Instituto Tecnológico de Costa Rica - TEC

Inteligencia Artificial

Docente: Kenneth Obando Rodríguez

Trabajo Corto 3: Árboles de Decisión

Estudiantes:

- Renzo Giuliano Barra Mostajo
- Ana María Guevara Roselló
- Jonathan Alberto Guzmán Araya

Link del Cuaderno (recuerde configurar el acceso a público):

- Link de su respuesta

  Nota: Este trabajo tiene como objetivo promover la comprensión de la materia y su importancia en la elección de algoritmos. Los alumnos deben evitar copiar y pegar directamente información de fuentes externas, y en su lugar, demostrar su propio análisis y comprensión.

Entrega

Debe entregar un archivo comprimido por el TecDigital, incluyendo un documento pdf con los resultados de los experimentos y pruebas. La fecha de entrega es el domingo 17 de setiembre, antes de las 10:00pm.

Instrucciones:

Las alternativas se rifarán en clase utilizando números aleatorios. Deberá realizar la asignación propuesta. Si realiza ambos ejercicios, recibirá 20 puntos en **la nota porcentual de la actividad**, para aplicar a la totalidad de los puntos extra es necesario que ambas actividades se completen al 100%

Actividad - Taller

1. Cree una clase nodo con atributos necesarios para un árbol de decisión: feature, umbral, gini, cantidad_muestras, valor, izquierda, derecha

```
import numpy as np
from collections import Counter
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score


# Step 1: Node Class Creation
# This class represents a node in the decision tree.
class Node:
    def __init__(self, feature=None, threshold=None, impurity=None, sample_count=None, value=None, left=None, right=None):
        self.feature = feature
        self.threshold = threshold
        self.impurity = impurity
        self.sample_count = sample_count
        self.value = value
        self.left = left
        self.right = right
```

```
# Function to find the most common class in a list of labels
def most_common_class(y):
    class_counts = Counter(y)
    most_common = class_counts.most_common(1)[0][0]
    return most_common


# Function to select the best feature and threshold for splitting
def find_best_split(X, y, criterion='gini'):
    best_impurity = float('inf')
    best_feature = None
    best_threshold = None

    for feature in range(X.shape[1]):
        unique_thresholds = np.unique(X[:, feature])
        for threshold in unique_thresholds:
            left_indices = X[:, feature] <= threshold
            right_indices = X[:, feature] > threshold
```

```
            impurity = calculate_impurity(
                y[left_indices], y[right_indices], criterion)

            if impurity < best_impurity:
                best_impurity = impurity
                best_feature = feature
                best_threshold = threshold

    return best_feature, best_threshold


# Impurity Functions (Gini and Entropy)
# These functions calculate the impurity of a set of labels.
def calculate_impurity(y_left, y_right, criterion='gini'):
    if criterion == 'gini':
        impurity_left = gini_impurity(y_left)
        impurity_right = gini_impurity(y_right)
        total_samples = len(y_left) + len(y_right)
        weighted_impurity = (len(y_left) / total_samples) * impurity_left + \
            (len(y_right) / total_samples) * impurity_right
        return weighted_impurity
    elif criterion == 'entropy':
        entropy_left = entropy_impurity(y_left)
        entropy_right = entropy_impurity(y_right)
        total_samples = len(y_left) + len(y_right)
        weighted_entropy = (len(y_left) / total_samples) * entropy_left + \
            (len(y_right) / total_samples) * entropy_right
        return weighted_entropy
    else:
        raise ValueError(
            "Invalid criterion. Supported criteria are 'gini' and 'entropy'.")


# Function to calculate the entropy of a set of labels
def entropy_impurity(labels):
    num_samples = len(labels)
    if num_samples == 0:
        return 0.0

    class_counts = Counter(labels)
    impurity = 0.0
    for class_count in class_counts.values():
        class_probability = class_count / num_samples
        impurity -= class_probability * np.log2(class_probability)

    return impurity


# Function to calculate the Gini index of a set of labels
def gini_impurity(labels):
    num_samples = len(labels)
    if num_samples == 0:
        return 0.0
    class_counts = Counter(labels)
    impurity = 1.0
    for class_count in class_counts.values():
        class_probability = class_count / num_samples
        impurity -= class_probability ** 2

    return impurity


# Function to split the dataset into left and right subsets
def split_dataset(X, y, feature, threshold):
    left_indices = X[:, feature] <= threshold
    right_indices = X[:, feature] > threshold

    X_left = X[left_indices]
    y_left = y[left_indices]

    X_right = X[right_indices]
    y_right = y[right_indices]

    return X_left, y_left, X_right, y_right
```

