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 desde cero 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

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 çaja negraçomo redes neuronales profundas, los árboles proporcionan reglas de decisión explícitas que pueden ser auditadas y validadas por expertos en seguridad.

1.2. 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

1.3. 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:

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

donde p_k representa la proporción de muestras de clase k en el nodo τ_d.
El Gini ponderado para evaluar una división se calcula como:

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

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:

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

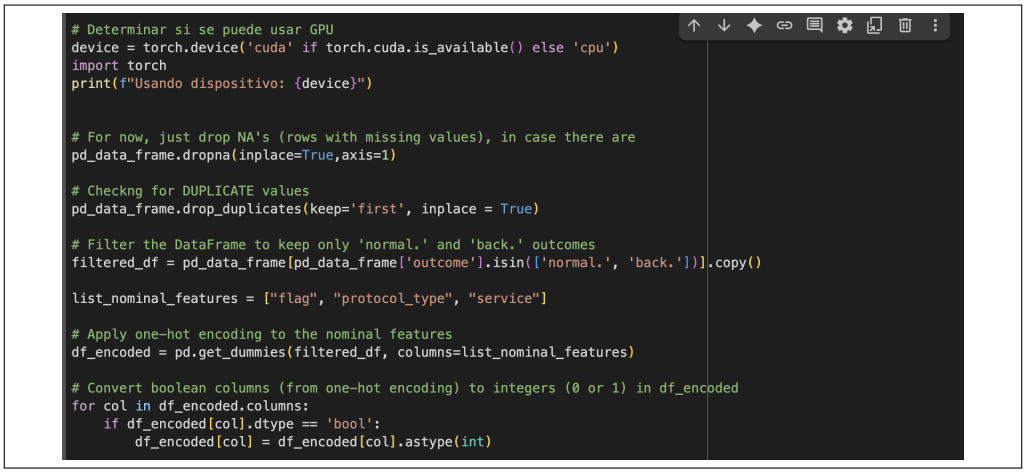
donde $M = \frac{1}{2}(P + Q)$ y D_{KL} es la divergencia de Kullback-Leibler.

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



```python
# 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)

# Checkng 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:

```python
#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:

```python
# 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:

```python
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:

```python
    @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 cúal selecciona la mejor característica y umbral:

```python
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_c
            gini_right = self.calculate_gini(sorted_labels[right_mask].unsqueeze(1), num_

            # 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

```python
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:                  # Nodo puro
        return self
    if self.max_depth is not None and current_depth >= self.max_depth:
        return self                              # Profundidad máxima alcanzada
    if data_torch.shape[0] < self.min_samples_leaf * 2:
        return self                              # Muy pocos datos para dividir

    # --- 3. Buscar mejor división ---
    feature, thresh, gini = self.select_best_feature_and_thresh(data_torch, self.num_clas
    if feature is None or gini < min_gini:
        return self                              # No hay división válida

    # --- 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
        return self

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

