Côme Sören Noé NZUZI MANIEMA
240 AMM 031

RAE 411

Fifth Practical Exercise Report

Cybersecurity with ML

Côme Sören Noé NZUZI MANIEMA
240 AMM 031

Côme Sören Noé NZUZI MANIEMA
240 AMM 031

Table of contents:

Côme Sören Noé NZUZI MANIEMA
240 AMM 031

Introduction:

In this work, we have focused on exploring the practical application of machine learning techniques, particularly decision tree algorithms, to solve classification problems. In Task 1, we implemented decision tree algorithms. For Task 2, we delved deeper into understanding the output of these machine learning algorithms by interpreting decision trees, confusion matrices, and other evaluation metrics. This systematic approach allowed us to assess the strengths and limitations of both algorithms, providing a comprehensive understanding of their application in real-world scenarios.

Côme Sören Noé NZUZI MANIEMA
240 AMM 031

Task 1

In this script, we implement a Binary Search Tree (BST) data structure in Python, featuring essential operations like insertion, search, deletion, and traversal, along with a method for visualizing the tree structure.

Class construction

The BST is built around two main classes: the Node class, which represents individual nodes in the tree, and the BinarySearchTree class, which manages the overall structure and operations.

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
```

```
# Definition of the BinarySearchTree class
class BinarySearchTree:
    def __init__(self):
        self.root = None

#Method to insert a key into the tree
    def insert(self, key):
        self.root = self._insert(self.root, key)

# Private method for recursive insertion
    def _insert(self, root, key):
        if root is None:
            return Node (key)
        if key < root.key:
            root.left = self._insert(root.left, key)
        else:
            root.right = self._insert(root.right, key)
        return root

# Method to search for a key in the tree
    def search(self, key):
        return self._search(self.root, key)
```

```
# Private method for recursive search
    def _search(self, root, key):
        if root is None:
            return False
        if root.key == key:
            return True
        if key < root.key:
            return self._search(root.left, key)
        return self._search(root.right, key)

# Method to delete a key from the tree
    def delete(self, key):
        self.root = self._delete(self.root, key)

# Private method for recursive deletion
    def _delete(self, root, key):
        if root is None:
            return root
        if key < root.key:
            root.left = self._delete(root.left, key)
        elif key > root.key:
            root.right = self._delete(root.right, key)
        else:
            if root.left is None:
                return root.right
            elif root.right is None:
                return root.left
            root.key = self._get_min_value(root.right)
            root.right = self._delete(root.right, root.key)
        return root
```

```python
# Private method to get the minimum value in a tree
    def _get_min_value(self, root):
        while root.left is not None:
            root = root.left
        return root.key

# Method for inorder traversal of the tree
    def inorder_traversal(self):
        result = []
        self._inorder_traversal(self.root, result)
        return result

# Private method for recursive inorder traversal
    def _inorder_traversal(self, root, result):
        if root is not None:
            self._inorder_traversal(root.left, result)
            result.append(root.key)
            self._inorder_traversal(root.right, result)
```

Thus, the Node class defines a node with a key and pointers to its left and right children. The BinarySearchTree class includes methods for tree manipulation: Insertion, Search, Deletion and Traversal. The recursive "_insert"method places nodes at their correct position, ensuring the BST property is maintained. Then, the _search method checks for the existence of a given key by traversing the tree recursively. The "_delete" method handles removal of nodes while preserving the BST structure, addressing edge cases such as leaf nodes, nodes with one child, and nodes with two children. So, the "inorder_traversal" method retrieves the tree's keys in sorted order using recursion.

Class methods

For visual representation, the plot_tree method leverages the NetworkX and Matplotlib libraries. It constructs a directed graph with nodes and edges corresponding to the tree's structure.

```python
# Method to visualize the tree
    def plot_tree(self, title, color = "skyblue"):
        G = nx.DiGraph()
        pos = self._build_graph(G, self.root)
        plt.title(title) # put this before drawing the graph because it will be overwritten by the graph otherwise
        nx.draw(G, pos, with_labels=True, arrows=False, node_size=800, node_color=color, font_size=10)
        plt.show()
```

The private _build_graph method recursively positions nodes for a layered visualization, enabling clear insight into the tree's architecture.