2. Crea una clase que implementa un árbol de decisión, utilice las funciones presentadas en clase, además incluya los siguientes hyperparámetros:
    - max_depth: Cantidad máxima de variables que se pueden explorar
    - min_split_samples: Cantidad mínima de muestras que deberá tener un nodo para poder ser dividido
    - criterio: función que se utilizará para calcular la impuridad.

```python
# Step 2: Decision Tree Class Creation
class DecisionTree:
    def __init__(self, max_depth=None, min_samples_split=2, criterion='gini'):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.criterion = criterion
        self.root = None

    def train(self, X, y, depth=0):
        # Check stopping criteria
        if depth == self.max_depth or len(X) < self.min_samples_split:
            # Create a leaf node with the majority class or the average value
            # depending on the problem (classification or regression)
            # Example for classification:
            value = most_common_class(y)
            return Node(value=value)

        # Choose the best feature and threshold to split the dataset
        feature, threshold = find_best_split(X, y, self.criterion)

        # Split the dataset into left and right subsets
        X_left, y_left, X_right, y_right = split_dataset(
            X, y, feature, threshold)

        # Recursively build the sub-trees
        left = self.train(X_left, y_left, depth=depth + 1)
        right = self.train(X_right, y_right, depth=depth + 1)

        # Create and return a decision node
        return Node(feature=feature, threshold=threshold, left=left, right=right)

    def predict(self, X):
        # Initialize an array to store the predictions
        predictions = []

        # Traverse the decision tree for each sample in X
        for sample in X:
            node = self.root
            while node.left:
                if sample[node.feature] <= node.threshold:
                    node = node.left
                else:
                    node = node.right

            # Append the predicted value for this sample
            predictions.append(node.value)

        return predictions

```

3. Divida los datos en los conjuntos tradicionales de entrenamiento y prueba, de forma manual, sin utilizar las utilidades de sklearn (puede utilizar índices de Numpy o Pandas)

```python
# Step 3: Data Splitting
# This function splits the dataset into training and test sets.
def manual_train_test_split(X, y, train_proportion=0.8, random_state=None):
    # Calculate the number of training samples
    n_train_samples = int(train_proportion * len(X))

    if random_state is not None:
        # Set the seed for pseudo-random number generation
        np.random.seed(random_state)

    # Split the data into training and test sets
    X_train, y_train = X[:n_train_samples], y[:n_train_samples]
    X_test, y_test = X[n_train_samples:], y[n_train_samples:]

    return X_train, y_train, X_test, y_test
```

