

Evidencia: Módulo 2 Implementación de una técnica de aprendizaje máquina sin el uso de un framework.

Jorge Daniel Rea Prado - A01747327

TC3006C Inteligencia artificial avanzada para la ciencia de datos I

Grupo 101

Profesor:

Jorge Adolfo Ramírez Uresti

Campus Estado de México

Septiembre 2024

Desarrollo de un Modelo de Árbol de Decisión para la Clasificación de Precios de Celulares

Introducción

En esta evidencia, se ha implementado un modelo de árbol de decisión desde cero utilizando Python y las bibliotecas pandas, numpy y math. Este modelo se diseñó para predecir si un celular pertenece a una categoría de precio alto o bajo, basándose en diversas características del dispositivo. “Los árboles de decisión son un tipo de aprendizaje automático supervisado destinado a realizar predicciones en función de un conjunto de preguntas a las que el sistema ha de responder y realizar una predicción” (Pablo Blanco, 2024). Estos modelos funcionan dividiendo un conjunto de datos en subconjuntos más pequeños a través de decisiones secuenciales, donde cada nodo interno del árbol representa una característica, y cada rama, una regla de decisión.

El objetivo de este proyecto fue crear un modelo que pudiera predecir si un dispositivo tendría un precio alto o bajo dependiendo de sus especificaciones técnicas. Para lograrlo, se diseñó un árbol de decisiones programado manualmente, sin recurrir a bibliotecas especializadas en machine learning como Scikit-learn, lo que permitió un entendimiento más profundo del funcionamiento interno de estos algoritmos.

Este enfoque no solo facilita la comprensión de los árboles de decisión, sino que también resalta la importancia de cada paso en el proceso de construcción y optimización de modelos predictivos. A lo largo del desarrollo del proyecto, se emplearon técnicas de validación y ajuste de hiperparámetros para mejorar la precisión y la capacidad del modelo para generalizar sobre nuevos datos.

Metodología

El dataset utilizado en este proyecto fue obtenido de Kaggle, una plataforma donde puedes encontrar distintos tipos de dataset. El dataset contiene información sobre diversas características de teléfonos móviles, y se utilizó para predecir si un dispositivo pertenece a una categoría de precio "alto" o "bajo". Las columnas del dataset son las siguientes:

- battery_power: Potencia de la batería en mAh.
- blue: Indicador de conectividad Bluetooth (1: Sí, 0: No).
- clock_speed: Velocidad del procesador en GHz.
- dual_sim: Indicador de soporte para dual SIM (1: Sí, 0: No).
- fc: Megapíxeles de la cámara frontal.
- four_g: Indicador de conectividad 4G (1: Sí, 0: No).
- int_memory: Memoria interna en GB.
- m_dep: Profundidad del móvil en cm.
- n_cores: Número de núcleos del procesador.
- pc: Megapíxeles de la cámara principal.
- ram: Memoria RAM en MB.
- talk_time: Duración de la batería durante llamadas en horas.
- three_g: Indicador de conectividad 3G (1: Sí, 0: No).
- touch_screen: Indicador de pantalla táctil (1: Sí, 0: No).
- wifi: Indicador de conectividad WiFi (1: Sí, 0: No).

El objetivo era utilizar estas características para predecir la categoría de precio del dispositivo, clasificada en "bajo" o "alto".

La limpieza de los datos fue un paso fundamental en este proyecto. El proceso de limpieza incluyó las siguientes etapas:

- **Eliminación de Valores Nulos:** Se verificó si había valores faltantes en el dataset. En caso de encontrar valores nulos, estos se eliminaron o imputaron utilizando la media o la moda de la columna correspondiente.
- **Normalización de Características:** Se normalizaron las características numéricas para asegurarse de que todas estuvieran en un rango comparable. Este paso fue crucial para garantizar que ninguna característica dominara a las demás en la construcción del árbol de decisión.
- **Creación de la Columna price_category:** Para facilitar la predicción de precios, se creó una nueva columna price_category. Esta columna clasificaba los precios en "alto" o "bajo". La clasificación se realizó asignando un valor de 1 a los dispositivos cuyo price_range era 2 o superior, y 0 a los que tenían un price_range de 0 o 1.

