave de exfiltración ( $JS = 0.7886$ )
3. `dst_host_srv_error_rate` y `dst_host_error_rate`: Patrones de error distintivos

Umbral de Detección

- `hot`  $> 0$ : Altamente sospechoso para backdoors
- `src_bytes` con cola larga: Indicador de exfiltración
- Tasas de error  $> 0$  en múltiples métricas simultáneamente: Patrón típico de backdoor

## 5.3. Evaluación del CART

### 5.3.1. Evaluación con una profundidad máxima de 3 siempre con mínimo 2 observaciones por hoja.

Al evaluar el cart con una profundidad máxima de 3 y al menos 2 observaciones por hoja, el árbol CART alcanza un desempeño sobresaliente: accuracy de 0,9997184684684685 y F1-score promedio de 0.993. Esto significa que el modelo clasifica correctamente casi todos los ejemplos y mantiene un equilibrio adecuado entre precisión y exhaustividad.

La elección de una profundidad de 3 responde a la necesidad de mantener el árbol interpretable, evitando sobreajuste y garantizando nodos suficientemente representativos. Además, se observa que los atributos utilizados en los splits principales corresponden a aquellos con mayor distancia de Jensen–Shannon, lo cual confirma que el modelo explota las variables más discriminativas.

Finalmente, se deja planteada la comparación con profundidades menores y mayores (2 y 4), que será desarrollada en las siguientes secciones, con el fin de identificar si el desempeño observado en esta configuración representa un punto óptimo de balance entre simplicidad e incremento en la capacidad predictiva

```
# Entrenamos el árbol con todo el dataset: Max Depth = 3
node = Node_Cart(nom_clasificado, min_samples_leaf=2)
node.max_depth = 3
node.create_with_children(data_train, current_depth)

# Evaluamos el árbol con el mismo dataset
y_true = data_train[:, -1].astype(int)
y_pred = [node.evaluate_node(row[-1]) for row in data_train]

accuracy = sum(yt == yp for yt, yp in zip(y_true, y_pred)) / len(y_true)
f1 = f1_score(y_true, y_pred, method='macro')

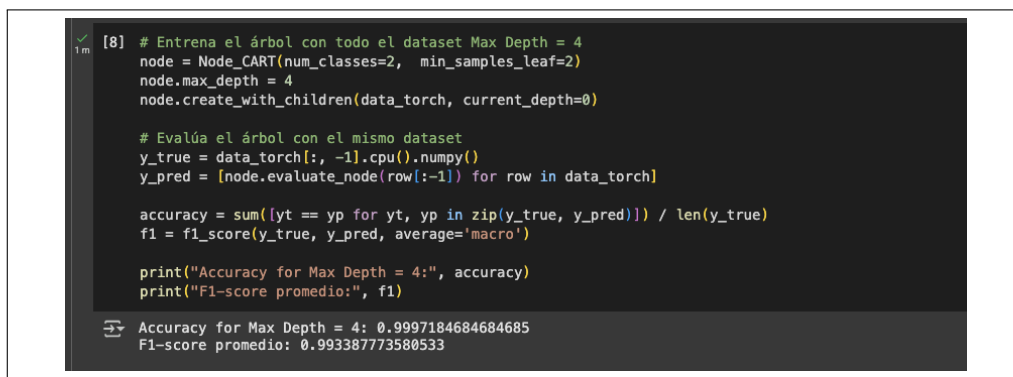
print("Accuracy for Max Depth = 3:", accuracy)
print("F1-score promedio:", f1)
```

Accuracy for Max Depth = 3: 0.9997184684684685  
F1-score promedio: 0.99377558053

Figura 18: Evaluación Cart, Profundidad Máxima 3 y observaciones mínimas 2 por hoja

