[ProjectCode: TCDT]

Differentiated Thyroid Cancer Recurrence using Decision Tree Decision Tree based Learning Mode

GROUP NO. - 32

(run: python -W ignore main.py)
Github Link:Link

To develop a decision tree learning model for predicting the recurrence of well-differentiated thyroid cancer, you can follow these steps:

Data Preprocessing: Download the dataset from the provided link and preprocess it as needed. This may include handling missing values, encoding categorical variables, and splitting the dataset into training and validation sets.

Implementing Decision Tree Algorithm: using entropy-based information gain for attribute selection.

```
class DecisionTree:
```

This nested class represents a node in the decision tree. It has attributes attribute (the attribute used for splitting at this node), final_class (the predicted class if this node is a leaf), and next (a dictionary mapping attribute values to child nodes).

```
    class node:
        def __init__(self, attribute=None, final_class=None) -> None:
            self.attribute = attribute
            self.final_class = final_class
            self.next = {}
```

The constructor initializes the max_depth (maximum depth of the tree) and depth (current depth of the tree) attributes.

```
    def __init__(self, max_depth=None) -> None:
        self.max_depth = max_depth
        self.depth = 0
```

This method is used to train the decision tree model. It calls the build method to recursively build the tree based on the training data (X_train and y_train). The attributes_taken parameter is a dictionary that keeps track of the attributes already used for splitting at each level of the tree. The current_depth parameter tracks the current depth of the tree during the recursive build process.

```
    def fit(self, X_train, y_train):
```

```
        self.root = self.build(X_train, y_train, attributes_taken={}, 
current_depth=1)
        print(f"Done training \nDepth of tree = {self.depth}")
```

This method takes a test dataset X_test and predicts the class labels for each instance in X_test using the trained decision tree. It iterates over each row in X_test and calls the get_class method to traverse the tree and make predictions.

```
    def predict(self, X_test):
        y_pred = pd.Series(range(X_test.shape[0]), index=X_test.index)

        for index, row in X_test.iterrows():
            y_pred[index] = self.get_class(row, self.root)

        return y_pred

    def entropy(self, X_train, y_train, attributes_taken):
        y_count = { key : 0 for key in y_train.unique() }

        for index, row in X_train.iterrows():
            correct_row = True

            for attr, val in attributes_taken.items():
                if row[attr] != val:
                    correct_row = False

            if not correct_row:
                continue

            y_count[y_train[index]] += 1

        tot = sum(value for value in y_count.values())

        # if any(value==sum for value in y_count.values()):
        #     for key in y_count:
        #         if y_count[key] == tot:
        #             return key
        # else:
        entropy = 0.0
        for value in y_count.values():
            if value == 0:
                continue
            entropy -= (value/tot)*math.log2(value/tot)

        return entropy
```

This method calculates the information gain of an attribute by splitting the dataset (X_train) based on that attribute. It uses the initial entropy of the dataset (initial_entropy) and the entropy of the dataset after the split to calculate the information gain.

```
    def information_gain(self, X_train, y_train, attributes_taken, attr,
initial_entropy):
        attr_values_cnt = { key : 0 for key in X_train[attr].unique() }
        attr_values_entropies = { key : 0 for key in X_train[attr].unique() }

        tot_cnt = 0

        for attr_value in attr_values_cnt:
            attributes_taken[attr] = attr_value

            attr_values_entropies[attr_value] = self.entropy(X_train, y_train,
attributes_taken)

            cnt = 0
            for index, row in X_train.iterrows():
                correct_row = True

                for attribute_taken, value in attributes_taken.items():
                    if row[attribute_taken] != value:
                        correct_row = False

                if not correct_row:
                    continue

                #Correct row
                cnt += 1

            # store the Sv
            attr_values_cnt[attr_value] = cnt
            tot_cnt += cnt

            # remove from attributes taken attr
            attributes_taken.pop(attr)

        new_entropy = 0
        for attr_value in attr_values_cnt:
            new_entropy += (attr_values_cnt[attr_value]/tot_cnt) *
attr_values_entropies[attr_value]

        # print(f"this is new entropy {new_entropy}")
```

```
        return initial_entropy - new_entropy
```