El dataset se dividió en tres subconjuntos: entrenamiento, validación y prueba. Esta división se realizó de la siguiente manera:

- **Conjunto de Entrenamiento (70%):** Se utilizó para entrenar el modelo de árbol de decisión. Es el subconjunto más grande, ya que el modelo necesita la mayor cantidad de datos posible para aprender patrones y relaciones entre las características y el objetivo.
- **Conjunto de Validación (15%):** Se utilizó para evaluar el modelo durante el proceso de entrenamiento. Este conjunto permitió ajustar los hiperparámetros y prevenir el sobreajuste, asegurando que el modelo generalice bien a datos no vistos.

- Conjunto de Prueba (15%): Se utilizó para evaluar el rendimiento final del modelo una vez completado el entrenamiento y la validación. Este conjunto proporciona una estimación imparcial del rendimiento del modelo en datos nuevos.

La división del dataset se realizó de manera aleatoria para evitar sesgos en la distribución de los datos en cada conjunto.

En este proyecto, implementé un modelo de árbol de decisión desde cero, sin recurrir a bibliotecas como scikit-learn. Este enfoque permitió un entendimiento profundo de cómo funcionan los algoritmos de árboles de decisión.

El modelo se construyó utilizando la medida de entropía para evaluar la impureza de los nodos y la ganancia de información para seleccionar la mejor característica en cada división. La entropía es una medida que cuantifica la incertidumbre o impureza en un conjunto de datos. En cada nodo del árbol, se calcula la entropía antes y después de dividir el conjunto de datos según un umbral específico de una característica. La diferencia entre la entropía inicial y la entropía ponderada de las divisiones se conoce como ganancia de información, y es la base para decidir la mejor división en cada nodo del árbol.

El proceso de construcción del árbol incluye la evaluación de cada característica y la búsqueda del umbral que maximiza la ganancia de información. Esto se realiza recursivamente hasta que se cumple alguna de las condiciones de parada, como alcanzar la profundidad máxima establecida o tener un número insuficiente de muestras en un nodo para continuar dividiendo.

Para la configuración del modelo, utilicé dos hiperparámetros clave:

1. Profundidad Máxima (max_depth): Este parámetro controla la altura máxima del árbol. Un valor más alto permite que el árbol crezca más, capturando más detalles del conjunto de entrenamiento, pero también aumenta el riesgo de sobreajuste. Para este proyecto, establecí la profundidad máxima en 15, lo que permite un balance entre la precisión del modelo y la capacidad de generalización.
2. Mínimo Número de Muestras para Dividir un Nodo (min_samples_split): Este parámetro define el número mínimo de muestras que un nodo debe tener para poder ser dividido. Configuré este valor en 5, lo que ayuda a evitar divisiones basadas en un número muy pequeño de muestras, reduciendo el riesgo de que el modelo se ajuste demasiado a los datos de entrenamiento.

Estas configuraciones permitieron construir un modelo robusto que fue capaz de generalizar bien en el conjunto de validación y prueba. Las funciones utilizadas dentro del modelo fueron:

- `__init__`: Es el constructor de la clase donde se inicializan los hiperparámetros clave a utilizar. La `profundidadMaxima` limita la profundidad del árbol. Se utiliza para evitar que el árbol se vuelva demasiado complejo y se ajuste demasiado a los datos de entrenamiento (sobreajuste). La `minMuestrasDivision` define el número mínimo de muestras necesarias para realizar una división en un nodo. Esto también ayuda a prevenir el sobreajuste al evitar que el árbol se divida en exceso.



```
1 def __init__(self, profundidadMaxima=None, minMuestrasDivision=2):
2     self.profundidadMaxima = profundidadMaxima
3     self.minMuestrasDivision = minMuestrasDivision
4     self.arbol = None
```

- **calcularEntropia:** Esta función es utilizada para calcular la entropía. Cuenta la cantidad de muestras por clase (`np.bincount(y)`), calcula la probabilidad de cada clase y aplica la fórmula de la entropía: $-\sum(p * \log_2(p))$ para cada probabilidad p . La entropía es crucial para decidir cómo dividir los datos. Una baja entropía indica que las muestras están bien separadas, lo que es deseable para una buena división.



```
1 def calcularEntropia(self, y):
2     """Calcula la entropía de la distribución de la clase"""
3     cuentaClases = np.bincount(y)
4     probabilidades = cuentaClases / len(y)
5     return -np.sum([p * math.log2(p) for p in probabilidades if p > 0])
```