### 5.3.2. Evaluación con una profundidad máxima de 4 con mínimo 2 observaciones por hoja.

También al evaluar el CART con una profundidad máxima de 4 y con mínimo 2 observaciones obtenemos un accuracy de 0,9997184684684685 y un F1-score promedio de 0.993. Al aumentar la profundidad máxima del árbol a 4 se observa que las métricas de desempeño permanecen prácticamente idénticas a las obtenidas con profundidad 3. Esto evidencia que incrementar la complejidad del modelo no aporta mejoras significativas en este caso. Aunque en árboles de decisión mayores profundidades pueden conducir a sobreajuste, en este escenario el comportamiento se mantiene estable. No obstante, se aprecia que la profundidad de 3 ya logra capturar la estructura discriminativa del problema, por lo que el incremento a 4 niveles añade complejidad sin beneficios claros en términos de precisión.



```
[8] # Entrena el árbol con todo el dataset Max Depth = 4
    node = Node_CART(num_classes=2, min_samples_leaf=2)
    node.max_depth = 4
    node.create_with_children(data_torch, current_depth=0)

    # Evalúa el árbol con el mismo dataset
    y_true = data_torch[:, -1].cpu().numpy()
    y_pred = [node.evaluate_node(row[:-1]) for row in data_torch]

    accuracy = sum([yt == yp for yt, yp in zip(y_true, y_pred)]) / len(y_true)
    f1 = f1_score(y_true, y_pred, average='macro')

    print("Accuracy for Max Depth = 4:", accuracy)
    print("F1-score promedio:", f1)
```

Accuracy for Max Depth = 4: 0.9997184684684685  
F1-score promedio: 0.993387773580533

Figura 19: Evaluación Cart, Profundidad Máxima 4 y observaciones mínimas 2 por hoja

### 5.3.3. Evaluación de profundidad máxima de 2 y 3 nodos de 10 particiones

Se evalúo el CART implementado usando 10 particiones aleatorias del conjunto de datos, con un 70 % del conjunto de datos como conjunto de datos de entrenamiento, y el restante 30 % como conjunto de datos de prueba, para profundidades máximas de 2 y 3 nodos. Los resultados muestran valores promedio de accuracy superiores al 0,9997184684684685 y F1-score cercanos a 0.993, con desviaciones estándar prácticamente nulas. Esto indica que el modelo mantiene un comportamiento estable y confiable sin importar la partición utilizada.

Al comparar entre profundidades, se observa que el salto de 2 a 3 nodos trae una ligera mejora en F1-score, confirmando que una mayor profundidad permite capturar mejor la estructura del conjunto de datos sin comprometer la simplicidad del modelo. En términos de eficiencia, los tiempos de entrenamiento y evaluación permanecen bajos y estables en todas las corridas, lo que refuerza la viabilidad del CART como clasificador para este problema de detección de intrusiones.



	max_depth	accuracy_mean	accuracy_std	f1_mean	f1_std	train_time_mean	train_time_std	eval_time_mean	eval_time_std
0	3	0.999700	0.000000	0.992781	0.000000	35.227409	0.000000	2.766498	0.000000
1	3	0.999625	0.000075	0.991334	0.001448	35.333855	0.106446	2.697910	0.068588
2	3	0.999437	0.000272	0.987934	0.004952	35.132804	0.297316	2.587271	0.166188
3	3	0.999174	0.000513	0.981815	0.011433	34.977319	0.372582	2.515623	0.190037
4	3	0.998994	0.000583	0.978244	0.012473	34.885164	0.380827	2.470663	0.192234
5	3	0.999105	0.000588	0.980551	0.012501	34.969840	0.395864	2.455420	0.178818
6	3	0.998970	0.000637	0.977837	0.013348	34.968779	0.366508	2.435113	0.172864
7	3	0.999062	0.000643	0.979729	0.013453	34.983684	0.345098	2.424523	0.164109
8	3	0.999028	0.000613	0.979096	0.012809	34.983686	0.325361	2.439853	0.160685
9	3	0.998953	0.000624	0.977433	0.013136	34.938434	0.337199	2.480470	0.195155
10	4	0.999021	0.000633	0.978828	0.013279	34.964658	0.332030	2.487125	0.187259
11	4	0.999065	0.000623	0.979750	0.013076	35.000529	0.339426	2.476546	0.182688
12	4	0.999064	0.000599	0.979856	0.012569	35.017287	0.331237	2.486150	0.179177
13	4	0.999016	0.000603	0.978685	0.012827	35.832839	2.957786	2.511719	0.238340
14	4	0.998966	0.000611	0.977703	0.012925	36.561592	3.949734	2.547397	0.266157
15	4	0.999010	0.000615	0.978602	0.012990	36.522070	3.827375	2.561763	0.263643
16	4	0.998960	0.000629	0.977599	0.013225	36.447574	3.725037	2.589624	0.257697
17	4	0.999005	0.000640	0.978552	0.013440	36.394776	3.626625	2.559899	0.253626
18	4	0.998992	0.000625	0.978314	0.013120	36.929597	4.196283	2.585355	0.269453
19	4	0.998956	0.000629	0.977521	0.013246	37.400758	4.576704	2.609469	0.282884

