TP1_DecisionTrees_LB_WJ_EV

April 22, 2022

1 Curso de Aprendizaje Automático

2 Trabajo Practico 1: Arboles de Clasificación y Regresión

Escuela de Ingeniería en Computación | Instituto Tecnológico de Costa Rica

Realizado por * Luis Badilla Ortiz * William Jiménez García * Esteban Villalobos Gómez

Fecha de entrega * 24 de abril de 2022

Entrega * Un archivo .zip con el código fuente LaTeX o Lyx, el pdf, y un notebook en jupyter, debidamente documentado, con una función definida por ejercicio. A través del TEC-digital.

Modo de trabajo * Grupos de 3 personas.

En el presente trabajo práctico se introducirán los arboles de clasificación y regresión (CART).

---

3 Imports

[ ]:
```
import io
import pandas
import torch
import numpy as np
from sklearn.model_selection import ShuffleSplit, KFold
from google.colab import files
```

4 Variables de uso general

[ ]:
```
# Constants
min_int = np.iinfo('int').min
max_int = np.iinfo('int').max

# Misc
features = ['Rooms', 'Size', 'Toilets', 'Parking']
ranges = [min_int, 400000, 580000, 900000, max_int]
categories = [1, 2, 3, 4]
```

5 1. Implementación de la clasificación multi-clase con árboles de decisión

5.1 1.1 Pre-procesamiento de los Datos

[ ]:
```
#dataset taken from https://www.kaggle.com/yashsawarn/wifi-stretgth-for-rooms

def read_dataset(csv_name = 'sao-paulo-properties-april-2019.csv'):
    """
    Reads a csv dataset
    returns it as a pytorch tensor
    """
    #Upload local file
    #uploaded = files.upload()
    #data_frame = pandas.read_csv(io.BytesIO(uploaded[csv_name]))

    data_frame = pandas.read_csv(csv_name)

    # Discretize to 1-4 categories
    data_frame['Category'] = pandas.cut(data_frame['Price'], ranges,
                                        labels=categories)

    colums = features.copy()
    colums.append('Category')

    data_frame = data_frame[colums]

    data_tensor = torch.tensor(data_frame.to_numpy())

    return data_tensor

dataset_torch = read_dataset()
print(dataset_torch)
```