```python
# Private method to recursively build the graph for visualization
    def _build_graph(self, G, node,pos=None, x=0, y=0, layer=1):
        if pos is None:
            pos = {node.key: (x,y)}
        else:
            pos[node.key] = (x,y)

        if node.left is not None:
            left_pos= (x-1/(2**layer), y-1)
            G.add_edge(node.key, node.left.key)
            self._build_graph(G, node.left, pos, x-1/(2**layer), y-1, layer+1)

        if node.right is not None:
            right_pos = (x+1/(2**layer), y-1)
            G.add_edge(node.key, node.right.key)
            self._build_graph(G, node.right, pos, x+1/(2**layer), y-1, layer+1)
        return pos
```

Therefore, this script forms the foundation for creating and manipulating binary search trees, providing both functional and graphical outputs for enhanced understanding of each operation.

Côme Sören Noé NZUZI MANIEMA
240 AMM 031

Tree creation

With the main code we demonstrate the functionality of the Binary Search Tree (BST) implementation through the creation and manipulation of three distinct trees, labeled A, B, and C.

```
a = [49, 38, 65, 97, 60, 76, 13, 27, 5, 1]
```

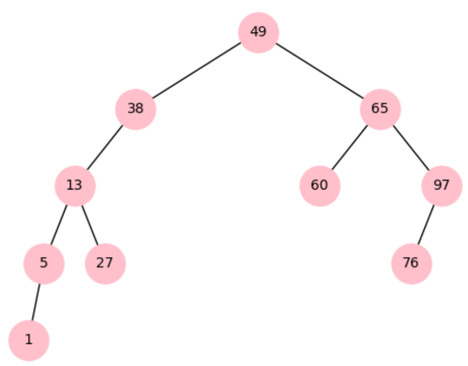
```
b = [149, 38, 65, 197, 60, 176, 13, 217, 5, 11]
```

```
c = [49, 38, 65, 97, 64, 76, 13, 77, 5, 1, 55, 50, 24]
```

For each tree, the operations include insertion, deletion, search, and graphical visualization. For instance, let's delve into the operations done on tree A.

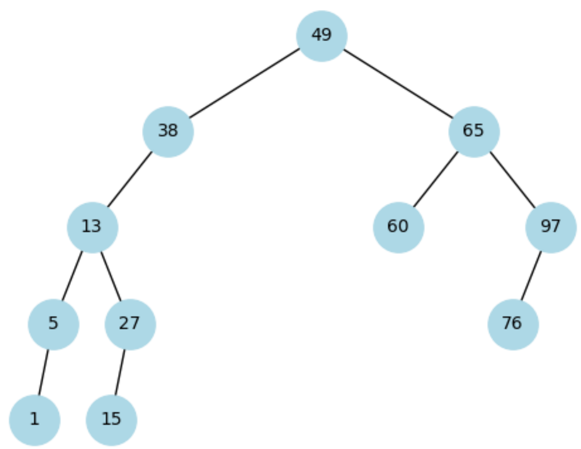
Tree A is initialized using a list of values [49, 38, 65, 97, 60, 76, 13, 27, 5, 1].

Tree A



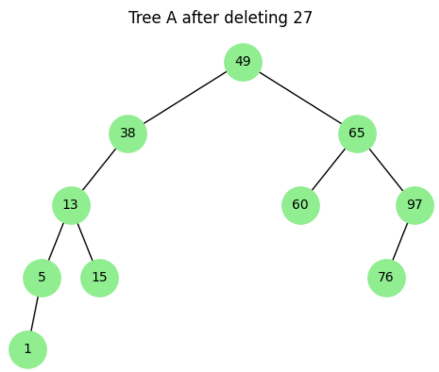
After building the tree, the value 15 is inserted, and the structure is visualized to reflect this change.

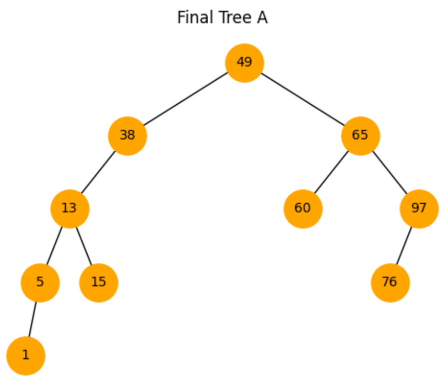
Tree A after inserting 15



Côme Sören Noé NZUZI MANIEMA
240 AMM 031

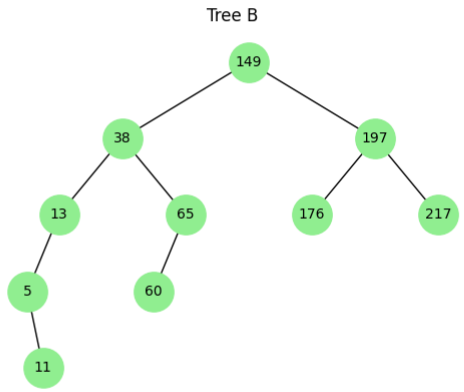
Next, the value 27 is deleted, followed by another visualization of the updated tree. The code then performs searches for the values 65 (found) and 2 (not found). The final state of Tree A is displayed graphically.

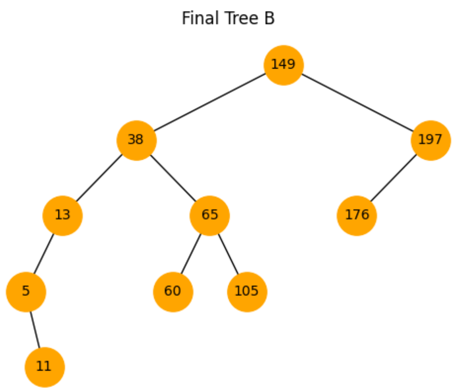