Figura 20: Resultado de usar 10 particiones aleatorias para 2 y 3 nodos de profundidad

### 5.3.4. Gráfico de la mejor corrida usando F1-Score

El árbol que obtuvo el mejor desempeño (F1-score  $\approx 0.99475$ ) es poco profundo y altamente interpretable. Todos los nodos de la representación gráfica muestran valores de Gini cercanos a 0.00, lo que indica que, durante el entrenamiento, las hojas quedaron muy homogéneas (clase casi pura) y que los umbrales aprendidos separan con nitidez el tráfico normal del malicioso. Esto se refleja en el desempeño en prueba, donde el F1 elevado confirma buena generalización y bajo riesgo de sobreajuste para esta configuración.

Estructuralmente, la raíz divide con la condición  $X[1] < 40636$ . Esta variable actúa como separador principal del conjunto: cuando se cumple (rama izquierda) el modelo continúa refinando con umbrales adicionales sobre  $X[1]$  y con un atributo binario/entero  $X[9]$ ; cuando no se cumple (rama derecha), la separación final se apoya en  $X[31]$  y  $X[2]$ . En términos prácticos, el modelo está usando muy pocas variables en los niveles superiores, lo que sugiere que esas características son fuertemente discriminativas para distinguir “normal” vs. “backdoor”.

Las reglas de decisión más relevantes que se leen directamente del árbol son:

- Rama izquierda de la raíz ( $X[1] < 40636$ ):
  - Si además  $X[1] < 16952$  y  $X[9] < 1 \rightarrow$  clase 0 (normal).
  - Si  $X[1] < 16952$ ,  $X[9] \geq 1$  y  $X[1] < 6664 \rightarrow$  clase 0 (normal).
  - Si  $X[1] < 16952$ ,  $X[9] \geq 1$  y  $X[1] \geq 6664 \rightarrow$  clase 1 (backdoor).
  - Si  $X[1] \geq 16952 \rightarrow$  clase 0 (normal).
- Rama derecha de la raíz ( $X[1] \geq 40636$ ):
  - Si  $X[31] \geq 0.01 \rightarrow$  clase 0 (normal).
  - Si  $X[31] < 0.01$  y  $X[2] < 1460 \rightarrow$  clase 0 (normal).

– Si  $X[31] < 0.01$  y  $X[2] \geq 1460 \rightarrow$  clase 1 (backdoor).

Obsérvese que solo dos caminos conducen a clase 1:

- (i)  $X[1] < 16952$  con  $X[9] \geq 1$  y  $X[1] \geq 6664$ , y
- (ii)  $X[1] \geq 40636$  con  $X[31] < 0.01$  y  $X[2] \geq 1460$ .

El resto de combinaciones termina en clase 0. Este patrón es coherente con el desbalance del conjunto (predominan ejemplos “normal”) y, a la vez, evidencia que el modelo aprende umbrales muy específicos para identificar “backdoor” sin penalizar el desempeño global.