5.2 1.2 Implementación de funciones del arbol de clasificación

[ ]:
```
class Node_CART:
    def __init__(self, num_classes = 4, ref_CART = None, current_depth = 0):
        """
        Create the node attributes
        param num_classes: K number of classes to classify
        param ref_cart: reference to the tree containing the node
        param current_depth: current depth of the node in the tree
        """
        self.ref_CART = ref_CART
        self.threshold_value = 0
        self.feature_num = 0
```

```python
        self.node_right = None
        self.node_left = None
        self.data_torch_partition = None
        self.gini = 0
        self.dominant_class = None
        self.accuracy_dominant_class = None
        self.num_classes = num_classes
        self.current_depth = current_depth

    def to_xml(self, current_str = ""):
        """
        Recursive function to write the node content to an xml formatted string
        param current_str : the xml content so far in the whole tree
        return the string with the node content
        """

        str_node = f"<node>" \
                   f"<thresh>{str(self.threshold_value)}</thresh>" \
                   f"<feature>{str(self.feature_num)}</feature>" \
                   f"<depth>{str(self.current_depth)}</depth>" \
                   f"<gini>{str(self.gini)}</gini>"

        if self.node_right:
          str_node += f'<right>{self.node_right.to_xml(current_str)}</right>'
        if self.node_left:
          str_node += f'<left>{self.node_left.to_xml(current_str)}</left>'

        if self.is_leaf():
          str_node += f"<dominant_class>{str(self.dominant_class)}" \
                      f"</dominant_class><acc_dominant_class>" \
                      f"{str(self.accuracy_dominant_class)}" \
                      f"</acc_dominant_class>"

        str_node += "</node>"
        return str_node

    def is_leaf(self):
        """
        Checks whether the node is a leaf
        """
        return self.node_left is None and self.node_right is None

    def create_with_children(self, data_torch, current_depth,
                             list_selected_features = [], min_gini = 0.000001):
        """
        Creates a node by selecting the best feature and threshold, and
        if needed, creating its children.
```

```python
        param data_torch: dataset with the current partition to deal with in
                          the node
        param current_depth: depth counter for the node
        param list_selected_features: list of selected features so far for the
                                      CART building process
        param min_gini: hyperparameter selected by the user defining the minimum
                        tolerated Gini coefficient for a  node
        return the list of selected features so far
        """
        # update depth of children
        depth_children = current_depth + 1
        if depth_children <= self.ref_CART.get_max_depth():
            num_observations = data_torch.shape[0]
            # Careful with max depth
            # if no threshold and feature were selected, select it using a
            # greedy approach
            (threshold_value, feature_num, gini) = \
              self.select_best_feature_and_thresh(
                  data_torch, list_features_selected = list_selected_features,
                  num_classes = num_observations)

            list_selected_features += [feature_num]
            # store important data in attributes
            self.threshold_value = threshold_value
            self.feature_num = feature_num
            self.data_torch_partition = data_torch
            self.gini = gini
            num_features = data_torch.shape[1]

            # create the right and left node data if the current gini
            # is still high

            if self.gini > min_gini:
                data_torch_left = data_torch[data_torch[:, feature_num] <␣
↪threshold_value]
                data_torch_right = data_torch[data_torch[:, feature_num] >=␣
↪threshold_value]
                #if the new partitions have more than min_observations, make␣
↪them
                if data_torch_left.shape[0] >= self.ref_CART.
↪get_min_observations() \
                        and data_torch_right.shape[0] >= self.ref_CART.
↪get_min_observations():
                    #add data to the right and left children
                    self.node_right = Node_CART(num_classes = self.num_classes,
                                                ref_CART = self.ref_CART,
```

```
                                                        current_depth = depth_children)
                    self.node_left = Node_CART(num_classes = self.num_classes,
                                                       ref_CART = self.ref_CART,
                                                       current_depth = depth_children)
                    list_selected_features = self.node_right.
↪create_with_children(
                        data_torch_right, depth_children,
                        list_selected_features = list_selected_features) + \
                    self.node_left.create_with_children(
                        data_torch_left, depth_children,
                        list_selected_features = list_selected_features)
        #if is leaf, fill the dominant class and accuracy
        if self.is_leaf():
            labels_data = data_torch[:,  -1]
            self.dominant_class = torch.mode(labels_data).values.item()
            num_obs_label = labels_data[labels_data == self.dominant_class].
↪shape[0]
            self.accuracy_dominant_class = num_obs_label / labels_data.shape[0]

        return list_selected_features

    def select_best_feature_and_thresh(self, data_torch,
                                       list_features_selected = [],
                                       num_classes = 4):
        """
        ONLY USE  2 FORS
        Selects the best feature and threshold that minimizes the gini␣
↪coefficient
        param data_torch: dataset partition to analyze
        param list_features_selected list of features selected so far, thus␣
↪must be ignored
        param num_classes: number of K classes to discriminate from
        return min_thresh, min_feature, min_gini found for the dataset␣
↪partition when
        selecting the found feature and threshold
        """

        rows, columns = data_torch.size()
        min_gini = max_int
        min_feature = -1
        min_thresh = -1

        # Iterate over features
        for position in range(columns - 1):

            featureValues = data_torch[:, position]
            tested_values = set()
```

```python
            # Iterate over feature values
            for val in featureValues:

              # Skip already used thresholds
              if val in tested_values:
                continue

              tested_values.add(val)

              # LEFT DATA
              data_left = data_torch[featureValues < val]
              nleft = data_left.size()[0]
              df = pandas.DataFrame(data_left.numpy())
              df = df.groupby([columns - 1])[columns - 1].count()
              left_tensor = torch.tensor(df.to_numpy())

              # RIGHT DATA
              data_right = data_torch[featureValues >= val]
              nright = data_right.size()[0]
              df = pandas.DataFrame(data_right.numpy())
              df = df.groupby([columns - 1])[columns - 1].count()
              right_tensor = torch.tensor(df.to_numpy())

              # Calc Gini weighted
              gini_left = nleft/rows * self.calculate_gini(left_tensor, nleft)
              gini_right = nright/rows * self.calculate_gini(right_tensor,
                                                               nright)

              current_gini = gini_left + gini_right

              if current_gini < min_gini:
                  min_gini = current_gini
                  min_feature = position
                  min_thresh = val

        #print(f"Min Gini: {min_gini} | Min Feature: {features[min_feature]} |␣
↪Min Thresh: {min_thresh}")
        # Return selected cut
        return min_thresh, min_feature, min_gini

    def calculate_gini(self, data_partition_torch, num_classes = 4):
        """
        Calculates the gini coefficient for a given partition with the given
        number of classes

        param data_partition_torch: current dataset partition as a tensor
        param num_classes: K number of classes to discriminate from
```

```
        returns the calculated gini coefficient
        """

        # Data divided by num classes
        data_partition_torch = data_partition_torch / num_classes

        # Power by 2 data
        data = torch.pow(data_partition_torch, 2)

        # Sum powered data
        data = torch.sum(data)

        # Calc Gini
        gini = 1 - data

        return gini

    def evaluate_node(self, input_torch):
        """
        Evaluates an input observation within the node.
        If is not a leaf node, send it to the corresponding node
        return predicted label
        """
        feature_val_input = input_torch[self.feature_num]
        if self.is_leaf():
            return self.dominant_class
        else:
            if feature_val_input < self.threshold_value:
                return self.node_left.evaluate_node(input_torch)
            else:
                return self.node_right.evaluate_node(input_torch)


class CART:
    def __init__(self, dataset_torch, max_CART_depth = 4, min_observations = 2):
        """
        CART has only one root node
        """
        #min observations per node
        self.min_observations = min_observations
        self.root = Node_CART(num_classes = 4, ref_CART = self, current_depth = 0)
        self.max_CART_depth = max_CART_depth
        self.list_selected_features = []

    def get_root(self):
        """
```

```python
        Gets tree root
        """
        return self.root

    def get_min_observations(self):
        """
        return min observations per node
        """
        return self.min_observations

    def get_max_depth(self):
        """
        Gets the selected max depth of the tree
        """
        return self.max_CART_depth

    def build_CART(self, data_torch):
        """
        Build CART from root
        """
        self.list_selected_features = self.root.create_with_children(
            data_torch, current_depth = 0)

    def to_xml(self, xml_file_name):
        """
        write Xml file with tree content
        """
        str_nodes = self.root.to_xml()
        file = open(xml_file_name,"w+")
        file.write(str_nodes)
        file.close()
        return str_nodes


    def evaluate_input(self, input_torch):
        """
        Evaluate a specific input in the tree and get the predicted class
        """
        return self.root.evaluate_node(input_torch)


def train_CART(dataset_torch, name_xml = "", max_CART_depth = 3,
               min_obs_per_leaf = 2):
    """
    Train CART model
    """
    tree = CART(dataset_torch = dataset_torch, max_CART_depth = max_CART_depth,
```

```
                min_observations =  min_obs_per_leaf)
    tree.build_CART(dataset_torch)
    if name_xml != "":
        tree.to_xml(name_xml)
    return tree

def test_CART(tree, testset_torch):
    """
    Test a previously built CART
    """
    #TODO: COMPLETE | Use tree.evaluate_input(current_observation) for this

    n = testset_torch.shape[0]
    correct = 0

    for current_observation in testset_torch:
        if current_observation[-1] == tree.evaluate_input(current_observation):
            correct += 1

    accuracy = correct / n

    print(f"Total: {n} | Correct: {correct} | Accuracy: {accuracy}")

    return accuracy
```