This method is called when a leaf node is to be created. It calculates the majority class in the current subset of the dataset and returns a leaf node with the majority class as the final class.

```
    def build_leaf(self, X_train, y_train, attributes_taken):
        # build leaf node here, check majority of current classes,
        class_cnt = { key : 0 for key in y_train.unique() }
        for index, row in X_train.iterrows():
            correct_row = True

            for attribute_taken, value in attributes_taken.items():
                if row[attribute_taken] != value:
                    correct_row = False

            if not correct_row:
                continue

            class_cnt[y_train[index]] += 1

        # get key max value in class_cnt
        choosen_class = max(class_cnt, key=class_cnt.get)

        # print(choosen_class)

        #create a node object
        return self.node(final_class=choosen_class)
```

This is the main method for recursively building the decision tree. It selects the best attribute to split based on the information gain, creates a new node for that attribute, and recursively builds the child nodes for each value of the selected attribute. If the maximum depth (max_depth) is reached or if the dataset is pure (i.e., all instances belong to the same class), it creates a leaf node.

```
    def build(self, X_train, y_train, attributes_taken, current_depth):
        initial_entropy = self.entropy(X_train, y_train, attributes_taken)
        self.depth = max(self.depth, current_depth)    #update current depth of
tree

        if not self.max_depth is None and self.max_depth == current_depth:
            '''
                Prune the branch
                Get the majority class
            '''
            leaf = self.build_leaf(X_train, y_train, attributes_taken)
```

```
            return leaf

        elif initial_entropy == 0.0:
            #Make a leaf node
            # print(attributes_taken, end=" ")

            leaf = self.build_leaf(X_train, y_train, attributes_taken)
            return leaf
        else:
            igs = {}
            for attr in X_train.columns:
                if attr in attributes_taken:
                    continue

                igs[attr] = self.information_gain(X_train, y_train,
attributes_taken, attr, initial_entropy)

            # print(igs)
            choosen_attr = max(igs, key=igs.get)

            # print(f"We have choose {choosen_attr} with IG
{igs[choosen_attr]}")
            # new_node = self.node(attribute=choosen_one)

            if len(igs) == 0:
                # Exhausted all atributes
                leaf = self.build_leaf(X_train, y_train, attributes_taken)
                return leaf
            else:
                # move to childs
                new_node = self.node(attribute=choosen_attr)
                new_node.next = { key : None for key in
X_train[choosen_attr].unique() }

                for value in new_node.next:
                    attributes_taken[choosen_attr] = value
                    new_node.next[value] = self.build(X_train, y_train,
attributes_taken, current_depth+1)
                    attributes_taken.pop(choosen_attr)

                return new_node
```

This method traverses the decision tree to predict the class label for a given instance (`row`) by following the appropriate path based on the attribute values of the instance.

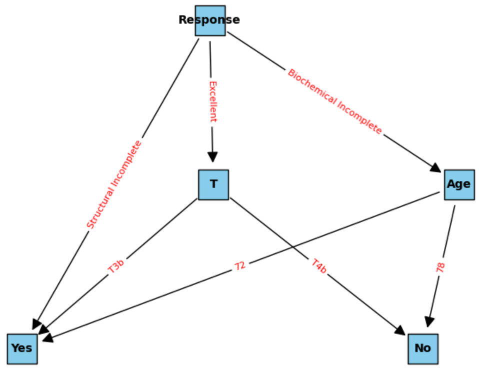
```
    def get_class(self, row, current_node):
        # some thing wrong happened
        if not current_node:
            return None

        if current_node.final_class:
            return current_node.final_class
        else:
            # print(current_node.attribute, row[current_node.attribute],
row[current_node.attribute] in current_node.next)
            if not row[current_node.attribute] in current_node.next:
                return None
            return self.get_class(row,
current_node.next[row[current_node.attribute]])
```

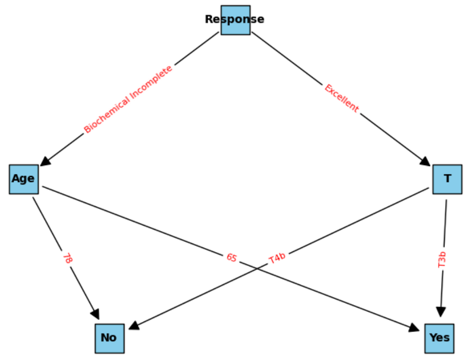
This method calculates the accuracy of the model by comparing the predicted class labels (y_pred) with the actual class labels (y_test). It returns the proportion of correctly predicted instances.