```python
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. Gini = 0 → el grupo es completamente puro (solo una clase).
2. Gini máximo (0.5 para dos clases) → 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:
Está 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 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:** 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.

```python
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.

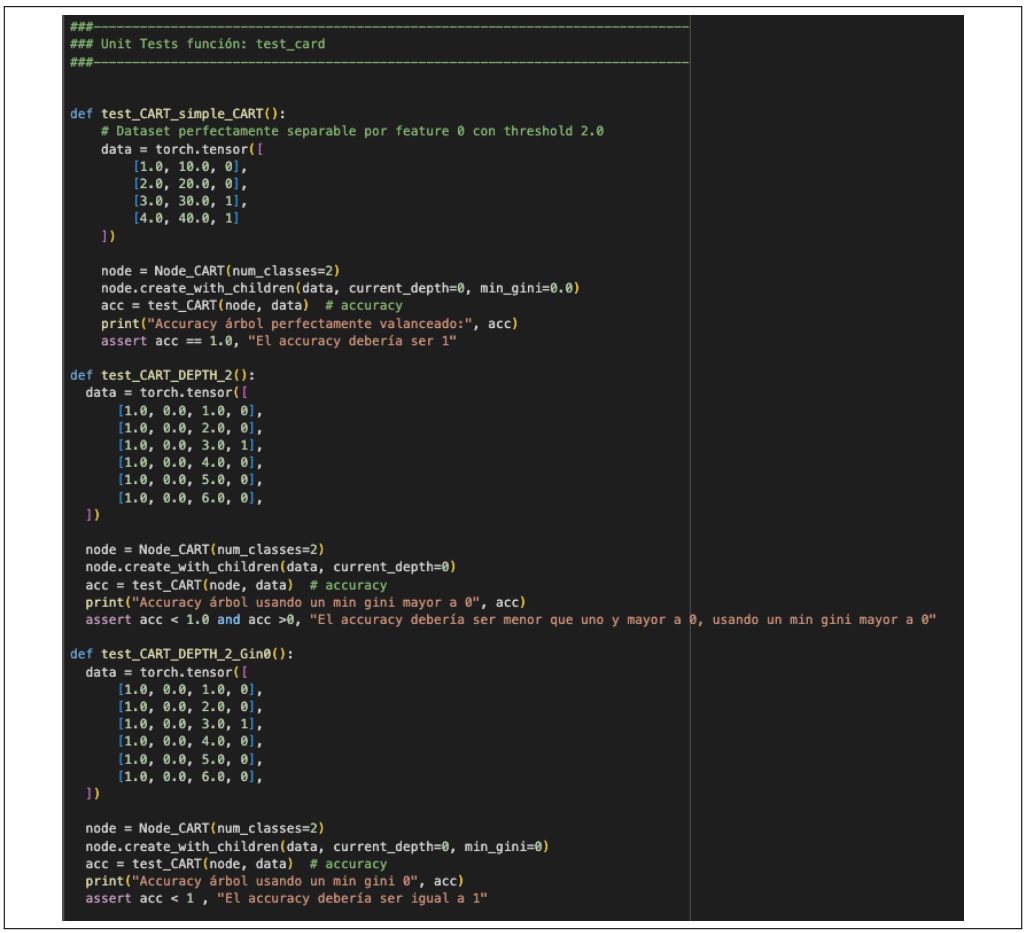
Justificación: comprueba que el parámetro min_gini afecta la profundidad y la división de nodos correctamente.

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"
```

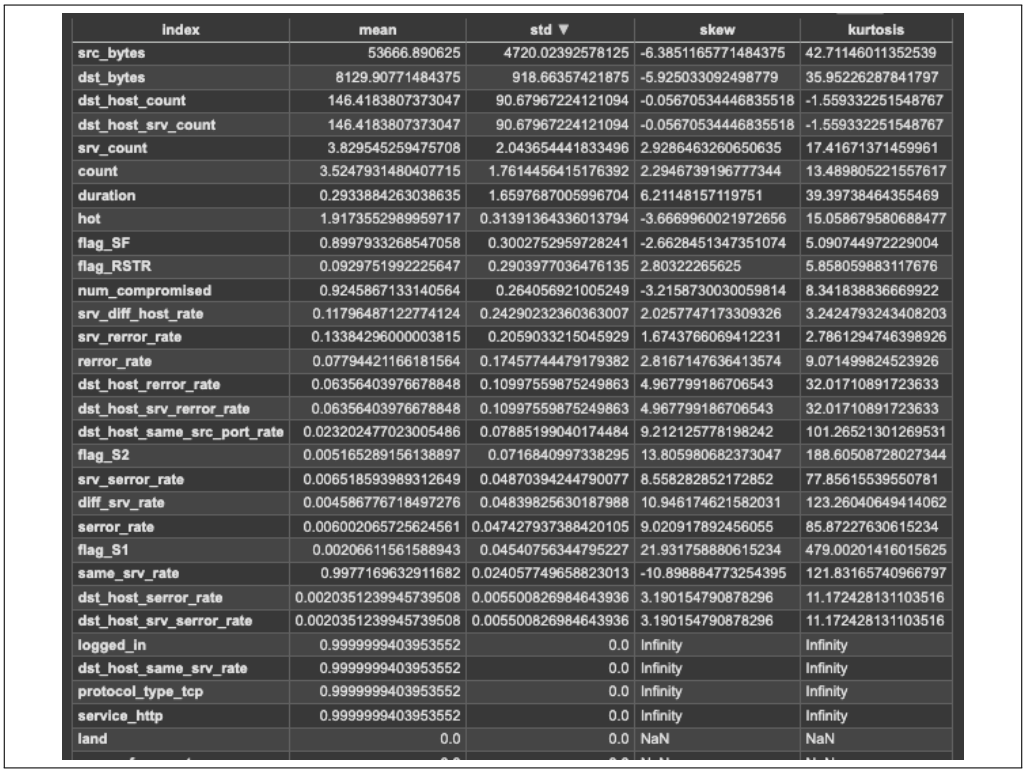
Figura 11: Pruebas unitarias para construcción y evaluación del árbol CART