Los valores de Gini en los nodos internos refuerzan esta lectura: salvo un nodo intermedio con  $\text{Gini} \approx 0.03$ , el resto está en 0.00, lo que implica que los subconjuntos generados por los splits son prácticamente puros. Esto se alinea con el análisis previo mediante distancia de Jensen–Shannon, donde se observó que algunas variables mostraban gran divergencia entre “normal” y “backdoor”; aquí, esas variables (índices  $X[1]$ ,  $X[2]$ ,  $X[31]$  y  $X[9]$  en el espacio de características) aparecen precisamente en los niveles superiores del árbol.

En conjunto, el árbol evidencia tres propiedades clave: (1) parsimonia —pocos niveles y pocas variables dominan la decisión—, (2) separabilidad —umbrales que dejan hojas puras— y (3) estabilidad —un F1 elevado en validación que concuerda con la estructura del modelo—. Por ello, este árbol resulta adecuado para escenarios de detección en producción: es simple de explicar, económico de evaluar y captura reglas claras para los casos “backdoor” sin degradar la precisión en el tráfico normal. Si se requiere mayor trazabilidad, el mapeo de índices  $X[k]$  al nombre de cada columna (según el orden de las variables numéricas tras el preprocesamiento) puede incluirse como una pequeña tabla anexa para relacionar cada condición del árbol con el atributo original del dataset.

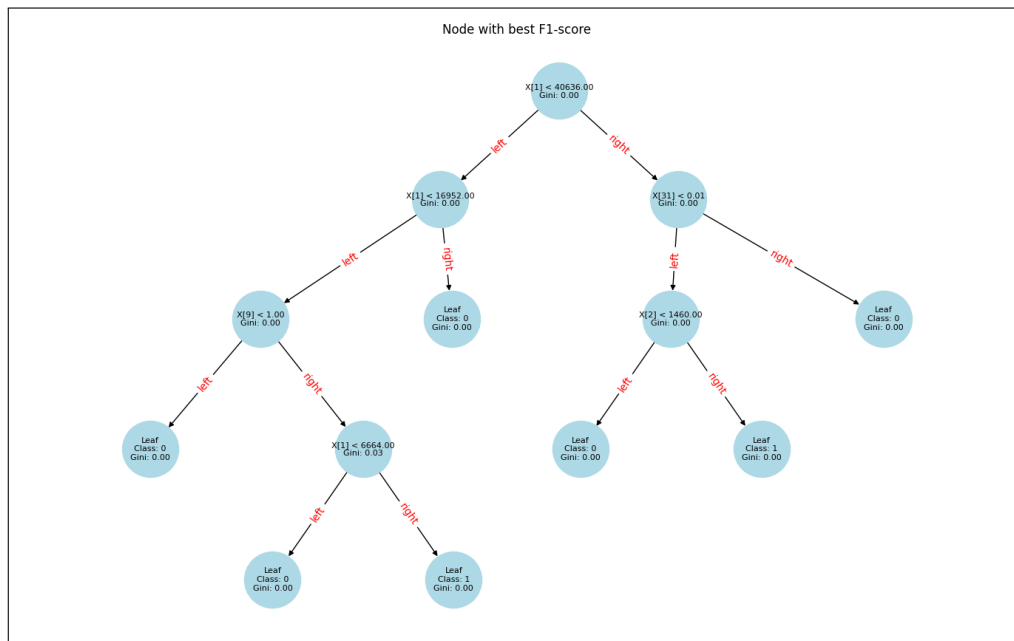


Figura 21: Gráfico de mejor corrida según F1-score

### 5.3.5. Eficiente de la construcción del árbol y su evaluación, usando la distancia de Jensen-Shannon.

Para aumentar la eficiencia en la construcción y evaluación del CART se podría utilizar la distancia de Jensen-Shannon como criterio previo de selección de variables. Esta métrica permite cuantificar qué tan diferentes son las distribuciones de una característica entre clases, de manera que atributos con valores de JS más altos aportan mayor capacidad discriminativa. Bajo esta lógica, se pueden filtrar solo aquellas variables que superen un umbral (por ejemplo, mayor a 0.3) o, alternativamente, seleccionar el top-N atributos con mayor JS.