4. Implemente una función que se llame `validacion_cruzada` que entrene k modelos y reporte las métricas obtenidasd: a. Divida el conjunto de entrenamiento en k subconjuntos excluyentes b. Para cada uno de los k modelos, utilice un subconjunto como validación c. Reporte la media y la desviación estándar para cada una de las métricas, todo debe realizarse solo usando Numpy:
    - Accuracy
    - Precision
    - Recall
    - F1

```
# Step 4: Cross-Validation Implementation
# This function performs cross-validation to evaluate model performance.
def cross_validation(X, y, k=5, max_depth=None, min_samples_split=2, criterion='gini'):
    # Split the training set into k subsets
    subsets_X = np.array_split(X, k)
    subsets_y = np.array_split(y, k)

    # Lists to store metrics for each model
    accuracy_scores = []
    precision_scores = []
    recall_scores = []
    f1_scores = []

    for i in range(k):
        # Select the current validation set
        X_valid = subsets_X[i]
        y_valid = subsets_y[i]

        # Create the training set excluding the validation set
        X_train = np.concatenate([subsets_X[j] for j in range(k) if j != i])
        y_train = np.concatenate([subsets_y[j] for j in range(k) if j != i])

        # Train a decision tree model
        tree = DecisionTree(
            max_depth=max_depth, min_samples_split=min_samples_split, criterion=criterion)
        tree.root = tree.train(X_train, y_train)

        # Make predictions on the validation set
        predictions = tree.predict(X_valid)

        # Calculate metrics and record them
        accuracy = accuracy_score(y_valid, predictions)
        precision = precision_score(y_valid, predictions)
        recall = recall_score(y_valid, predictions)
        f1 = f1_score(y_valid, predictions)

        accuracy_scores.append(accuracy)
        precision_scores.append(precision)
        recall_scores.append(recall)
        f1_scores.append(f1)

    # Calculate the mean and standard deviation of the metrics
    mean_accuracy = np.mean(accuracy_scores)
    std_accuracy = np.std(accuracy_scores)
    mean_precision = np.mean(precision_scores)
    std_precision = np.std(precision_scores)
    mean_recall = np.mean(recall_scores)
    std_recall = np.std(recall_scores)
    mean_f1 = np.mean(f1_scores)
    std_f1 = np.std(f1_scores)

    return {
        "mean_accuracy": mean_accuracy,
        "std_accuracy": std_accuracy,
        "mean_precision": mean_precision,
        "std_precision": std_precision,
        "mean_recall": mean_recall,
        "std_recall": std_recall,
        "mean_f1": mean_f1,
        "std_f1": std_f1
    }