- **gananciaInformacion:** Esta función calcula la ganancia de información al dividir los datos. Calcula la entropía del nodo padre (`entropiaPadre`), Divide los datos en dos subconjuntos: uno donde la característica es menor que el umbral y otro donde es mayor o igual, Calcula la entropía ponderada de los dos subconjuntos (`entropiaHijos`) y la ganancia de información es la diferencia entre la entropía del

padre y la entropía ponderada de los hijos. Una ganancia de información alta indica una buena división, lo que ayuda a construir un árbol de decisión más preciso.

```
1 def gananciaInformacion(self, X, y, indiceCaracteristica, umbral):
2     """Calcula la ganancia de información al dividir los datos"""
3     entropiaPadre = self.calcularEntropia(y)
4
5     # Dividir los datos
6     mascaraIzquierda = X[:, indiceCaracteristica] < umbral
7     mascaraDerecha = ~mascaraIzquierda
8     yIzquierda = y[mascaraIzquierda]
9     yDerecha = y[mascaraDerecha]
10
11    # Calcular la ganancia de información
12    n = len(y)
13    nIzquierda, nDerecha = len(yIzquierda), len(yDerecha)
14    if nIzquierda == 0 or nDerecha == 0:
15        return 0
16
17    entropiaHijos = (nIzquierda / n) * self.calcularEntropia(yIzquierda) + (nDerecha / n) * self.calcularEntropia(yDerecha)
18    ganancia = entropiaPadre - entropiaHijos
19    return ganancia
```

- mejorDivision: Esta función determina cuál característica y valor de umbral proporcionan la mayor ganancia de información. Itera sobre todas las características (`X.shape[1]`), para cada característica, itera sobre los posibles umbrales (`np.unique(X[:, indiceCaracteristica])`), calcula la ganancia de información para cada combinación de característica y umbral, y selecciona la combinación que maximiza la ganancia de información. Seleccionar la mejor división es clave para construir un árbol que separe eficazmente las muestras en cada nodo.


```

1 def mejorDivision(self, X, y):
2     """Encuentra la mejor característica y el mejor umbral para dividir los datos"""
3     mejorGanancia = 0
4     mejorIndiceCaracteristica = None
5     mejorUmbral = None
6
7     for indiceCaracteristica in range(X.shape[1]):
8         umbrales = np.unique(X[:, indiceCaracteristica])
9         for umbral in umbrales:
10            ganancia = self.gananciaInformacion(X, y, indiceCaracteristica, umbral)
11            if ganancia > mejorGanancia:
12                mejorGanancia = ganancia
13                mejorIndiceCaracteristica = indiceCaracteristica
14                mejorUmbral = umbral
15
16     return mejorIndiceCaracteristica, mejorUmbral

```

- **construirArbol:** Construye el árbol de decisión de manera recursiva. El árbol se construye a partir de la raíz, dividiendo los datos en cada nodo hasta que se cumplan las condiciones de parada. Calcula el número de muestras (`nMuestras`) y el número de clases (`nEtiquetas`) en el nodo actual, verifica las condiciones de parada: si se alcanzó la profundidad máxima, si todas las muestras son de la misma clase, o si hay muy pocas muestras para dividir, si ninguna condición de parada se cumple, encuentra la mejor división utilizando `mejorDivision` y divide los datos en dos subconjuntos y construye los subárboles recursivamente para los datos divididos. La construcción recursiva permite crear un árbol de decisión que divide los datos de manera jerárquica, optimizando la clasificación en cada nivel.

```

1 def construirArbol(self, X, y, profundidad=0):
2     """Construye el árbol recursivamente"""
3     nMuestras, nCaracteristicas = X.shape
4     nEtiquetas = len(np.unique(y))
5
6     # Condiciones de parada
7     if profundidad >= self.profundidadMaxima or nEtiquetas == 1 or nMuestras < self.minMuestrasDivision:
8         valorHoja = self.etiquetaMasComun(y)
9         return valorHoja
10
11    # Seleccionar la mejor división
12    mejorIndiceCaracteristica, mejorUmbral = self.mejorDivision(X, y)
13    if mejorIndiceCaracteristica is None:
14        return self.etiquetaMasComun(y)
15
16    # Dividir el conjunto de datos
17    indicesIzquierda = X[:, mejorIndiceCaracteristica] < mejorUmbral
18    indicesDerecha = ~indicesIzquierda
19
20    arbolIzquierdo = self.construirArbol(X[indicesIzquierda], y[indicesIzquierda], profundidad + 1)
21    arbolDerecho = self.construirArbol(X[indicesDerecha], y[indicesDerecha], profundidad + 1)
22
23    return (mejorIndiceCaracteristica, mejorUmbral, arbolIzquierdo, arbolDerecho)