```
Searching for 65 in Tree A: True
Searching for 2 in Tree A: False
Tree A completed.
```

The same work is done on tree B, but the values are different.



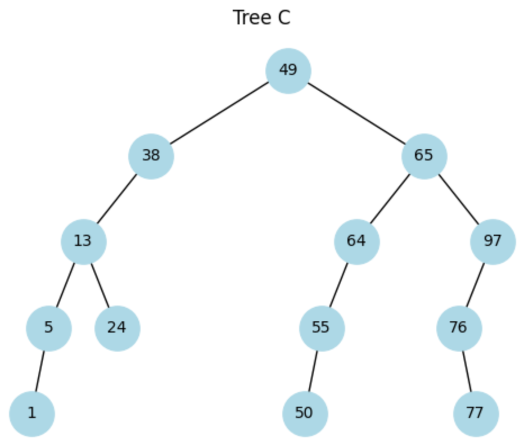


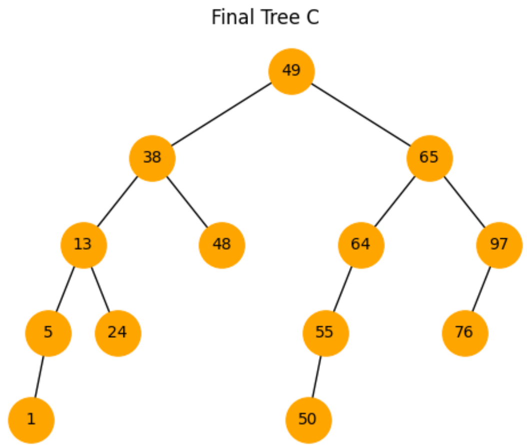

```
Tree B in progress...

Searching for 10 in Tree B: False
Searching for 5 in Tree B: True
Tree B completed.
```

Also, Search operations are performed for the values 10 (not found) and 5 (found).

Côme Sören Noé NZUZI MANIEMA
240 AMM 031

Then, for tree C is it also the same but values change.





However, for each tree, the “plot_tree” method is utilized to provide a graphical representation, enhancing the clarity of changes made during each step.

To conclude, this practical task illustrates the fundamental operations of BSTs and their impact on tree structure, with graphical outputs complementing the textual results.

Côme Sören Noé NZUZI MANIEMA
240 AMM 031

Task 2

In this second script, we are going to implement a custom decision tree classifier designed to support two distinct algorithms, **CART** (Classification and Regression Tree) and **ID3** (Iterative Dichotomiser 3), for supervised classification tasks. So, the classifier includes methods for fitting a model to the dataset, predicting labels for new data, and evaluating splits based on the chosen algorithm.

Firstly, the DecisionTreeClassifierManual class initializes with two parameters. The "max_depth" defines the maximum depth of the tree to prevent overfitting. If set to None, the tree grows until leaves are pure. Then, the "algorithm" parameter specifies whether to use CART (based on Gini impurity) or ID3 (based on information gain) as the splitting criterion.

Class construction

```
class DecisionTreeClassifierManual:
    def __init__(self, max_depth=None, algorithm='CART'):
        """
        Parameters:
        - max_depth: Maximum depth of the tree. If None, the tree will grow until all leaves are pure.
        - algorithm: 'CART' (Gini Impurity) or 'ID3' (Information Gain)
        """
        self.max_depth = max_depth
        self.algorithm = algorithm
        self.tree = None
```

Class methods

Let's explore the splitting criteria. The Gini Impurity method (_gini) calculates the impurity of a dataset by measuring the likelihood of incorrect classification.