5.2.1 Pruebas (clasificación)

[ ]:
```
# Test xml
CART_1 = CART(dataset_torch)
CART_1.to_xml("arbolito_vacio.xml")
nodo_A = Node_CART(num_classes = 2, current_depth = 1)
CART_1.root.node_left = nodo_A
CART_1.to_xml("arbolito_peque.xml")
```

[ ]:
```
# TEST calculate_gini
nc = Node_CART()

data = torch.tensor([3, 1])

gini = nc.calculate_gini(data, 4)
print(gini)
```

[ ]:
```
# TEST select_best_feature_and_thresh

nc = Node_CART()
```

```
data = torch.tensor([[3, 22, 7.2, 0],
                     [1, 38, 71.3, 0],
                     [3, 26, 7.9, 1],
                     [1, 35, 53.1, 0]])

gini = nc.select_best_feature_and_thresh(data, [], 4)
print(gini)
```

6 2. Evaluación del CART

6.1 2.1 Conjunto de datos completo

[ ]:
```
# DEPTH 3
print("DEPTH 3")
print("===== TRAINING =====")
tree = train_CART(dataset_torch, name_xml = "Tree_ResultDepth3.xml",
                  max_CART_depth=3)
rr = tree.to_xml("xml_result.xml")
print(rr)
print("===== RESULT =====")
acc = test_CART(tree, dataset_torch)

# DEPTH 2
print("DEPTH 2")
print("===== TRAINING =====")
tree = train_CART(dataset_torch, name_xml = "Tree_ResultDepth2.xml",
                  max_CART_depth=2)
print("===== RESULT =====")
acc = test_CART(tree, dataset_torch)
```

6.1.1 Resultados obtenidos de Arboles de Clasificación

Las tasas de aceirtos para ambos CART es la siguiente:

Accuracy **CART-Profundidad 2**: **0.6510**

Accuracy **CART-Profundidad 3**: **0.6581**

6.2 2.2 Particiones del conjunto de datos

[ ]:
```
def partition_validation(dataset_torch, max_CART_depth, num_splits,
                         train_fn=train_CART, test_fn=test_CART):
  """
  Create and test dataset partitions
  """

  # Shuffle Split
```

```
  shuffle_split = ShuffleSplit(n_splits=num_splits, test_size=.30)

  # Iteration counter
  iteration = 1

  # Results
  results = []

  for train_index, test_index in shuffle_split.split(dataset_torch):

    print(f"Iteration: {iteration}")
    iteration += 1

    # Get Train Data
    train_torch = dataset_torch[train_index]

    # Get Test Data
    test_torch = dataset_torch[test_index]

    # TRAIN
    print("===== TRAINING =====")
    tree = train_fn(train_torch,
                    name_xml = f"Tree_Result_Partition_{iteration}.xml",
                    max_CART_depth=max_CART_depth)

    # TEST
    print("===== RESULT =====")
    acc = test_fn(tree, test_torch)

    # Append Accuracy Result
    results.append(acc)

  return results
```

[ ]:
```
# Depth 2
results_depth2 = partition_validation(dataset_torch, max_CART_depth=2,
                                      num_splits=10)

# Depth 3
results_depth3 = partition_validation(dataset_torch, max_CART_depth=3,
                                      num_splits=10)
```

6.2.1 Evaluación de resultados del Arbol de Clasificación

Se realizaron 10 corridas con cada CART, y los resultados obtenidos se detallan en la tabla a continuación.

La tasa de aciertos para cada corrida, promedio y desviación estándar se detalla a continuación:

Corrida	CART Profundidad-2	CART Profundidad-3
1	0.6535	0.6684
2	0.6466	0.6392
3	0.6548	0.6562
4	0.6521	0.6569
5	0.6385	0.6603
6	0.6453	0.6473
7	0.6623	0.6514
8	0.6508	0.6501
9	0.6508	0.6426
10	0.6303	0.6514
PROMEDIO	0.6485	0.6524
S.D.	0.0089	0.0085

Otras métricas obtenidas:

[ ]:
```
# Show results
results = {
    'CART Depth-2': results_depth2,
    'CART Depth-3': results_depth3
}

results_df = pandas.DataFrame(results)

results_df.describe()
```

7 3. Implementación del Boque Aleatorio (Random Forest)

[ ]:
```
class RandomForests:
    """
    Random Forests Class Implementation
    Used to train and test a random forest
    """
    def __init__(self, num_CARTS, max_CART_depth):
        self.forest = []
        self.num_CARTS = num_CARTS
        self.max_CART_depth = max_CART_depth

    # 3.a
    def train_random_forest(self, dataset_torch):
      """
      Train n CARTS to make a forest
      """
      kf = KFold(n_splits=self.num_CARTS, shuffle=True)
```

```python
    self.forest = []

    iteration = 1

    for train_index, test_index in kf.split(dataset_torch):
      print(f"Iteration Random Forest: {iteration}")
      iteration += 1

      train_torch = dataset_torch[train_index]

      print(f"===== TRAINING =====")
      tree = train_CART(train_torch, name_xml = "",
                        max_CART_depth=self.max_CART_depth)

      self.forest.append(tree)

  # 3.b
  def evaluate_random_forest(self, input_torch):
    """
    Evaluate input torch over the forest
    """
    predictions = []

    for tree in self.forest:
      prediction = tree.evaluate_input(input_torch)
      predictions.append(prediction)

    evaluations_tensor = torch.tensor(predictions)

    # Voting
    voted = torch.mode(evaluations_tensor).values.item()

    return voted

  # 3.c
  def test_random_forest(self, testset_torch):
    """
    Test the forest
    """
    n = testset_torch.shape[0]
    correct = 0

    for current_observation in testset_torch:
      if(current_observation[-1] == self.evaluate_random_forest(
          current_observation)):
        correct += 1
```

```
    accuracy = correct / n

    print(f"Total: {n} | Correct: {correct} | Accuracy: {accuracy}")

    return accuracy
```

7.1 3.c.1 Implementacion de función `test_random_forest`

```
[ ]: # Create Random Forest of 3 CARTS
rf3 = RandomForests(num_CARTS=3, max_CART_depth=3)

# Train Random Forest
rf3.train_random_forest(dataset_torch)

# Test Random Forest
rf3.test_random_forest(dataset_torch)
```

```
[ ]: # Create Random Forest of 5 CARTS
rf = RandomForests(num_CARTS=5, max_CART_depth=3)

# Train Random Forest
rf.train_random_forest(dataset_torch)

# Test Random Forest
rf.test_random_forest(dataset_torch)
```

7.1.1 Resultados obtenidos RF

Despues de un tiempo considerable de entrenamiento de los Random Forests, se obtiene:

Accuracy **RF-3** CARTS: **0.65999**

Accuracy **RF-5** CARTS: **0.65897**

Por lo que pareciera que ambos tienen un accuracy similar, por lo que agregar mas CARTs podría no agregar mayor beneficio.