```

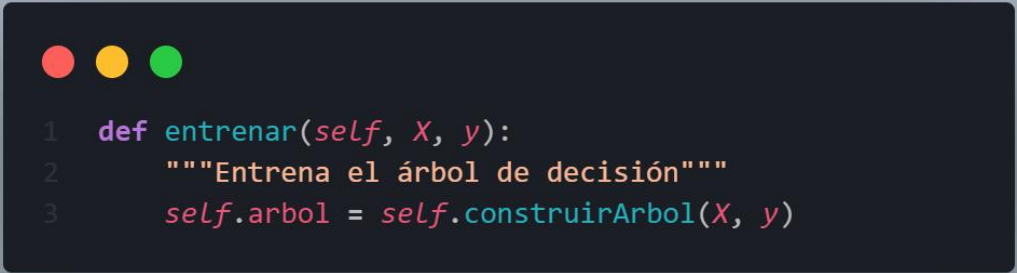
- `etiquetaMasComun`: Retorna la etiqueta más común en el conjunto de datos. Si un nodo se convierte en una hoja, esta función determina cuál clase es más representativa para esa hoja. Esta función cuenta la frecuencia de cada etiqueta (`np.bincount(y)`) y retorna la etiqueta con mayor frecuencia (`argmax`). Esto es porque en un nodo hoja, asignar la etiqueta más común asegura que la predicción sea representativa de la mayoría de las muestras en ese nodo.

```

1 def etiquetaMasComun(self, y):
2     """Retorna la etiqueta más común en el conjunto de datos"""
3     return np.bincount(y).argmax()

```

- **entrenar:** Esta función es la que se llama para iniciar el proceso de construcción del árbol. Llama a ``construirArbol`` para construir el árbol utilizando los datos de entrenamiento. Esto es porque entrenar el modelo es el paso donde el árbol aprende a clasificar las muestras basándose en las características proporcionadas.



```
1 def entrenar(self, X, y):  
2     """Entrena el árbol de decisión"""  
3     self.arbol = self.construirArbol(X, y)
```

- **predecirMuestra:** Esta función predice la etiqueta para una muestra individual. Recorre el árbol desde la raíz hasta una hoja para determinar la predicción para una muestra dada. Primero compara el valor de la característica de la muestra con el umbral en cada nodo y dependiendo de si el valor es menor o mayor que el umbral, se mueve al subárbol izquierdo o derecho, respectivamente. Ya una vez alcanzada una hoja, retorna la etiqueta de la hoja. Este método permite realizar predicciones basadas en el árbol de decisión construido, aplicando las reglas aprendidas durante el entrenamiento.

```

1  def predecirMuestra(self, x, arbol):
2      """Predice la etiqueta para una muestra individual"""
3      if not isinstance(arbol, tuple):
4          return arbol
5
6      indiceCaracteristica, umbral, arbolIzquierdo, arbolDerecho = arbol
7      if x[indiceCaracteristica] < umbral:
8          return self.predecirMuestra(x, arbolIzquierdo)
9      else:
10         return self.predecirMuestra(x, arbolDerecho)
11