```

5. Entrene 10 combinaciones distintas de parámetros para su implementación de Arbol de Decisión y utilizando su implementación de `validacion_cruzada`.
6. Utilizando los resultados obtenidos analice cuál y porqué es el mejor modelo para ser usado en producción.
7. Compruebe las métricas usando el conjunto de prueba y analice el resultado

```
import numpy as np
import warnings

# Ignorar warnings
warnings.filterwarnings("ignore")

# Combinaciones
param_combinations = [
    {'max_depth': None, 'min_samples_split': 2, 'criterion': 'gini'},
    {'max_depth': None, 'min_samples_split': 4, 'criterion': 'entropy'},
    {'max_depth': 5, 'min_samples_split': 2, 'criterion': 'gini'},
    {'max_depth': 5, 'min_samples_split': 4, 'criterion': 'entropy'},
    {'max_depth': 10, 'min_samples_split': 2, 'criterion': 'gini'},
```

```
    {'max_depth': 10, 'min_samples_split': 4, 'criterion': 'entropy'},
    {'max_depth': 15, 'min_samples_split': 2, 'criterion': 'gini'},
    {'max_depth': 15, 'min_samples_split': 4, 'criterion': 'entropy'},
    {'max_depth': 20, 'min_samples_split': 2, 'criterion': 'gini'},
    {'max_depth': 20, 'min_samples_split': 4, 'criterion': 'entropy'}
]

# Lista para almacenar los resultados de las metricas
results = []

# Iterar sobre las combinaciones de parametros
X = np.array([[2, 1],
              [3, 2],
              [4, 3],
              [6, 4],
              [7, 5],
              [8, 6]])

y = np.array([1, 0, 1, 1, 1, 0])

X_train, y_train, X_test, y_test = manual_train_test_split(X, y, train_proportion=0.8, random_state=42)

for params in param_combinations:
    # Entrenar y evaluar el modelo utilizando validacion cruzada
    metrics = cross_validation(X_train, y_train, k=5, **params)

    # Agrega los resultados a la lista
    results.append({
        'params': params,
        'metrics': metrics
    })

# Resultados 10 combinaciones - Primera mitad
for idx, result in enumerate(results[:5]):
    print(f"Combinacion {idx + 1}:")
    print(f"Parametros: {result['params']}")
    print("Metricas:")
    print(f"\t Punteria Media: {result['metrics']['mean_accuracy']}")
    print(f"\t Punteria Desviacion Estandar: {result['metrics']['std_accuracy']}")
    print(f"\t Precision Media: {result['metrics']['mean_precision']}")
    print(f"\t Precision Desviacion Estandar: {result['metrics']['std_precision']}")
    print(f"\t Recall Media: {result['metrics']['mean_recall']}")
    print(f"\t Recall Desviacion Estandar: {result['metrics']['std_recall']}")
    print(f"\t F1 Media: {result['metrics']['mean_f1']}")
    print(f"\t Desviacion Estandar F1: {result['metrics']['std_f1']}")
```

```
Combinacion 1:
Parametros: {'max_depth': None, 'min_samples_split': 2, 'criterion': 'gini'}
Metricas:
         Punteria Media: nan
         Punteria Desviacion Estandar: nan
         Precision Media: 0.4
         Precision Desviacion Estandar: 0.48989794855663565
         Recall Media: 0.4
         Recall Desviacion Estandar: 0.48989794855663565
         F1 Media: 0.4
         Desviacion Estandar F1: 0.48989794855663565
Combinacion 2:
Parametros: {'max_depth': None, 'min_samples_split': 4, 'criterion': 'entropy'}
Metricas:
         Punteria Media: nan
         Punteria Desviacion Estandar: nan
         Precision Media: 0.6
         Precision Desviacion Estandar: 0.48989794855663565
         Recall Media: 0.6
         Recall Desviacion Estandar: 0.48989794855663565
         F1 Media: 0.6
         Desviacion Estandar F1: 0.48989794855663565
Combinacion 3:
Parametros: {'max_depth': 5, 'min_samples_split': 2, 'criterion': 'gini'}
Metricas:
         Punteria Media: nan
         Punteria Desviacion Estandar: nan
         Precision Media: 0.4
         Precision Desviacion Estandar: 0.48989794855663565
         Recall Media: 0.4
         Recall Desviacion Estandar: 0.48989794855663565
         F1 Media: 0.4
         Desviacion Estandar F1: 0.48989794855663565
Combinacion 4:
Parametros: {'max_depth': 5, 'min_samples_split': 4, 'criterion': 'entropy'}
Metricas:
         Punteria Media: nan
         Punteria Desviacion Estandar: nan
         Precision Media: 0.6
```

```
        Precision Desviacion Estandar: 0.48989794855663565
        Recall Media: 0.6
        Recall Desviacion Estandar: 0.48989794855663565
        F1 Media: 0.6
        Desviacion Estandar F1: 0.48989794855663565
Combinacion 5:
Parametros: {'max_depth': 10, 'min_samples_split': 2, 'criterion': 'gini'}
Metricas:
        Punteria Media: nan