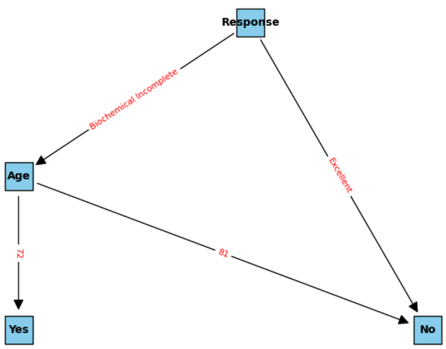
```
    def accuracy(self, y_pred, y_test):
        same = 0
        diff = 0
        for index in y_test.index:
            if y_pred[index] == y_test[index]:
                same += 1
            else:
                diff += 1
        return same/(same+diff)

    def draw_tree(self, filepath: str):
        G = nx.DiGraph()

        def add_edges(node):
            for edge, child_node in node.next.items():
                G.add_edge(node.attribute if node.attribute is not None else
node.final_class,
                        child_node.attribute if child_node.attribute is not
None else child_node.final_class,
                        label=edge)
                add_edges(child_node)

        add_edges(self.root)

        pos = nx.nx_agraph.graphviz_layout(G, prog='dot', args='-Gnodesep=1
-Granksep=1 -Goverlap=false -Gsplines=line -Gmodel=subset -Gstrict=false
-Grankdir=TB')
```

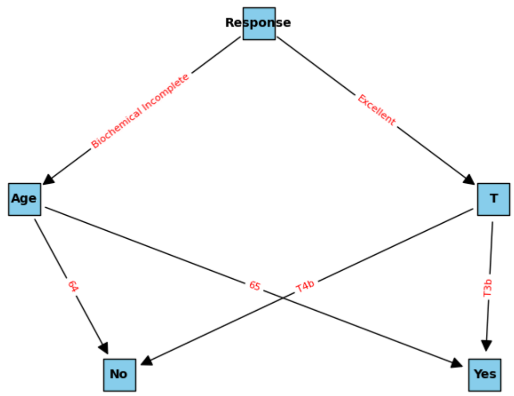
```
        # Customize node shape and size
        node_shape = 's'  # Square
        node_size = 700

        #clear the plot
        plt.clf()

        # Visualize the tree with straight edges and square nodes
        nx.draw(G, pos, with_labels=True, node_size=node_size,
node_color="skyblue", font_size=10, font_color="black", font_weight="bold",
node_shape=node_shape, arrowsize=20, edgecolors='black', linewidths=1, width=1)

        # Draw edge labels
        edge_labels = nx.get_edge_attributes(G, 'label')
        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
font_color='red', font_size=8)

        # save the plot
        os.makedirs(os.path.dirname(filepath), exist_ok=True)
        plt.savefig(filepath)
```

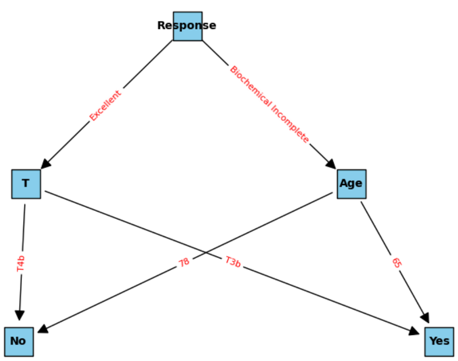
Trees in different folds:











Implement tree pruning using Reduced Error Pruning. Create a function to run your model using different values for the maximum depth (max depth) parameter (e.g., from 1 to 25) and visualize the results to see how the accuracy differs for each criterion value (i.e., for Gini index and information gain).

Compare the accuracy of the pruned trees at different depths and select the best one based on the validation data.

Comparison with sci-kit-learn:

```
clf = DecisionTreeClassifier()
y_pred = clf.fit(X_train, y_train).predict(X_test)
```

The code uses 5-fold cross-validation (KFold) to split the data into training and testing sets for each fold.
For each fold, a decision tree classifier (DecisionTreeClassifier from sklearn.tree) is instantiated.
The classifier is trained on the training data (X_train, y_train) using the fit method.
The trained classifier is then used to predict the labels for the testing data (X_test), and these predictions (y_pred) are evaluated against the actual labels (y_test) using classification_report.
The classification_report function provides metrics such as precision, recall, F1-score, and support for each class (in this case, binary classes 'No' and 'Yes').

Pruning(Reduced error pruning)
Sure! Let's break down the `prune_subtree_rec` method step by step:

Pruning Logic:
- The method checks if the `node` is a leaf node by checking if `node.attribute` is `None`. If it is a leaf node, the method returns, as there is no need to prune a leaf node.
- If the `node` is not a leaf node, it iterates over each attribute value and its corresponding child node in `node.next`.
  - For each attribute value, it extracts a subset of the validation set `X_val` where `X_val[node.attribute]` equals the current attribute value, and the corresponding subset of labels `y_val_sub`.
  - If the subset of labels `y_val_sub` is empty (i.e., no samples with this attribute value in the validation set), it skips pruning for this attribute value.
  - Otherwise, it recursively calls `prune_subtree_rec` on the child node with the subset of validation set and labels.

