Trabajo Práctico 1: Árboles de decisión

Andrey Arguedas Espinoza
2020426569
and12rx12@gmail.com

Heiner León Aguilar
2013006040
heiner@hey.com

Victor A Ortiz Ruiz
8705127
voruiz@gmail.com

1. Introducción

Para este práctica realizamos una implementación del algoritmo de "Arboles de Decisión" así como los "Bosques Aleatorios" [1] utilizando un dataset que nos provee la información de bienes raíces en la ciudad de Sao Paulo, Brasil.

En este dataset encontramos información general de bienes raíces tales como el tamaño, el tipo, la cantidad de cuartos, baños, piscinas y estacionamientos así como el precio en el que fue vendida la propiedad y la localidad. Nuestro trabajo será poder predecir mediante los atributos anteriormente mencionados en que escala de precios cae una propiedad X.

```
...        Price  Condo  Size  Rooms  Toilets  Suites  Parking  Elevator  \
0     10000000      0   343      4        7       4        5         0
1      9979947      0   343      4        6       4        5         1
2      8500000   7200   420      4        6       4        4         1
3      8039200      0   278      4        7       4        4         1
4      8000000      0   278      4        5       3        5         1
...        ...    ...   ...    ...      ...     ...      ...       ...
4889    245000    340    47      2        1       0        1         0
4890    245000    257    57      2        1       0        1         0
4891    245000    490    56      2        1       0        1         0
4892    245000    393    48      2        1       0        1         0
4893    245000    269    50      2        1       0        1         0

      Furnished  Swimming Pool  New                   District  \
0             0              0    0          Iguatemi/São Paulo
1             0              1    0          Iguatemi/São Paulo
2             0              1    0   Jardim Paulista/São Paulo
3             1              1    0      Vila Olimpia/São Paulo
4             0              1    0      Vila Olimpia/São Paulo
...         ...            ...  ...                         ...
4889          0              0    0        Aricanduva/São Paulo
4890          0              0    0          Cangaíba/São Paulo
4891          0              1    0 Ermelino Matarazzo/São Paulo
4892          0              0    0        São Lucas/São Paulo
4893          0              1    0        São Mateus/São Paulo

     Negotiation Type Property Type   Latitude  Longitude
0                sale     apartment -23.585487 -46.681676
1                sale     apartment -23.585487 -46.681676
2                sale     apartment -23.564044 -46.660862
3                sale     apartment -23.596469 -46.680587
4                sale     apartment -23.596469 -46.680587
...               ...           ...        ...        ...
4889             sale     apartment -23.573408 -46.503064
4890             sale     apartment -23.496113 -46.510703
4891             sale     apartment -23.493728 -46.470292
4892             sale     apartment -23.606920 -46.523416
4893             sale     apartment   0.000000   0.000000

[4894 rows x 16 columns]
```

Figura 1: Dataset original

Para poder lograr nuestro objetivo tomaremos en cuenta solamente ciertas características del dataset tales como 'Rooms', 'Size', 'Toilets', 'Parking' e ignoramos el resto. Posteriormente categorizamos los rangos de precios en etiquetas con valores de 1 a 4:

- Categoría 4: $900000 <$ precio.
- Categoría 3: $580000 <$ precio < 900000.
- Categoría 2: $400000 <$ precio < 580000.
- Categoría 1: precio < 400000.

	Rooms	Size	Toilets	Parking	Class
0	4	343	7	5	4
1	4	343	6	5	4
2	4	420	6	4	4
3	4	278	7	4	4
4	4	278	5	5	4
...	...	...	...	...	...
4889	2	47	1	1	1
4890	2	57	1	1	1
4891	2	56	1	1	1
4892	2	48	1	1	1
4893	2	50	1	1	1

[4894 rows x 5 columns]

Figura 2: Dataset modificado

2. Arboles de decisión

Para los árboles de decisión, necesitamos calcular y minimizar el gini para las particiones. Para esto, implementamos las siguientes funciones:

2.1. Calculo del gini

```python
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
    """
    uniq, counts = data_partition_torch.unique(return_counts=True)
    totalQty = data_partition_torch.shape[0]
    gini = 1 - sum( (counts / totalQty)  **2)
    return gini
```

Figura 3: Función de caluclo del gini para una partición

2.2. Best feature and tresh y gini ponderado

```python
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
    """
    min_thresh = 0
    min_feature = ''
    min_gini = 1

    for feature_num in range(num_classes):
        for tresh in data_torch[:, feature_num].unique():
            data_torch_left = data_torch[data_torch[:, feature_num] < tresh]
            data_torch_right = data_torch[data_torch[:, feature_num] >= tresh]

            left_classes, right_classes = data_torch_left[:, -1], data_torch_right[:, -1]

            gini_left = self.calculate_gini(left_classes, num_classes)
            gini_right = self.calculate_gini(right_classes, num_classes)
            gini_ponderado = ((data_torch_left.shape[0] / data_torch.shape[0]) * gini_left) + ((data_torch_right.shape[0] / data_torch.shape[0]) * gini_right)

            if gini_ponderado < min_gini:
                min_gini = gini_ponderado
                min_feature = feature_num
                min_thresh = tresh.item()


    return (min_thresh, min_feature, min_gini)
```