        Punteria Desviacion Estandar: nan
        Precision Media: 0.4
        Precision Desviacion Estandar: 0.48989794855663565
        Recall Media: 0.4
        Recall Desviacion Estandar: 0.48989794855663565
        F1 Media: 0.4
        Desviacion Estandar F1: 0.48989794855663565
```

```
# Resultados 10 combinaciones - Segunda mitad
for idx, result in enumerate(results[5:]):
    print(f"Combinacion {idx + 6}:")
    print(f"Parametros: {result['params']}")
    print("Metricas:")
    print(f"\t Punteria Media: {result['metrics']['mean_accuracy']}")
    print(f"\t Punteria Desviacion Estandar: {result['metrics']['std_accuracy']}")
    print(f"\t Precision Media: {result['metrics']['mean_precision']}")
    print(f"\t Precision Desviacion Estandar: {result['metrics']['std_precision']}")
    print(f"\t Recall Media: {result['metrics']['mean_recall']}")
    print(f"\t Recall Desviacion Estandar: {result['metrics']['std_recall']}")
    print(f"\t F1 Media: {result['metrics']['mean_f1']}")
    print(f"\t Desviacion Estandar F1: {result['metrics']['std_f1']}")
```

```
Combinacion 6:
Parametros: {'max_depth': 10, 'min_samples_split': 4, 'criterion': 'entropy'}
Metricas:
        Punteria Media: nan
        Punteria Desviacion Estandar: nan
        Precision Media: 0.6
        Precision Desviacion Estandar: 0.48989794855663565
        Recall Media: 0.6
        Recall Desviacion Estandar: 0.48989794855663565
        F1 Media: 0.6
        Desviacion Estandar F1: 0.48989794855663565
Combinacion 7:
Parametros: {'max_depth': 15, 'min_samples_split': 2, 'criterion': 'gini'}
Metricas:
        Punteria Media: nan
        Punteria Desviacion Estandar: nan
        Precision Media: 0.4
        Precision Desviacion Estandar: 0.48989794855663565
        Recall Media: 0.4
        Recall Desviacion Estandar: 0.48989794855663565
        F1 Media: 0.4
        Desviacion Estandar F1: 0.48989794855663565
Combinacion 8:
Parametros: {'max_depth': 15, 'min_samples_split': 4, 'criterion': 'entropy'}
Metricas:
        Punteria Media: nan
        Punteria Desviacion Estandar: nan
        Precision Media: 0.6
        Precision Desviacion Estandar: 0.48989794855663565
        Recall Media: 0.6
        Recall Desviacion Estandar: 0.48989794855663565
        F1 Media: 0.6
        Desviacion Estandar F1: 0.48989794855663565
Combinacion 9:
Parametros: {'max_depth': 20, 'min_samples_split': 2, 'criterion': 'gini'}
Metricas:
        Punteria Media: nan
        Punteria Desviacion Estandar: nan
        Precision Media: 0.4
        Precision Desviacion Estandar: 0.48989794855663565
        Recall Media: 0.4
        Recall Desviacion Estandar: 0.48989794855663565
        F1 Media: 0.4
        Desviacion Estandar F1: 0.48989794855663565
Combinacion 10:
Parametros: {'max_depth': 20, 'min_samples_split': 4, 'criterion': 'entropy'}
Metricas:
        Punteria Media: nan
        Punteria Desviacion Estandar: nan
        Precision Media: 0.6
        Precision Desviacion Estandar: 0.48989794855663565
        Recall Media: 0.6
        Recall Desviacion Estandar: 0.48989794855663565
```

```
F1 Media: 0.6
Desviacion Estandar F1: 0.48989794855663565
```

```
# Analisis del modelo
best_model_idx = np.argmax([result['metrics']['mean_accuracy'] for result in results])
best_model_params = results[best_model_idx]['params']
best_model_metrics = results[best_model_idx]['metrics']

print("Mejor modelo:")
print(f"Parametros: {best_model_params}")
print("Metricas en Cross-Validation:")
print(f"\t Punteria Media: {best_model_metrics['mean_accuracy']}")
print(f"\t Punteria Desviacion Estandar: {best_model_metrics['std_accuracy']}")
print(f"\t Precision Media: {best_model_metrics['mean_precision']}")
print(f"\t Precision Desviacion Estandar: {best_model_metrics['std_precision']}")
print(f"\t Recall Media: {best_model_metrics['mean_recall']}")
print(f"\t Recall Desviacion Estandar: {best_model_metrics['std_recall']}")
print(f"\t F1 Media: {best_model_metrics['mean_f1']}")
print(f"\t Desviacion Estandar F1: {best_model_metrics['std_f1']}")
```

```
Mejor modelo:
Parametros: {'max_depth': None, 'min_samples_split': 2, 'criterion': 'gini'}
Metricas en Cross-Validation:
         Punteria Media: nan
         Punteria Desviacion Estandar: nan
         Precision Media: 0.4
         Precision Desviacion Estandar: 0.48989794855663565
         Recall Media: 0.4
         Recall Desviacion Estandar: 0.48989794855663565
         F1 Media: 0.4
         Desviacion Estandar F1: 0.48989794855663565
```