8 4. Evaluación del Random Forest

```
[ ]: def partition_validation_forest(dataset_torch, num_CARTS, max_CART_depth,
                                num_splits):
  """
  Create and test random forest dataset partitions
  """

  # Shuffle Split
  shuffle_split = ShuffleSplit(n_splits=num_splits, test_size=.30,
                              random_state=0)
```

```python
    # Iteration counter
    iteration = 1

    # Results
    results = []

    # Random Forest
    rf = RandomForests(num_CARTS, max_CART_depth)

    for train_index, test_index in shuffle_split.split(dataset_torch):

      print(f"Iteration: {iteration}")
      iteration += 1

      # Get Train Data
      train_torch = dataset_torch[train_index]

      # Get Test Data
      test_torch = dataset_torch[test_index]

      # TRAIN
      print("===== TRAINING =====")
      rf.train_random_forest(train_torch)

      # TEST
      print("===== RESULT =====")
      acc = rf.test_random_forest(test_torch)

      # Append Accuracy Result
      results.append(acc)

    return results
```

8.1 4.1 Particiones del conjunto de datos

[ ]:
```python
# Num CARTS 3
results_numCARTS3 = partition_validation_forest(dataset_torch, num_CARTS=3,
                                                max_CART_depth=3, num_splits=10)
```

[ ]:
```python
# Num CARTS 5
results_numCARTS5 = partition_validation_forest(dataset_torch, num_CARTS=5,
                                                max_CART_depth=3, num_splits=10)
```

8.1.1 Evaluación de resultados del Random Forest

Los resultados mostrados por el Random Forest, no muestran ninguna mejora significativa con respecto a los resultados de los arboles de 2 y 3 niveles de profundidad entrenados con la totalidad de los datos.

De hecho los Random Forests muestran una dispersión mayor del accuracy promedio.

Corrida	RF-3	RF-5
1	0.6515	0.6514
2	0.6195	0.6310
3	0.6399	0.6541
4	0.6657	0.6705
5	0.6474	0.6432
6	0.6297	0.6296
7	0.6617	0.6548
8	0.6664	0.6630
9	0.6392	0.6439
10	0.6351	0.6439
PROMEDIO	0.6456	0.6486
S.D.	0.0158	0.0129

[ ]:
```python
# Show results
results = {
    'RF 3 CARTs': results_numCARTS3,
    'RF 5 CARTs': results_numCARTS5,
    'CART Depth-2': results_depth2,
    'CART Depth-3': results_depth3
}

results_df = pandas.DataFrame(results)

results_df.describe()
```

9 5. CART para regresión

La implementación de `CART` es posible adaptarla para realizar regresión en vez de clasificación.

9.1 5.1.a Cambios para implementar regresión

Dicha adaptación require los siguentes cambios:

- Cambiar función de error a optimizar, el Gini no es adecuado en este caso, y se debe utilizar una métrica de regresión como: RSME, o MAE.
- El algoritmo de optimizacíon busca el feature y threshold con menor error acumulado tanto del lado izquierdo como derecho del nodo particular.

- El valor estimado por nodo, puede ser un momento estadístico que describa los valores de la variable dependiente para dicha partición. En este caso hemos eligido utilizar el promedio como valor de respuesta.
- La selección del threshold debe cambiarse, ya que ir probando cada valor como lo hace la solución original, genera un arbol desbalanceado, con una hoja de un valor unitario siempre a la izquierda y el resto de valores a la derecha. Dicha hoja presenta un error de 0, lo cual muestra un sobreajuste del arbol a los datos. Por ende, la selección del threshold se modificó para que el punto de corte sea el promedio de los valores del feature escogido.
- El arbol de regresión retorna un valor númerico, equivalente al estimado del precio de la propiedad (observación). Para comparar con el arbol de clasificación se agregó una función sencilla que convierte tanto el precio estimado y como el real a la categoría correspondiente, y calcular así el accuracy de manera análoga.
- El preprocesamiento tuvo que ser reimplementado para retornar como variable dependiente la columna de precio, asi como un torch de numeros flotantes.

Referencias:

1. https://fhernanb.github.io/libro_mod_pred/arb-de-regre.html
2. Apuntes del curso: "Validación" de Saúl Calderón.

9.2 5.1.b Variante del CART para regresión

9.2.1 Preprocesamiento

Para el preprocesamiento, solo se seleccionan los descriptores deseados, la última columna contiene contiene la variable dependiente que para la regresión es el precio, no la categoría.

En este caso, nos interesa que el tensor sea de tipo flotante, pues la estimación va a ser un valor aproximado flotante.