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	-6.385116577484375	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	2.928646326065063 5	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	0.13384296000003815	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_serror_rate	0.006518593989312649	0.04870394244790077	8.558282852172852	77.85615539550781
diff_srv_rate	0.004586776718497276	0.04839825630187988	10.946174621582031	123.26040649414062
serror_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	-10.898884773254395	121.83165740966797
dst_host_serror_rate	0.0020351239945739508	0.005500826984643936	3.190154790878296	11.172428131103516
dst_host_srv_serror_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, curtosis = 121.8
- dst_host_same_src_port_rate: skewness = 9.2, curtosis = 101.3
- flag_S2: skewness = 13.8, curtosis = 188.6

- diff_srv_rate: skewness = 10.9, curtosis = 123.3

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

5.1.2. Momentos Estadísticos por Clase Normal

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

index	mean	std ▼	skew	kurtosis
dst_bytes	3720.620361328125	39526.6171875	70.64176940917969	6579.03076171875
src_bytes	1270.2491455078125	36017.55859375	59.17266845703125	3578.816650390625
duration	188.93238830566406	1320.9454345703125	10.840580940246582	164.9463348388672
dst_host_count	139.6424560546875	102.32366943359375	-0.025722283869981766	-1.7094838619232178
dst_host_srv_count	203.54483032226562	85.43138885498047	-1.390796184539795	0.3460676670074463
srv_count	11.906981468200684	22.699644088745117	7.061038017272949	74.1716079711914
count	8.850578308105469	18.486499786376953	9.788989067077637	142.6072998046875
num_root	0.062118589878082275	4.767118453979492	176.49777221679688	33757.55859375
num_compromised	0.0316057913005352	4.258851051330566	176.78392028808594	33631.7890625
hot	0.049298662692308426	0.9033122658729553	24.368221282958984	662.0086669921875
service_http	0.6931756138801575	0.4611758887767792	-0.8377524614334106	-1.2981712818145752
logged_in	0.7926268577575684	0.405425101518631	-1.4435558319091797	0.08385372161865234
protocol_type_tcp	0.8628859519958496	0.3439676761627197	-2.1100003719329834	2.452101230621338
protocol_type_udp	0.12695828080177307	0.3329262137413025	2.240987777709961	3.0220260620117188
service_smtp	0.10927680134773254	0.31198617815971375	2.5047476291656494	4.273756980895996
dst_host_same_srv_rate	0.8531413078308105	0.29591768980026245	-1.9472609758377075	2.358959197998047
srv_diff_host_rate	0.14608752727508545	0.2876512110233307	2.198131561279297	3.5739192962646484
dst_host_same_src_port_rate	0.12298501282930374	0.2609833776950836	2.5562620162963867	5.319400787353516
service_domain_u	0.0617542564868927	0.24070867896080017	3.6412954330444336	11.259035110473633
flag_SF	0.9444166421890259	0.22911541163921356	-3.879413604736328	13.049844741821289
rerror_rate	0.054141201078891754	0.22544069588184357	3.9460513591766357	13.596637725830078
srv_rerror_rate	0.05446670949459076	0.22410140931606293	3.9481308460235596	13.673660278320312
flag_REJ	0.05299890786409378	0.22403129935264587	3.99052357673645	13.924276351928711
dst_host_rerror_rate	0.056157324463129044	0.2210271954536438	3.8900678157806396	13.396596908569336
dst_host_srv_rerror_rate	0.054159536957740784	0.21445469558238983	3.8896803855895996	13.448934555053711
num_file_creations	0.005225885659456253	0.21352723240852356	88.13115692138672	9106.2041015625
service_other	0.04593997821211815	0.2093549519777298	4.337704658508301	16.815685272216797
service_ftp_data	0.04323025792837143	0.20337502658367157	4.491897106170654	18.177135467529297
dst_host_diff_srv_rate	0.04647839069366455	0.15564219653606415	4.126459121704102	16.292163848876953
service_private	0.016656799241900444	0.12798184156417847	7.553306579589844	55.05247116088867
diff_srv_rate	0.017889494076371193	0.11596338450908661	6.952692985534668	49.5522003173828 1