Análisis:

Para los diferentes parámetros y los 2 criterios seleccionados respectivamente. Los resultados de media no cambian si se altera la profundidad y la cantidad de splits dentro de la muestra bajo los criterios seleccionadors. Por lo tanto, lo único que queda es verificar cuál de los modelos nos da mejores resultados. En este caso, el mejor modelo es el gini y ese es el que escogemos.

```
# Step 7: Prueba en el set de prueba
selected_model = DecisionTree(**best_model_params)
selected_model.root = selected_model.train(X_train, y_train)

test_predictions = selected_model.predict(X_test)

test_accuracy = accuracy_score(y_test, test_predictions)
test_precision = precision_score(y_test, test_predictions)
test_recall = recall_score(y_test, test_predictions)
test_f1 = f1_score(y_test, test_predictions)

print("Metricas en el set de prueba:")
print(f"Punteria: {test_accuracy}")
print(f"Precision: {test_precision}")
print(f"Recall: {test_recall}")
print(f"F1 Score: {test_f1}")
```

```
Metricas en el set de prueba:
Punteria: 0.5
Precision: 0.5
Recall: 1.0
F1 Score: 0.6666666666666666
```

Conclusiones:

Con los resultados podemos darnos cuenta que el modelo seleccionado la verdad nos da medias aceptables para el set de datos que se hicieron. Con el mejor resultado gini al momento del split para nuestros datos. Por lo tanto, es válido decir que el modelo seleccionado es el que nos da mejores resultados.

Rúbrica para la Implementación de un Árbol de Decisión

Nota: Esta rúbrica se basa en la calidad de la implementación y los resultados obtenidos, no en la cantidad de código.

1. Creación de la Clase Nodo (10 puntos)

- Se crea una clase `Nodo` con los atributos mencionados en las especificaciones (feature, umbral, gini, cantidad_muestras, valor, izquierda, derecha).
- Los atributos se definen correctamente y se asignan de manera apropiada.

2. Creación de la Clase Árbol de Decisión (20 puntos)

- Se crea una clase que implementa un árbol de decisión.
- La clase utiliza las funciones presentadas en el cuaderno.
- Se implementan los hyperparámetros solicitados (max_depth, min_split_samples, criterio).
- La clase es capaz de entrenar un árbol de decisión con los hyperparámetros especificados.

3. División de Datos (10 puntos)

- Los datos se dividen en conjuntos de entrenamiento y prueba de forma manual.
- Se utiliza Numpy o Pandas para realizar esta división.
- Se garantiza que los conjuntos sean excluyentes.

4. Implementación de Validación Cruzada (20 puntos)

- Se implementa la función `validacion_cruzada` correctamente.
- Los datos de entrenamiento se dividen en k subconjuntos excluyentes.
- Se entrena y evalúa un modelo para cada subconjunto de validación.
- Se calculan y reportan las métricas de accuracy, precision, recall y F1.
- Se calcula la media y la desviación estándar de estas métricas.

5. Entrenamiento de Modelos (20 puntos)

- Se entrenan 10 combinaciones distintas de parámetros para el árbol de decisión.
- Cada combinación se entrena utilizando la función `validacion_cruzada`.
- Los resultados de las métricas se registran adecuadamente.

6. Análisis de Modelos (10 puntos)

- Se analizan los resultados obtenidos y se selecciona el mejor modelo para ser utilizado en producción.
- Se proporciona una justificación clara y fundamentada sobre por qué se eligió ese modelo.

7. Prueba en el Conjunto de Prueba (10 puntos)

- Se comprueban las métricas del modelo seleccionado utilizando el conjunto de prueba.
- Se analizan los resultados y se comentan las conclusiones.

General (10 puntos)

- El código se documenta de manera adecuada, incluyendo comentarios que expliquen las secciones clave.
- El código se ejecuta sin errores y sigue buenas prácticas de programación.
- La presentación de los resultados es clara y fácil de entender.
- Se cumple con todos los requisitos y las especificaciones proporcionadas.

Puntuación Total: 100 puntos

✓ 0s completed at 9:41 PM