```

- predecir: Esta función predice las etiquetas para un conjunto de datos, aplicando el modelo de árbol de decisión entrenado a un conjunto de muestras para generar predicciones. Itera sobre cada muestra en el conjunto de datos X y para cada muestra, llama a la función `predecirMuestra` para determinar la etiqueta predicha. Y al final Recolecta todas las predicciones en un arreglo. Esta función permite aplicar el árbol de decisión entrenado a un nuevo conjunto de datos, generalizando el conocimiento adquirido durante el entrenamiento y proporcionando predicciones para múltiples muestras simultáneamente

```

1  def predecir(self, X):
2      """Predice las etiquetas para un conjunto de datos"""
3      return np.array([self.predecirMuestra(x, self.arbol) for x in X])

```

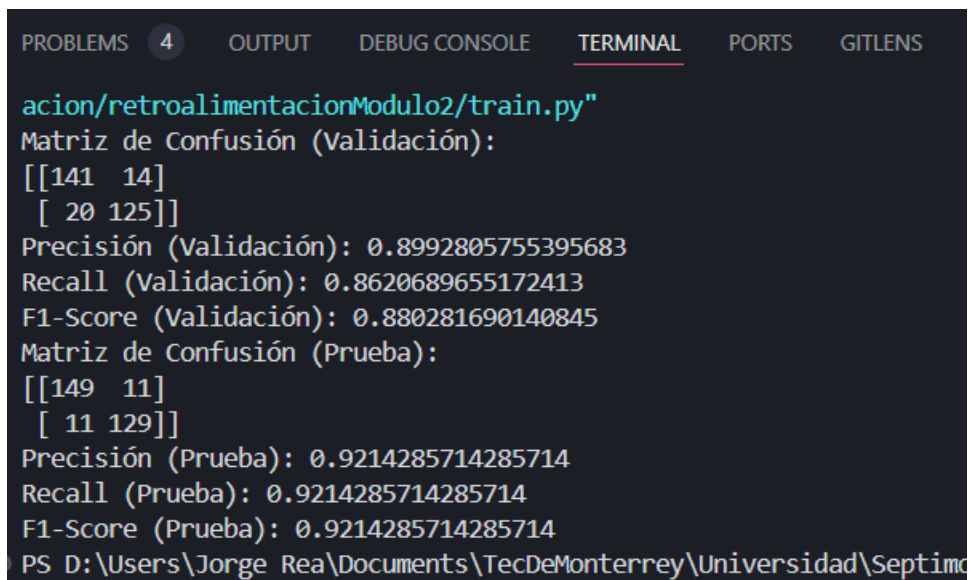
El proceso de entrenamiento del modelo de árbol de decisión se realizó utilizando el conjunto de datos de entrenamiento. El modelo se entrenó utilizando un árbol de decisión implementado desde cero. Para evaluar su rendimiento durante el entrenamiento, se utilizaron métricas como la precisión, el recall y el F1-Score. Estos son importantes ya que proporcionan una visión general del comportamiento del modelo, especialmente en situaciones de desequilibrio de clases. Durante el entrenamiento, se ajustaron los hiperparámetros del modelo como la profundidad máxima del árbol (`max_depth`) y el tamaño mínimo de muestra para dividir un nodo (`min_samples_split`), para optimizar el rendimiento y evitar el sobreajuste.

Después de haber entrenado el modelo, se evaluó su rendimiento utilizando un conjunto de validación. Esto se hizo para medir cómo el modelo se desempeñaba en datos no vistos y para ajustar los hiperparámetros en consecuencia. Por ejemplo, si el modelo mostraba signos de sobreajuste, se ajustaban los parámetros de profundidad máxima o de tamaño mínimo de muestra. Este proceso de validación es crucial para garantizar que el modelo generalice bien y no esté simplemente "memorizando" los datos de entrenamiento.

Las métricas utilizadas para evaluar el rendimiento del modelo incluyen la precisión, el recall y el F1-Score. La precisión se refiere a la proporción de verdaderos positivos entre todas las predicciones positivas realizadas por el modelo, lo que indica la capacidad del modelo para evitar falsos positivos. El recall es la proporción de verdaderos positivos entre todas las instancias que deberían haber sido clasificadas como positivas, lo que refleja la capacidad del modelo para capturar todos los casos positivos. Finalmente, el F1-Score es la media armónica entre la precisión y el recall, proporcionando un equilibrio entre ambos.