```
    def _gini(self, y):
        """Calculate Gini Impurity"""
        classes, counts = np.unique(y, return_counts=True)
        impurity = 1 - np.sum((count / len(y))**2 for count in counts)
        return impurity
```

Then, the Entropy method (_entropy) measures the level of disorder in the dataset, forming the basis for calculating Information Gain (_information_gain). The information gain is the reduction in entropy achieved after splitting the dataset.

```
    def _entropy(self,y):
        """Calculate Entropy"""
        classes, counts = np.unique(y, return_counts=True)
        probabilities = counts / len(y)
        return -np.sum(probabilities * np.log2(probabilities + 1e-9))
```

Côme Sören Noé NZUZI MANIEMA
240 AMM 031

The _best_split method iterates through all features and unique thresholds to find the optimal split. In other words, for CART, it minimizes the weighted Gini impurity and for ID3, it maximizes the information gain.

```python
    def _best_split(self, X, y):
        """Find the best split based on the chosen algorithm (CART or ID3)"""
        best_feature=None
        best_threshold=None
        best_metric = -float("inf") if self.algorithm =="ID3" else float("inf")
        n_features = X.shape[1]

        for feature_index in range(n_features):
            thresholds = np.unique(X[:, feature_index])
            for threshold in thresholds:
                _, _, y_left, y_right = self._split(X, y, feature_index, threshold)
                if len(y_left) == 0 or len(y_right) == 0:
                    continue

                if self.algorithm == 'CART':
                    gini_left = self._gini(y_left)
                    gini_right = self._gini(y_right)
                    weighted_gini = (len(y_left) / len(y)) * gini_left + (len(y_right) / len(y)) * gini_right
                    metric = weighted_gini
                    if metric < best_metric:
                        best_metric = metric
                        best_feature = feature_index
                        best_threshold = threshold

                    elif self.algorithm == 'ID3':
                        gain = self._information_gain(y, y_left, y_right)
                        if gain > best_metric:
                            best_metric = gain
                            best_feature = feature_index
                            best_threshold = threshold

        return best_feature, best_threshold
```

The _split method divides the dataset into left and right branches based on the feature index and threshold. It ensures the dataset is separated into subsets for recursive tree construction.

```python
    def _split(self, X, y, feature_index, threshold):
        """Split the dataset into left and right branches"""
        left_indices = X[:, feature_index] < threshold
        right_indices = ~left_indices
        return X[left_indices], X[right_indices], y[left_indices], y[right_indices]
```

The _build_tree method recursively constructs the decision tree. It terminates under the following conditions. All data must in the node belongs to the same class. The dataset must be empty and the specified maximum depth is reached. Non-terminal nodes store the feature index, threshold, and child nodes, while terminal nodes store the predicted class label.

```python
def _build_tree(self, X, y, depth):
    """Recursively build the decision tree"""
    if len(np.unique(y)) == 1 or len(y) == 0 or (self.max_depth is not None and depth >= self.max_depth):
        return {"type": "leaf", "class": np.bincount(y).argmax()}

    feature, threshold = self._best_split(X, y)
    if feature is None or threshold is None:
        return {"type": "leaf", "class": np.bincount(y).argmax()}

    X_left, X_right, y_left, y_right = self._split(X, y, feature, threshold)

    return {
        "type": "node",
        "feature_index": feature,
        "threshold": threshold,
        "left": self._build_tree(X_left, y_left, depth + 1),
        "right": self._build_tree(X_right, y_right, depth + 1)
    }
```

The _traverse_tree method traverses the tree for each data point in the input, following branches based on feature values until a leaf node is reached. The class stored in the leaf node is returned as the prediction.

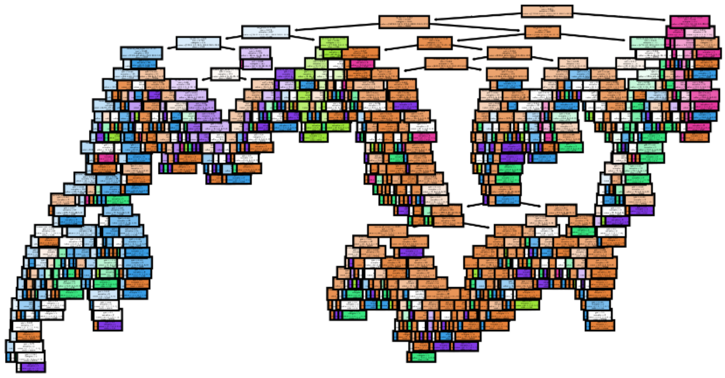
```python
    def _traverse_tree(self, x, node):
        """Traverse the tree to make predictions"""
        if node["type"] == "leaf":
            return node["class"]