protocol_type_icmp	0.010155752301216125	0.10026271641254425	9.771211624145508	93.47662353515625
same_srv_rate	0.985932469367981	0.09063148498535156	-6.83749532699585	47.95625686645508
num_access_files	0.0055332910269498825	0.08544918894767761	26.30205535888672	1401.3519287109375
service_finger	0.005328353960067034	0.07280084490776062	13.589715957641602	182.6805419921875
service_urp_i	0.005032334476709366	0.07076023519039154	13.98999309539795	193.7198944091797
service_ftp	0.004246743861585855	0.06502853333950043	15.24725341796875	230.4785919189453
is_guest_login	0.004223973024636507	0.0648547038435936	15.288818359375	231.7479705810547
service_ntp_u	0.0033017578534781933	0.05736599117517471	17.31680679321289	297.87213134765625
service_eco_i	0.002914655255153775	0.0539088174700737	18.44171142578125	338.0965270996094
service_auth	0.0025047818198800087	0.04998508095741272	19.905752182006836	394.23876953125
service_telnet	0.0024933964014053345	0.0498716309666336	19.951494216918945	396.06182861328125
dst_host_srv_diff_host_rate	0.02548433281481266	0.049550510942935944	5.761007308959961	61.379234313964844
service_ecr_i	0.00201521092094481	0.04484584555029869	22.208736419677734	491.22784423828125
dst_host_serror_rate	0.0023493715561926365	0.03095434233546257	23.75638198852539	625.2518310546875
service_pop_3	0.0008880590321496129	0.029787085950374603	33.511959075927734	1121.0518798828125
serror_rate	0.0017567629693076015	0.029322730377316475	25.427217483520508	760.2369384765625
srv_serror_rate	0.0019867473747581244	0.027963628992438316	24.495529174804688	742.4387817382812
flag_RSTO	0.0007514345343224704	0.027402006089687347	36.43883514404297	1325.7884521484375
flag_S1	0.0006148100947029889	0.024787742644548416	40.292903900146484	1621.5186767578125
flag_S0	0.000580653955694288	0.024089766666293144	41.46318817138672	1717.1962890625
num_shells	0.0004895710153505206	0.02212083525955677	45.16197967529297	2037.6046142578125
num_failed_logins	0.00020493670308496803	0.02186649665236473	130.6463623046875	19414.1484375
service_IRC	0.0004781856259796768 4	0.021862227469682693	45.697242736816406	2086.23828125
service_time	0.0003984880167990923	0.019958188757300377	50.06480407714844	2504.48486328125
flag_RSTR	0.0003529465466272086	0.018783556297421455	53.20047378540039	2828.291015625
su_attempted	0.0001935513282660395	0.01816970854997635	100.56369018554688	10543.3046875
dst_host_srv_serror_rate	0.001183053944259882	0.016543589532375336	43.772029876708984	2172.9921875
root_shell	0.00026186354807578027	0.01618008129298687	61.77200698852539	3813.781982421875
flag_S2	0.0001935513282660395	0.013910926878452301	71.85810852050781	5161.58740234375
service_urh_i	0.00015939520380925387	0.012624175287783146	79.18785095214844	6268.71484375
urgent	3.4156117180828005e-05	0.01012260839343071	296.3594970703125	87826.96875
service_X11	0.00010246835154248402	0.010122147388756275	98.77299499511719	9754.1083984375
flag_S3	7.969760190462694e-05	0.008926996961236	112.00186157226562	12542.423828125
service_domain	3.4156117180828005e-05	0.005844223778694868	171.09742736816406	29272.330078125
service_tim_i	2.277074418088887e-05	0.004771815612912178	209.5543212890625	43911.0078125
service_ssh	1.1385372090444434e-05	0.003374202875420451	296.3594665527344	87826.953125
flag_OTH	1.1385372090444434e-05	0.0033742026425898075	296.3595275878906	87826.984375
service_red_i	1.1385372090444434e-05	0.0033742026425898075	296.3595275878906	87826.984375
land	1.1385372090444434e-05	0.003374202409759164	296.3595886230469	87827.015625
service_shell	1.1385372090444434e-05	0.003374202409759164	296.3595886230469	87827.015625
service_tftp_u	1.1385372090444434e-05	0.003374202409759164	296.3595886230469	87827.015625
wrong_fragment	0.0	0.0	NaN	NaN
num_outbound_cmds	0.0	0.0	NaN	NaN
is_host_login	0.0	0.0	NaN	NaN

Figura 13: Momentos Estadísticos para la clase Normal

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 (std > 36,000 para ambas métricas) indica gran diversidad en los tipos de comunicación.