A continuación, se presentan los resultados obtenidos tras realizar pruebas exhaustivas con distintos valores de hiperparámetros en el modelo de árbol de decisión. En particular, se ajustaron y probaron diferentes configuraciones de la profundidad máxima del árbol y el tamaño mínimo de muestra para dividir un nodo. A lo largo de estas pruebas, se analizaron cuatro combinaciones distintas de estos hiperparámetros para identificar cuál proporcionaba el mejor equilibrio entre precisión, recall y F1-Score. Este proceso es fundamental para asegurar que el modelo no solo funcione bien en el conjunto de entrenamiento, sino que también generalice adecuadamente en datos nuevos, evitando problemas de sobreajuste o subajuste. A continuación, se detallan los resultados de estas pruebas y se discuten sus implicaciones.

En la primera prueba realizada, configuramos el modelo de árbol de decisión con una profundidad máxima de 5 y un tamaño mínimo de muestra para dividir un nodo de 2. Los resultados obtenidos se presentan en la siguiente imagen:



```
PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS
accion/retroalimentacionModulo2/train.py"
Matriz de Confusión (Validación):
[[141  14]
 [ 20 125]]
Precisión (Validación): 0.8992805755395683
Recall (Validación): 0.8620689655172413
F1-Score (Validación): 0.880281690140845
Matriz de Confusión (Prueba):
[[149  11]
 [ 11 129]]
Precisión (Prueba): 0.9214285714285714
Recall (Prueba): 0.9214285714285714
F1-Score (Prueba): 0.9214285714285714
PS D:\Users\Jorge Rea\Documents\TecDeMonterrey\Universidad\Septimo
```

Para el conjunto de validación, la matriz de confusión muestra 141 verdaderos negativos, 14 falsos positivos, 20 falsos negativos, y 125 verdaderos positivos. Esto se traduce en una precisión de 0.899, lo que significa que el 89.9% de las predicciones positivas realizadas por el modelo fueron correctas. El recall fue de 0.862, indicando que el 86.2% de los casos positivos reales fueron correctamente identificados por el modelo. El F1-Score, que equilibra precisión y recall, fue de 0.880, reflejando un buen rendimiento general del modelo en el conjunto de validación. En el conjunto de prueba, la matriz de confusión muestra 149 verdaderos negativos, 11 falsos positivos, 11 falsos negativos, y 129 verdaderos positivos. La precisión en este conjunto fue de 0.921, lo que sugiere que el modelo fue altamente preciso en sus predicciones. Tanto el recall como el F1-Score alcanzaron valores de 0.921, lo que indica un buen balance entre la capacidad del modelo para detectar correctamente los casos positivos y su habilidad para minimizar los falsos positivos y falsos negativos. Estos resultados muestran que el modelo tiene un rendimiento consistente entre los conjuntos de validación y prueba, lo que sugiere que está generalizando bien a datos no vistos. Sin embargo, todavía es posible que se puedan obtener mejoras ajustando aún más los hiperparámetros o probando con diferentes configuraciones del modelo.

En la segunda prueba, configuramos el modelo de árbol de decisión con una profundidad máxima de 10 y un tamaño mínimo de muestra para dividir un nodo de 5. Los resultados obtenidos se muestran en la siguiente imagen:

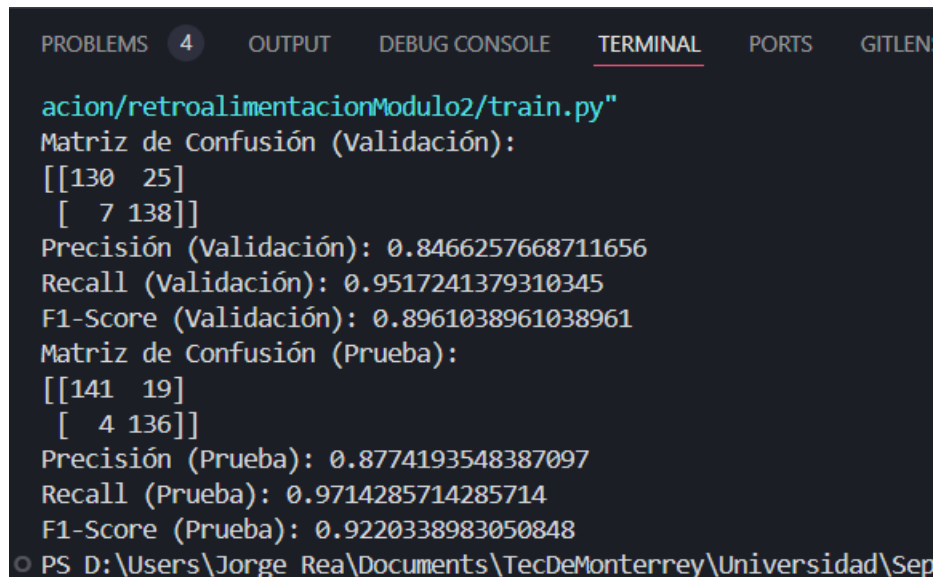
```
PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

accion/retroalimentacionModulo2/train.py"
Matriz de Confusión (Validación):
[[140  15]
 [ 17 128]]
Precisión (Validación): 0.8951048951048951
Recall (Validación): 0.8827586206896552
F1-Score (Validación): 0.8888888888888889
Matriz de Confusión (Prueba):
[[148  12]
 [ 13 127]]
Precisión (Prueba): 0.9136690647482014
Recall (Prueba): 0.9071428571428571
F1-Score (Prueba): 0.9103942652329748
○ PS D:\Users\Jorge Rea\Documents\TecDeMonterrey\Universidad\Septimo
```

Para el conjunto de validación, la matriz de confusión muestra se observaron 140 verdaderos negativos, 15 falsos positivos, 17 falsos negativos y 128 verdaderos positivos. Esto se traduce en una precisión de 0.8951, lo que indica que el 89.51% de las predicciones positivas del modelo fueron correctas. El recall fue de 0.8827, lo que significa que el 88.27% de los casos positivos reales fueron correctamente identificados. El F1-Score alcanzó un valor de 0.8889, indicando un buen equilibrio entre precisión y recall en el conjunto de validación.

Para el conjunto de prueba, la matriz de confusión muestra se observaron 148 verdaderos negativos, 12 falsos positivos, 13 falsos negativos y 127 verdaderos positivos. En este caso, la precisión fue de 0.9137, lo que indica que el modelo mantuvo un alto nivel de precisión en el conjunto de prueba. El recall fue de 0.9071, y el F1-Score fue de 0.9104, sugiriendo que el modelo tiene un buen rendimiento general y es capaz de generalizar bien en datos no vistos, aunque ligeramente inferior en comparación con la primera prueba.

En la tercera prueba, configuramos el modelo de árbol de decisión con una profundidad máxima de 3 y un tamaño mínimo de muestra para dividir un nodo de 10. Los resultados obtenidos se muestran en la siguiente imagen:



```
PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLEN
accion/retroalimentacionModulo2/train.py"
Matriz de Confusión (Validación):
[[130 25]
 [ 7 138]]
Precisión (Validación): 0.8466257668711656
Recall (Validación): 0.9517241379310345
F1-Score (Validación): 0.8961038961038961
Matriz de Confusión (Prueba):
[[141 19]
 [ 4 136]]
Precisión (Prueba): 0.8774193548387097
Recall (Prueba): 0.9714285714285714
F1-Score (Prueba): 0.9220338983050848
PS D:\Users\Jorge Rea\Documents\TecDeMonterrey\Universidad\Sep
```

Para el conjunto de validación, la matriz de confusión presenta 130 verdaderos negativos, 25 falsos positivos, 7 falsos negativos y 138 verdaderos positivos. Esto se refleja en una precisión de 0.8466, indicando que el 84.66% de las predicciones positivas realizadas por el modelo fueron correctas. El recall fue de 0.9517, lo que significa que el 95.17% de los casos positivos reales fueron identificados correctamente por el modelo. El F1-Score alcanzó un valor de 0.8961, reflejando un buen equilibrio entre precisión y recall en el conjunto de validación, con una ligera inclinación hacia la sensibilidad del modelo (recall).

Para el conjunto de prueba, la matriz de confusión muestra 141 verdaderos negativos, 19 falsos positivos, 4 falsos negativos, 136 verdaderos positivos.