Figura 4: Best feature and tresh

2.3. Test CART

```
def test_CART(tree, testset_torch):
    """
    Test a previously built CART
    """
    correct_observations = 0
    for observation in testset_torch:
        predicted = tree.evaluate_input(observation)
        classy = observation[-1].item()
        if predicted == classy:
            correct_observations += 1
    return correct_observations / testset_torch.shape[0]
```

Figura 5: Cantidad de aciertos entre registros totales

2.4. Validación de particiones

```
def partition_validation(dataset_torch, max_CART_depth, num_splits):
    rs = ShuffleSplit(n_splits=num_splits, train_size=0.7, test_size=0.3, random_state=0)
    accuracys = []
    for train_index, test_index in rs.split(dataset_torch):
        treex = train_CART(dataset_torch[train_index], name_xml = "CART_depth_partitions.xml", max_CART_depth = max_CART_depth)
        accx = test_CART(treex, dataset_torch[test_index])
        accuracys.append([accx])
    return accuracys
```

Figura 6: Creación de particiones con Sklearn

Gracias a los funciones anteriores nuestra implementación es capaz de crear los arboles de decisión y poder mostrarlos en archivos XML, seguidamente con el CART construido debemos evaluarlo.

2.5. Evaluación del CART

```
#### EVALUACION DEL CART

print("Probando con un depth de 2 nodos \n")
tree2 = train_CART(dataset_torch, name_xml = "CART_depth_2.xml", max_CART_depth = 2)
acc2 = test_CART(tree2, dataset_torch)
print("The accuracy for depth 2 is of: ", acc2, "\n")

print("Probando con un depth de 3 nodos \n")
tree3 = train_CART(dataset_torch, name_xml = "CART_depth_3.xml", max_CART_depth = 3)
acc3 = test_CART(tree3, dataset_torch)
print("The accuracy for depth 3 is of: ", acc3, "\n")
```

Figura 7: Evaluación con CARTs de 2 y 3 nodos

```
The accuracy for depth 2 is of:  0.6438496117695137

The accuracy for depth 3 is of:  0.6442582754393135
```

Figura 8: Resultados de evaluación con CARTs de 2 y 3 nodos

2.6. Evaluación del CART con múltiples particiones

Finalmente probamos con 10 particiones para 2 y 3 nodos y así poder visualizar el promedio y la desviación estándar

```python
### PARTICIONES

print("************** Testing with partitions *************** \n")

print("Testing with partitions and depth = 2 \n")

accuracys = partition_validation(dataset_torch, 2, 10)
accurracy_df = pandas.DataFrame(accuracys, columns = ['Accuracy of partition'])
print(accurracy_df)
print("\n The average accuracy is :", accurracy_df['Accuracy of partition'].mean())
print("\n The standard deviation is :", accurracy_df['Accuracy of partition'].std())


print("\n Testing with partitions and depth = 3 \n")

accuracys = partition_validation(dataset_torch, 3, 10)
accurracy_df = pandas.DataFrame(accuracys, columns = ['Accuracy of partition'])
print(accurracy_df)
print("\n The average accuracy is :", accurracy_df['Accuracy of partition'].mean())
print("\n The standard deviation is :", accurracy_df['Accuracy of partition'].std())
```

```
************** Testing with partitions ***************

Testing with partitions and depth = 2

   Accuracy of partition
0               0.641253
1               0.618788
2               0.633084
3               0.651464
4               0.638530
5               0.631042
6               0.639891
7               0.653506
8               0.632403
9               0.625596

 The average accuracy is : 0.6365554799183119

 The standard deviation is : 0.010760737115253691

 Testing with partitions and depth = 3

   Accuracy of partition
0               0.646018
1               0.626957
2               0.633084
3               0.651464
4               0.645337
5               0.634445
6               0.639891
7               0.658952
8               0.632403
9               0.629680

 The average accuracy is : 0.6398230088495576

 The standard deviation is : 0.010370883947508886
```

3. Random Forest

Utilizando los Random Forest podremos ver si nuestro modelo da mejores resultados al correr distintos arboles y poner las predicciones a votación para encontrar la clase que mas se ajuste, para esto implementamos las siguientes funciones:

3.1. Train Random Forest

```
    def generate_random_forest(self, partitions):
        for train_index, test_index in partitions.split(self.original_data):
            self.random_data_subsets.append(self.original_data[train_index])
        idx = 0
        for data_subset in self.random_data_subsets:
            treex = train_CART(data_subset, "Forest_CART" + str(idx) + ".xml", self.depth_of_trees, 2)
            self.list_of_carts.append(treex)
            idx += 1


def train_random_forest(testset_torch, k_partitions, depth_per_tree):
    forest = Random_Forest(testset_torch, k_partitions, depth_per_tree)
    kf = KFold(n_splits=k_partitions, shuffle=True)
    forest.generate_random_forest(kf)
    return forest
```

Figura 9: Train Random Forest

3.2. Evaluación Random Forest

```
def evaluate_random_forest(self, input_torch):
    predicted_categories = []
    for cart in self.list_of_carts:
        predicted_categories.append(cart.evaluate_input(input_torch))
    #Returns the most common element from the prediction of all trees
    return max(set(predicted_categories), key=predicted_categories.count)
```

Figura 10: Eval Random Forest

3.3. Test Random Forest

```
def test_random_forest(self, testset_torch):
    correct_observations = 0
    for observation in testset_torch:
        predicted = self.evaluate_random_forest(observation)
        classy = observation[-1].item()
        if predicted == classy:
            correct_observations += 1
    return correct_observations / testset_torch.shape[0]
```

```
#RANDOM FOREST

print("\n GENERATING RANDOM FOREST \n")

rf_model = train_random_forest(dataset_torch, 5, 3)
forest_acc = rf_model.test_random_forest(dataset_torch)

print("Accuracy of the random forest is:", forest_acc)
```

```
 GENERATING RANDOM FOREST

Accuracy of the random forest is: 0.6530445443400081
```

3.4. Evaluación de multiples Random Forest

```python
print("\n***** EVALUATING RANDOM FOREST ******\n")

accuracys = []
print("\n GENERATING RANDOM FOREST OF 3 CARTS - 10 RUNS \n")
for i in range(10):
    rf_model = train_random_forest(dataset_torch, 3, 3)
    forest_acc = rf_model.test_random_forest(dataset_torch)
    accuracys.append([forest_acc])
forest_accurracy_df = pandas.DataFrame(accuracys, columns = ['Accuracy of run'])
print("Accuracys of 10 runs with random forest of 3 CARTS \n", forest_accurracy_df)
print("\n The average accuracy is :", forest_accurracy_df['Accuracy of run'].mean())
print("\n The standard deviation is :", forest_accurracy_df['Accuracy of run'].std())

print("\n GENERATING RANDOM FOREST OF 5 CARTS - 10 RUNS \n")

accuracys = []
for i in range(10):
    rf_model = train_random_forest(dataset_torch, 5, 3)
    forest_acc = rf_model.test_random_forest(dataset_torch)
    accuracys.append([forest_acc])
forest_accurracy_df = pandas.DataFrame(accuracys, columns = ['Accuracy of run'])
print("Accuracys of 10 runs with random forest of 5 CARTS \n", forest_accurracy_df)
print("\n The average accuracy is :", forest_accurracy_df['Accuracy of run'].mean())
print("\n The standard deviation is :", forest_accurracy_df['Accuracy of run'].std())
```

```
***** EVALUATING RANDOM FOREST ******


 GENERATING RANDOM FOREST OF 3 CARTS - 10 RUNS

Accuracys of 10 runs with random forest of 3 CARTS
    Accuracy of run
0          0.642215
1          0.654884
2          0.656110
3          0.654475
4          0.653862
5          0.652023
6          0.653862
7          0.646915
8          0.650593
9          0.653862

 The average accuracy is : 0.6518798528810789

 The standard deviation is : 0.004283989628306299

 GENERATING RANDOM FOREST OF 5 CARTS - 10 RUNS

Accuracys of 10 runs with random forest of 5 CARTS
    Accuracy of run
0          0.653862
1          0.654475
2          0.656110
3          0.653249
4          0.648141
5          0.656110
6          0.653862
7          0.654884
8          0.653862
9          0.653862

 The average accuracy is : 0.6538414384961178

 The standard deviation is : 0.0022268107991692547
```

Figura 11: Resultados finales de la evaluacion con multiples Random Forest

4. Conclusión

En la implementación de los arboles de decisión para clasificación pudimos aprender la importancia de entender y abstraer el dataset para utilizar solamente lo necesario y que añade valor, además pudimos observar como para este ejemplo ene specifico los resultados no cambian tanto entre una implementación de un solo árbol contra un Random Forest, sin embargo pueden existir otros datasets donde si se experimentaría una mejora significativa, además se observa una pequeña mejoría entre el uso de más CARTs en el Random Forest tanto en el promedio como en la desviación estándar.

Referencias

[1] Satoshi Usami Timothy Hayes. Using classification and regression trees (cart) and random forests to analyze attrition: Results from two simulations. 2015.