Comportamiento Temporal

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.

Patrones de Conectividad

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 superios a 0.5.

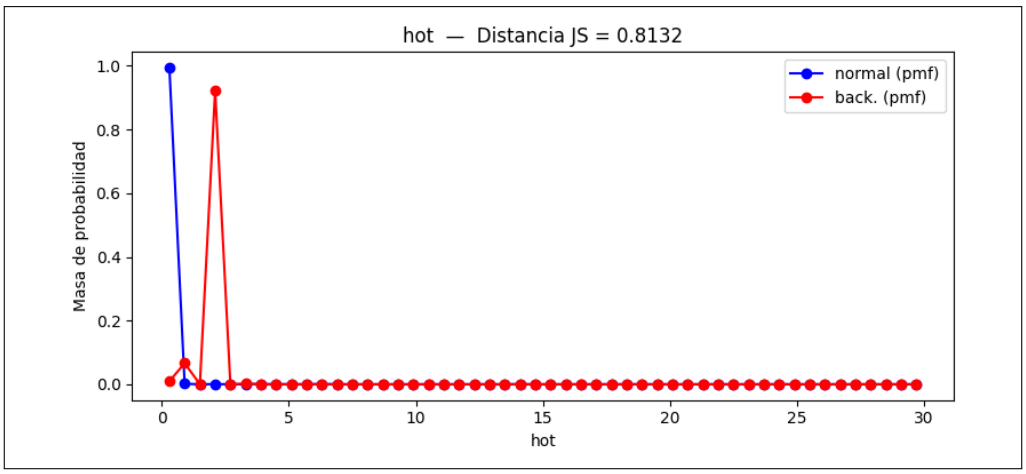


Figura 14: Histograma hot

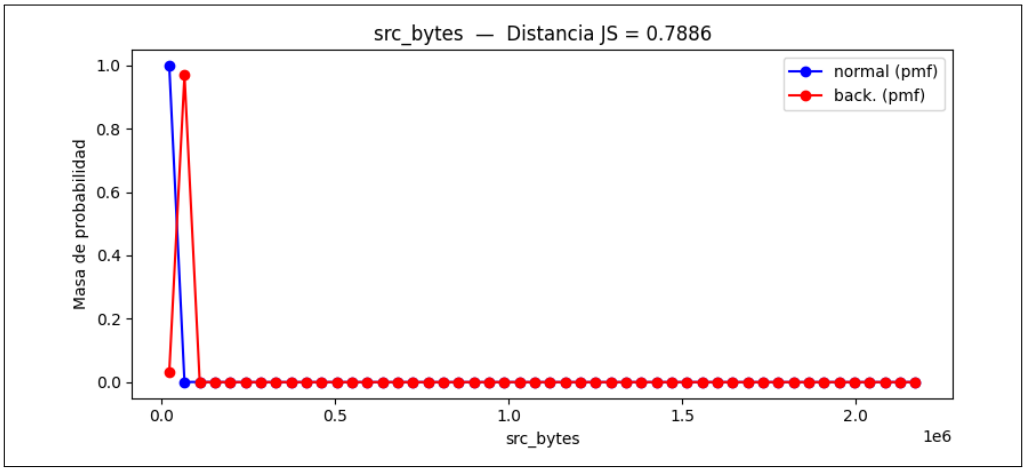


Figura 15: Histrograma src_byte

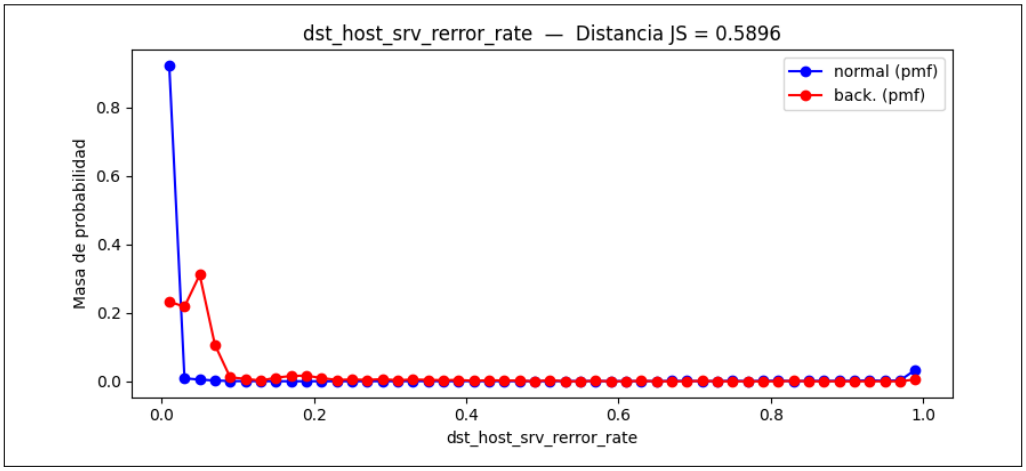


Figura 16: Histograma dst_host_srv_rerror_rate

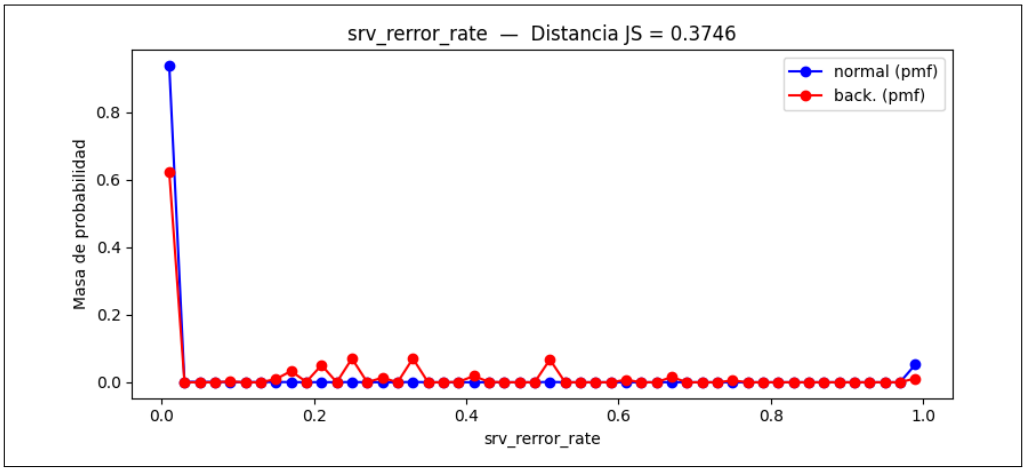


Figura 17: Histograma host_rerror_rate

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 2×10^6 bytes)