[ ]:
```
# Constants
min_int = np.iinfo('int').min
max_int = np.iinfo('int').max
features = ['Rooms', 'Size', 'Toilets', 'Parking']
label = ['Price']
ranges = [min_int, 400000, 580000, 900000, max_int]
```

[ ]:
```
def read_dataset(csv_name='sao-paulo-properties-april-2019.csv'):
    """
    Reads a csv dataset
    returns it as a pytorch tensor
    """
    data_frame = pandas.read_csv(csv_name, dtype={
        'Price': np.float64, 'Condo': np.int32, 'Size': np.int32,
        'Rooms': np.int32, 'Toilets': np.int32, 'Suites': np.int32,
        'Parking': np.int32, 'Elevator': np.int32, 'Furnished': np.int32,
        'Swimming Pool': np.int32, 'New': np.int32, 'District': str,
        'Negotiation Type': str, 'Property Type': str, 'Latitude': np.float64,
        'Longitude': np.float64
    })
```

```
    #Do data preprocessing and return a torch with targets in the last column
    return torch.tensor(data_frame[features + label].to_numpy())


dataset_regression = read_dataset()
print(dataset_regression)
dataset_regression.dtype
```

9.2.2 Implementación de arbol de regresión

Cambios en código adicionales: * Remover dataset_torch del constructor de CART2 pues no se utiliza. * Renombrar `self.gini` a `self.error`. * Renombrar `self.dominant_class` a `self.estimated_value`. * Eliminar la función calculate_gini. En su lugar se implementaron dos funciones de error a escoger, `calculate_mae` y `calculate_rsme`. Se utilizó el RSME debido a que penaliza mayormente los grandes errores (Apuntes de "Validación" de Saúl Calderón). * Se eliminó num_classes porque no tiene sentido en la regresión.