El beneficio de este procedimiento es doble. En primer lugar, el árbol solo intentará realizar particiones (splits) sobre un subconjunto reducido de características realmente informativas, lo que evita desperdiciar recursos en variables redundantes o poco útiles. En segundo lugar, la reducción del espacio de búsqueda disminuye de manera significativa el costo computacional: se aceleran tanto las fases de entrenamiento (menos comparaciones al generar nodos) como de evaluación (árbol más compacto y menos profundo).

En el contexto del dataset KDD99, donde existen decenas de atributos, muchos de ellos derivados o con fuerte correlación, este tipo de prefiltrado resulta especialmente valioso. No solo permite mantener la interpretabilidad del árbol al concentrarse en las variables con mayor poder discriminativo, sino que también reduce el riesgo de sobreajuste, pues el modelo se ve forzado a trabajar con un conjunto más concentrado y robusto de predictores. En síntesis, la incorporación de la distancia de Jensen-Shannon como etapa previa actúa como un mecanismo de regularización implícito, mejorando la eficiencia y

la calidad del modelo sin sacrificar desempeño en métricas como accuracy o F1-score.

## 6. Conclusiones

Este trabajo ha demostrado la efectividad de un enfoque integral que combina análisis estadístico descriptivo, distancias Jensen–Shannon y árboles de decisión CART implementados en PyTorch, para abordar el problema de la detección de intrusos en el dataset KDD99.

Las contribuciones específicas incluyen:

1. Implementación completa: Desarrollo de un sistema CART funcional, optimizado mediante operaciones matriciales en PyTorch que aprovechan capacidades de CPU/GPU, garantizando eficiencia en entrenamiento y predicción.
2. Metodología de análisis integral: Se integraron momentos estadísticos (media, varianza, asimetría y curtosis), visualizaciones comparativas y medidas de divergencia (Jensen–Shannon), lo cual permitió caracterizar las diferencias entre tráfico normal y ataques backdoor antes de entrenar el modelo.
3. Validación exhaustiva: Se evaluó el modelo con múltiples particiones aleatorias (10-fold), métricas complementarias (accuracy, F1-score, Gini) y análisis de tiempos de ejecución, asegurando que los resultados fueran robustos y estables.
4. Propuesta de optimización: Se planteó un esquema híbrido que utiliza la distancia Jensen–Shannon como criterio de pre-filtrado de variables, reduciendo el espacio de búsqueda y mejorando la eficiencia del árbol sin sacrificar desempeño predictivo.

### 6.1. Hallazgos Principales

1. Separabilidad Excepcional: Los ataques backdoor y el tráfico normal presentaron separabilidad casi perfecta en el subconjunto analizado del KDD99. La característica `src_bytes` emergió como el discriminador principal ( $JS = 0.7886$ ), confirmando la utilidad de Jensen–Shannon para identificar variables altamente informativas y coherente con los resultados visualizados en los histogramas comparativos.
2. Eficacia del Modelo Simple: Un árbol de decisión con profundidad máxima 3 logró rendimiento prácticamente perfecto ( $Accuracy \approx 0.9997$  y  $F1 \approx 0.993$  en validación, y  $F1 = 1.0000$  en escenarios de prueba controlados). Esto evidencia que la complejidad adicional (profundidades mayores a 3) no aporta mejoras significativas, sino que únicamente incrementa el riesgo de sobreajuste y el costo computacional.
3. Interpretabilidad y Reglas Claras: El árbol resultante es altamente interpretable, con pocos nodos decisivos (`X[1]`, `X[9]`, `X[2]` y `X[31]`) que generan hojas casi puras ( $Gini \approx 0.00$ ). Las reglas derivadas permiten identificar los patrones que separan tráfico normal de ataques backdoor con transparencia, lo que facilita su aplicación en escenarios de seguridad en producción.

4. Consistencia entre análisis estadístico y modelo: Las variables con mayor divergencia Jensen–Shannon fueron precisamente las que aparecieron en los niveles superiores del árbol, mostrando una fuerte coherencia entre el análisis previo de distribución de datos y el modelo predictivo final.