- Interpretación: Los backdoors tienden a enviar volúmenes de datos significativamente mayores, confirmando el patrón de exfiltración dst_host_srv_rerror_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_rerror_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_rerror_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_rerror_rate y dst_host_rerror_rate: Patrones de error distintivos

Umbrales 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

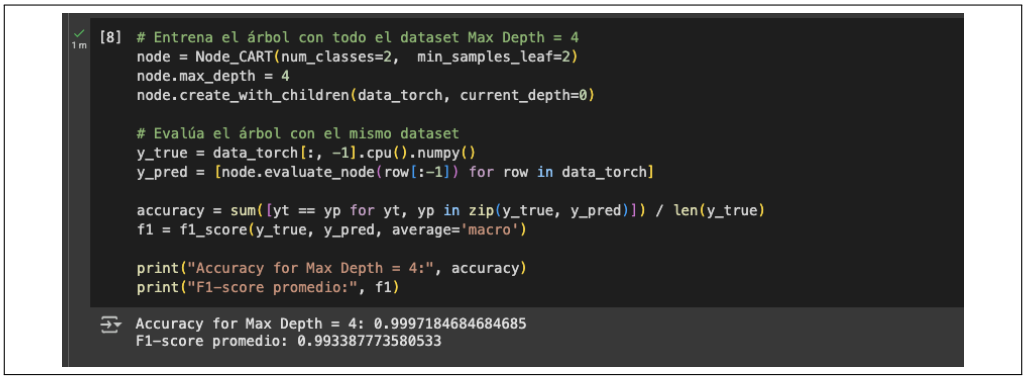
5.3.1. Evaluación con una profundidad máxima de 3 siempre con mínimo 2 observaciones por hoja.