[ ]:
```
class NodeCART2:
    def __init__(self, ref_CART=None, current_depth=0):
        """
        Create the node attributes
        param num_classes: K number of classes to classify
        param ref_cart: reference to the tree containing the node
        param current_depth: current depth of the node in the tree
        """
        self.ref_CART = ref_CART
        self.threshold_value = 0
        self.feature_num = 0
        self.node_right = None
        self.node_left = None
        self.error = .0
        self.estimated_value = .0
        self.current_depth = current_depth

    def to_xml(self, current_str=""):
        """
        Recursive function to write the node content to a xml formatted string
        param current_str : the xml content so far in the whole tree
        return the string with the node content
        """
        str_node = f"<node>" \
                   f"<thresh>{str(self.threshold_value)}</thresh>" \
                   f"<feature>{str(self.feature_num)}</feature>" \
                   f"<depth>{str(self.current_depth)}</depth>" \
                   f"<error>{str(self.error)}</error>"

        if self.node_right:
```

```
            str_node += f'<right>{self.node_right.to_xml(current_str)}</right>'
        if self.node_left:
            str_node += f'<left>{self.node_left.to_xml(current_str)}</left>'

        if self.is_leaf():
            str_node += f"<estimated_value>{str(self.estimated_value)}" \
                        "</estimated_value>"
        str_node += "</node>"
        return str_node

    def is_leaf(self):
        """
        Checks whether the node is a leaf
        """
        return self.node_left is None and self.node_right is None

    def create_with_children(self, data_torch, current_depth,
                             list_selected_features=[], min_error=50000):
        """
        Creates a node by selecting the best feature and threshold, and if
        needed, creating its children

        param data_torch: dataset with the current partition to deal with in
                          the node
        param current_depth: depth counter for the node
        param list_selected_features: list of selected features so far for the
                          CART building process
        param min_error: hyperparameter selected by the user defining the
                          minimum tolerated Gini coefficient for a  node
        return the list of selected features so far
        """
        # update depth of children
        depth_children = current_depth + 1
        # print(f'depth_children {depth_children} | n {data_torch.shape[0]}')
        if depth_children <= self.ref_CART.get_max_depth():
            # careful with max depth
            # if no threshold and feature were selected, select it using a
            # greedy approach
            (threshold_value, feature_num, error) = \
              self.select_best_feature_and_thresh(data_torch,
                                                  list_selected_features)
            list_selected_features += [feature_num]

            # store important data in attributes
            self.threshold_value = threshold_value
            self.feature_num = feature_num
            self.error = error
```

```
        # create the right and left node data if the current error is still␣
↪high
        if self.error > min_error:
            data_torch_left = data_torch[data_torch[:, feature_num] <␣
↪threshold_value]
            data_torch_right = data_torch[data_torch[:, feature_num] >=␣
↪threshold_value]

            # Test each partition apart, the tree won't be balanced anymore.
            if data_torch_left.shape[0] >= self.ref_CART.
↪get_min_observations() \
                    and data_torch_right.shape[0] >= self.ref_CART.
↪get_min_observations():

                self.node_left = NodeCART2(ref_CART=self.ref_CART,
                                           current_depth=depth_children)

                # add data to the right and left children
                self.node_right = NodeCART2(ref_CART=self.ref_CART,
                                            current_depth=depth_children)

                list_selected_features = \
                  self.node_left.create_with_children(
                    data_torch_left, depth_children, list_selected_features
                  ) + self.node_right.create_with_children(
                      data_torch_right,
                      depth_children, list_selected_features)

    # if is leaf, fill the expected values
    if self.is_leaf():
        labels_data = data_torch[:, -1]
        self.estimated_value = torch.mean(labels_data).item()

    return list_selected_features

  def select_best_feature_and_thresh(self, data_torch,
                                     list_features_selected=[],
                                     num_classes=4):
    """
    ONLY USE  2 FORS
    Selects the best feature and threshold that minimizes the error

    param data_torch: dataset partition to analyze
    param list_features_selected: list of features selected so far,
      thus must be ignored
    param num_classes: number of K classes to discriminate from
```

```python
        return min_thresh, min_feature, min_gini found for the dataset
          partition when selecting the found feature and threshold
        """
        min_error = max_int
        min_feature = -1
        min_thresh = -1

        # Iterate over features
        for feature_num in range(0, data_torch.shape[1] - 1):
            threshold_value = torch.mean(data_torch[:, feature_num]).item()
            data_torch_left = data_torch[data_torch[:, feature_num] < threshold_value]
            data_torch_right = data_torch[data_torch[:, feature_num] >= threshold_value]
            error_left = self.calculate_rsme(data_torch_left,
                                             data_torch_left.shape[0])
            error_right = self.calculate_rsme(data_torch_right,
                                              data_torch_right.shape[0])
            total_error = error_left + error_right

            if total_error < min_error:
                min_feature = feature_num
                min_thresh = threshold_value
                min_error = total_error
                # print(f'Total Error = {total_error} | min_feature = {min_feature} | min_thresh = {threshold_value}')

        # return selected cut
        return min_thresh, min_feature, min_error

    def calculate_rsme(self, data_partition_torch, num_obs):
        """
        Calculates the error of the current partition against the actual label
        using the RSME metric.

        """

        error = 0
        if num_obs:
            labels_data = data_partition_torch[:, -1]
            estimated_label = torch.mean(labels_data).item()
            error = torch.sqrt(
                torch.div(torch.sum(torch.pow(labels_data-estimated_label, 2)),
                          num_obs)).item()
        return error

    def calculate_mae(self, data_partition_torch, num_obs):
```

```
        """
        Calculates the error of the current partition against the actual label
        using the MAE metric.

        """

        error = 0
        if num_obs:
            labels_data = data_partition_torch[:, -1]
            estimated_label = torch.mean(labels_data).item()
            error = torch.div(torch.sum(torch.abs(labels_data-estimated_label)),
                              num_obs).item()
        return error

    def evaluate_node(self, input_torch):
        """
        Evaluates an input observation within the node.
        If is not a leaf node, send it to the corresponding node
        return predicted label
        """
        feature_val_input = input_torch[self.feature_num]
        if self.is_leaf():
            return self.estimated_value
        else:
            if feature_val_input < self.threshold_value:
                return self.node_left.evaluate_node(input_torch)
            else:
                return self.node_right.evaluate_node(input_torch)


class CART2:
    def __init__(self, max_CART_depth=4, min_observations=2):
        """
        CART has only one root node
        """
        # min observations per node
        self.min_observations = min_observations
        self.root = NodeCART2(ref_CART=self, current_depth=0)
        self.max_CART_depth = max_CART_depth
        self.list_selected_features = []

    def get_root(self):
        """
        Gets tree root
        """
        return self.root
```

```python
    def get_min_observations(self):
        """
        return min observations per node
        """
        return self.min_observations

    def get_max_depth(self):
        """
        Gets the selected max depth of the tree
        """
        return self.max_CART_depth

    def build_CART(self, data_torch):
        """
        Build CART from root
        """
        self.list_selected_features = self.root.create_with_children(
            data_torch, current_depth=0)

    def to_xml(self, xml_file_name):
        """
        write Xml file with tree content
        """
        str_nodes = self.root.to_xml()
        with open(xml_file_name, "w+") as file:
          file.write(str_nodes)
        return str_nodes

    def evaluate_input(self, input_torch):
        """
        Evaluate a specific input in the tree and get the predicted class
        """
        return self.root.evaluate_node(input_torch)


def train_CART2(dataset_torch, name_xml=None, max_CART_depth=3,␣
 ↪min_obs_per_leaf=2):
    """
    Train CART model
    """
    tree = CART2(max_CART_depth, min_obs_per_leaf)
    tree.build_CART(dataset_torch)
    if name_xml:
        tree.to_xml(name_xml)
    return tree
```

```python
def _get_category(price):
    for category in range(len(ranges)):
      if price < ranges[category]:
        return category


def test_CART2(tree: CART2, testset_torch):
    """
    Test a previously built CART
    """
    # Use tree.evaluate_input(current_observation) for this

    n = testset_torch.shape[0]
    correct = 0

    for current_observation in testset_torch:
        if _get_category(current_observation[-1]) == _get_category(
            tree.evaluate_input(current_observation)):
          correct += 1

    accuracy = correct / n
    print(f"Total: {n} | Correct: {correct} | Accuracy: {accuracy}")
    return accuracy
```