        if x[node["feature_index"]] < node["threshold"]:
            return self._traverse_tree(x, node["left"])
        else:
            return self._traverse_tree(x, node["right"])
```

This implementation provides flexibility by allowing us to compare the performance of CART and ID3 algorithms on the same dataset. By incorporating both Gini impurity and information gain as splitting metrics, the script highlights the differences in decision tree behavior under varying splitting criteria.

Output interpretations

Then, we have the decision tree diagram which is a visualization of the model's structure and how it makes decisions.

Decision Tree trained on all features



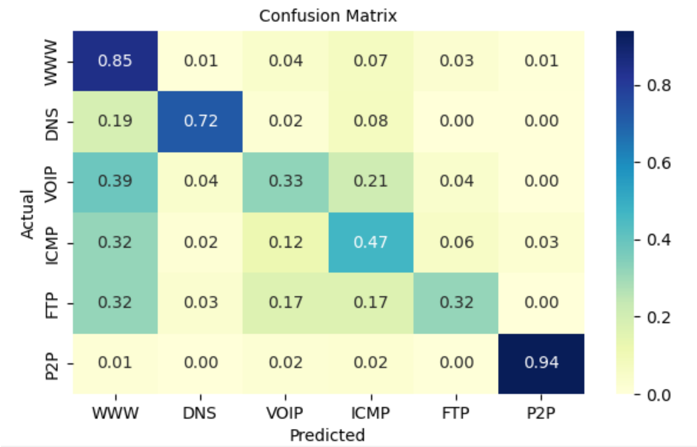
As mentioned in the script, the decision tree diagram has a depth of 6, this means that it can make up to 6 sequential decisions before reaching a conclusion about the category. Each node represents a decision based on one of the features. The leaf nodes at the bottom are the final predictions made by the tree. Each leaf is associated with a predicted class (e.g., WWW, DNS, etc.).

Côme Sören Noé NZUZI MANIEMA
240 AMM 031

The confusion matrix provides a detailed breakdown of the model's performance for each class. Each cell shows the proportion of actual instances of a class (rows) predicted as each class (columns). For example:

Row WWW → Column WWW: 0.85

85% of the WWW samples were correctly predicted.



Therefore, let's talk about class-specific performance. We have high accuracy for P2P (94%). That means that the model performs exceptionally well for P2P traffic. Nevertheless, according to the dataset, we have poor accuracy for VOIP (33%) and ICMP (47%). VOIP and ICMP categories are often misclassified, possibly because their features overlap with other categories. In addition, DNS has a 19% misclassification rate to WWW, which could suggest some shared traits between these classes.

To conclude, ID3 (Iterative Dichotomizer 3) and CART (Classification and Regression Trees) are decision tree algorithms with distinct approaches. ID3 uses Information Gain (based on entropy) to split features, making it effective for categorical data but less efficient for continuous features, as it requires prior discretization. It is limited to classification tasks and lacks built-in pruning, making it more prone to overfitting unless manually pruned. In contrast, CART uses Gini Impurity for classification or Mean Squared Error for regression, handling both continuous and categorical features natively. It supports classification and regression tasks and includes built-in pruning mechanisms, which improve generalization and reduce overfitting. CART is computationally more efficient and better suited for larger, more complex datasets, while ID3 is more intuitive and simpler for small, categorical datasets.

Côme Sören Noé NZUZI MANIEMA
240 AMM 031

Conclusion:

Through the implementation and analysis of ID3 and CART, we have gained valuable knowledge into the design and performance of decision tree algorithms. We observed how their unique methodologies, such as ID3's use of Information Gain and CART's reliance on Gini Impurity influence their suitability for specific data types and tasks. By interpreting their outputs and evaluating their performance using confusion matrices and other metrics, we identified their strengths and potential drawbacks. This hands-on experience not only deepened our understanding of decision tree algorithms but also enhanced our ability to apply machine learning techniques effectively in diverse contexts.