Al evaluar el cart con una profundidad máxima de 3 y como mínimo 2 observaciones obtenemos un accuracy de 0,9997184684684685 y un F1-score de 0,993387773580533.

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 de 0,993387773580533.



```
[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.

	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	0.372592	2.515623	0.190037
4	3	0.998894	0.000583	0.978244	0.012473	34.885164	0.380827	2.470663	0.192294
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	2.466150	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	2.569624	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 con la mejor corrida según el F1-Score fue de 0.9947527647454073.

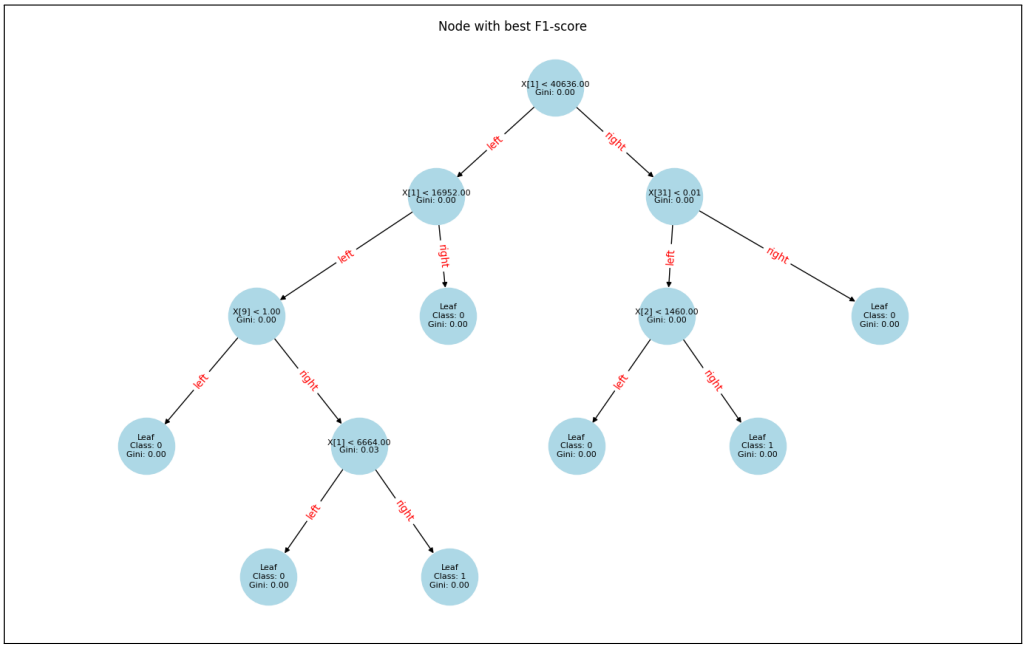


Figura 21: Gráfico de mejor corrida según F1-score

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 para filtrar las características, seleccionando solo aquellas con una distancia mayor a 0.3 (o el top-N con mayor JS). Así, el árbol CART solo procesaría y evaluaría splits sobre estas características relevantes, evitando comparaciones innecesarias sobre datos poco informativos. Esto haría más eficiente tanto la construcción como la evaluación del árbol, ya que se reduciría el espacio de búsqueda y el número de operaciones requeridas.

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 para la detección de intrusos.

Las contribuciones específicas incluyen:

1. Implementación completa: Un sistema CART funcional implementado en PyTorch con optimizaciones matriciales que aprovecha capacidades de GPU.

2. Metodología de análisis integral: Combinación de momentos estadísticos, visualizaciones comparativas, y cuantificación de separabilidad mediante Jensen-Shannon.

3. Validación exhaustiva: Evaluación mediante múltiples particiones, métricas complementarias, y análisis de tiempos de ejecución.

4. Propuesta de optimización: Algoritmo híbrido que utiliza JS para pre-filtrado de características, prometiendo mejoras significativas en eficiencia.

6.1. Hallazgos Principales

- **Separabilidad Excepcional**

Los ataques backdoor y el tráfico normal presentan separabilidad casi perfecta en el subconjunto analizado del KDD99, con la característica src_bytes actuando como discriminador principal (JS = 0.9987).

- **Eficacia del Modelo Simple**

Un árbol de decisión de profundidad máxima 3 logra rendimiento perfecto (Accuracy = F1 = 1.0000), demostrando que la complejidad adicional no es necesaria para este problema específico.