9.2.3 Pruebas básicas

[ ]:
```python
def test_CART2_small():
    data = torch.tensor([[3, 22, 7, 100.0],
                         [1, 38, 7, 200],
                         [2, 26, 8, 100],
                         [5, 35, 5, 150]])
    tree = train_CART2(data, name_xml='regression_tree.xml', min_obs_per_leaf=1)
    test_CART2(tree, data)

test_CART2_small()
```

[ ]:
```python
def test_CART2_full():
    tree = train_CART2(dataset_regression, name_xml='regression_tree_full.xml')
    test_CART2(tree, dataset_regression)

test_CART2_full()
```

9.3 5.1.c Evaluación de Arboles de Regresión

Se realizaron 10 corridas con particiones distintas con los arboles de regresión de 2 y 3 niveles, sus resultados son los siguientes:

Corrida	Regresión Profundidad-3	Regresión Profundidad-3
1	0.4745	0.5105
2	0.5711	0.6453
3	0.5773	0.5793
4	0.5807	0.5405
5	0.5786	0.5909
6	0.5759	0.5881
7	0.5521	0.6297
8	0.4479	0.5663
9	0.5799	0.5977
10	0.6051	0.5664
PROMEDIO	0.5543	0.5815
S.D.	0.0511	0.0394

[ ]:
```python
def partition_validation(dataset_torch, max_CART_depth, num_splits,
                         train_fn=train_CART2, test_fn=test_CART2):
  """
  Create and test dataset partitions
  """

  # Shuffle Split
  shuffle_split = ShuffleSplit(n_splits=num_splits, test_size=.30)

  # Iteration counter
  iteration = 1

  # Results
  results = []

  for train_index, test_index in shuffle_split.split(dataset_torch):

    print(f"Split: {iteration} of {num_splits}")
    iteration += 1

    # Get Train Data
    train_torch = dataset_torch[train_index]

    # Get Test Data
    test_torch = dataset_torch[test_index]

    acc = test_fn(train_fn(train_torch,
                           #  name_xml = f"Tree_Result_Partition_{iteration}.
↪xml",
                            max_CART_depth=max_CART_depth),
                  test_torch)
```

```
    # Append Accuracy Result
    results.append(acc)

  return results
```

[ ]:
```
# Depth 2
results_reg_depth2 = partition_validation(
    dataset_regression, max_CART_depth=2, num_splits=10, train_fn=train_CART2,
    test_fn=test_CART2)
```

[ ]:
```
# Depth 3
results_reg_depth3 = partition_validation(
    dataset_regression, max_CART_depth=3, num_splits=10, train_fn=train_CART2,
    test_fn=test_CART2)
```

Métricas adicionales:

[ ]:
```
# Show results
results_reg = {
    'Depth2': results_reg_depth2,
    'Depth3': results_reg_depth3
}

results_reg_df = pandas.DataFrame(results_reg)

results_reg_df.describe()
```

9.4 5.1.d Comparación CART para clasificación y CART para regresión

Ambos arboles fueron entrenados con los mismos predictores, sin embargo el objetivo de cada arbol es diferente. Para poder comparar el arbol de regresión con el de clasificación, se hizo la conversion a "categoría" del valor estimado.

La siguiente tabla muestra como el arbol de clasificación tuvo mejor accuracy en promedio en ambas profundidades, donde para dos niveles fue superior un 9.4%, y para tres niveles mostro una mejora del 7.1%.

Además es importante recalcar que la dispersion del accuracy es mucho menor en el arbol de clasificación: 0.0089 para profundidad de dos niveles y .0085 para profundidad de tres niveles; mientras el arbol de regresión muestra una dispersion de 0.05 para prof. de dos niveles y 0.039 para tres niveles de profundidad.

De los resultados se concluye que dados los valores de las medias y disperciones obtenidos, el arbol de clasificación parece ser mejor en esta tarea, mientras que se denota que hay espacio para mejoras en el modelo del arból de regresión.

[ ]:
```
results_comp = {
    'Clasificación Depht-2': results_depth2,
    'Regresion Depth-2': results_reg_depth2,
```

```
    'Clasificación Depht-3': results_depth3,
    'Regresion Depth-3': results_reg_depth3,
}

results_comp_df = pandas.DataFrame(results_comp)
results_comp_df.describe()
```