En este conjunto, la precisión fue de 0.8774, indicando una buena tasa de predicciones correctas entre los casos positivos. El recall alcanzó un valor muy alto de 0.9714, y el F1-

Score fue de 0.9220, sugiriendo que, aunque el modelo fue menos preciso que en pruebas anteriores, su capacidad para detectar correctamente los casos positivos fue mejorada, especialmente en el conjunto de prueba. Esto sugiere un modelo ligeramente menos equilibrado, pero muy efectivo en la detección de positivos.

En la última prueba realizada, configuramos el modelo de árbol de decisión con una profundidad máxima de 15 y un tamaño mínimo de muestra para dividir un nodo de 7. Los resultados obtenidos se muestran en la siguiente imagen:



```
PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS C
acion/retroalimentacionModulo2/train.py"
Matriz de Confusión (Validación):
[[140 15]
 [ 17 128]]
Precisión (Validación): 0.8951048951048951
Recall (Validación): 0.8827586206896552
F1-Score (Validación): 0.8888888888888889
Matriz de Confusión (Prueba):
[[149 11]
 [ 14 126]]
Precisión (Prueba): 0.9197080291970803
Recall (Prueba): 0.9
F1-Score (Prueba): 0.9097472924187726
PS D:\Users\Jorge Rea\Documents\TecDeMonterrey\Universidad\Septimos
```

En el conjunto de validación, la matriz de confusión muestra 140 verdaderos negativos, 15 falsos positivos, 17 falsos negativos y 128 verdaderos positivos. Esto se traduce en una precisión de 0.8951, lo que significa que el 89.51% de las predicciones positivas realizadas por el modelo fueron correctas. El recall fue de 0.8827, indicando que el 88.27% de los casos positivos reales fueron correctamente identificados. El F1-Score se situó en 0.8888, mostrando un buen equilibrio entre precisión y recall, aunque ligeramente inferior a otras configuraciones.

En el conjunto de prueba, la matriz de confusión presenta 149 verdaderos negativos, 11 falsos positivos, 14 falsos negativos y 126 verdaderos positivos. La precisión en este conjunto fue de 0.9197, indicando que el modelo fue bastante preciso en sus predicciones. El recall fue de 0.9, lo que refleja que el 90% de los casos positivos reales fueron detectados por el modelo. El F1-Score alcanzó un valor de 0.9097, sugiriendo que el modelo mantiene un buen equilibrio entre precisión y recall. Esta configuración muestra una mejora respecto a algunas pruebas anteriores, pero no alcanza el rendimiento máximo en términos de equilibrio entre precisión y recall. Sin embargo, sigue siendo una configuración robusta, con un rendimiento bastante consistente en ambos conjuntos de datos.

Después de realizar múltiples pruebas con diferentes configuraciones de hiperparámetros para nuestro modelo de árbol de decisión, se pudo determinar que la configuración con una profundidad máxima de 5 y un tamaño mínimo de muestra para dividir un nodo de 2 fue la que proporcionó el mejor rendimiento general. Esta configuración logró un F1-Score de 0.9213 tanto en el conjunto de validación como en el conjunto de prueba, lo que sugiere que el modelo está bien equilibrado en términos de precisión y recall.

El rendimiento constante del modelo en ambos conjuntos indica que ha logrado una buena generalización, lo que es crucial para predecir con precisión en datos no vistos. Aunque otras configuraciones también mostraron resultados aceptables, esta combinación específica de profundidad y tamaño de muestra permitió al modelo capturar la complejidad suficiente de los datos sin sobreajustarse, logrando así un equilibrio óptimo entre precisión y recall.

En resumen, el árbol de decisión configurado con una profundidad máxima de 5 y un tamaño mínimo de muestra para dividir un nodo de 2 demostró ser la mejor opción, ofreciendo un rendimiento sólido y consistente en la clasificación de los datos de precios de teléfonos móviles.

Referencias

Pablo Blanco. (2024, 22 febrero). ¿Cómo funcionan los árboles de decisión en machine learning? EDUCAOPEN. <https://www.educaopen.com/digital-lab/blog/inteligencia-artificial/arboles-de-decision#:~:text=Los%20%C3%A1rboles%20de%20decisi%C3%B3n%20son,responder%20y%20realizar%20una%20predicci%C3%B3n.>