Calculation of Validation Accuracy:
- Before pruning (`val_accuracy_before_pruning`): It calculates the validation accuracy of the decision tree on the entire validation set (`X_val_stat`) using the current state of the decision tree.
- After pruning (`val_accuracy_after_pruning`): It temporarily prunes the current node by setting `node.attribute` to `None` and `node.final_class` to the most frequent class in `y_val`. It then calculates the validation accuracy again.
- If the accuracy after pruning decreases by less than `0.00001` compared to before pruning, it reverts the pruning by restoring the original `node.attribute` and `node.final_class`. This is a form of post-pruning to prevent overfitting.

Final Decision:
- If the accuracy after pruning is not significantly lower than before pruning, the method keeps the node as a leaf node by clearing its `next` pointers (making it a terminal node).
- If the accuracy after pruning is significantly lower, the method reverts the node back to its original state (non-leaf node).

Output:

USING ID3 IMPLEMENTATION

TRAINING FOR FOLD 0
Done training
Depth of tree = 4
fold 0 accuracy= 93.5064935064935
No 59
Yes 18
Name: count, dtype: int64
Total number of nodes = 152

	precision	recall	f1-score	support
No	0.95	0.97	0.96	58
Yes	0.89	0.84	0.86	19
accuracy			0.94	77
macro avg	0.92	0.90	0.91	77
weighted avg	0.93	0.94	0.93	77

PRUNING ON FOLD 0

Accuracy on validation (before pruning) = 0.935064935064935
Total Nodes (before pruning) = 152
Accuracy on validation (after pruning) = 0.935064935064935
Total Nodes (after pruning) = 152
Accurary on training data (after pruning) = 1.0

TRAINING FOR FOLD 1
Done training
Depth of tree = 4
fold 1 accuracy= 81.81818181818183
No 62
Yes 15
Name: count, dtype: int64
Total number of nodes = 205

	precision	recall	f1-score	support
No	0.81	0.96	0.88	52
Yes	0.87	0.52	0.65	25

	precision	recall	f1-score	support
accuracy			0.82	77
macro avg	0.84	0.74	0.76	77
weighted avg	0.83	0.82	0.80	77

PRUNING ON FOLD 1
Accuracy on validation (before pruning) = 0.8181818181818182
Total Nodes (before pruning) = 205
Accuracy on validation (after pruning) = 0.948051948051948
Total Nodes (after pruning) = 79
Accurary on training data (after pruning) = 0.9738562091503268

TRAINING FOR FOLD 2
Done training
Depth of tree = 4
fold 2 accuracy= 79.22077922077922
No 68
Yes 9
Name: count, dtype: int64
Total number of nodes = 196

	precision	recall	f1-score	support
No	0.78	0.98	0.87	54
Yes	0.89	0.35	0.50	23
accuracy			0.79	77
macro avg	0.83	0.66	0.68	77
weighted avg	0.81	0.79	0.76	77

PRUNING ON FOLD 2
Accuracy on validation (before pruning) = 0.7922077922077922
Total Nodes (before pruning) = 196
Accuracy on validation (after pruning) = 0.974025974025974
Total Nodes (after pruning) = 5
Accurary on training data (after pruning) = 0.934640522875817

TRAINING FOR FOLD 3
Done training
Depth of tree = 4
fold 3 accuracy= 89.47368421052632
No 60
Yes 16
Name: count, dtype: int64
Total number of nodes = 216

	precision	recall	f1-score	support

	precision	recall	f1-score	support
No	0.88	0.98	0.93	54
Yes	0.94	0.68	0.79	22
accuracy			0.89	76
macro avg	0.91	0.83	0.86	76
weighted avg	0.90	0.89	0.89	76

PRUNING ON FOLD 3
Accuracy on validation (before pruning) = 0.8947368421052632
Total Nodes (before pruning) = 216
Accuracy on validation (after pruning) = 0.9736842105263158
Total Nodes (after pruning) = 85
Accurary on training data (after pruning) = 0.9576547231270358

TRAINING FOR FOLD 4
Done training
Depth of tree = 4
fold 4 accuracy= 88.1578947368421
No 64
Yes 12
Name: count, dtype: int64
Total number of nodes = 216

	precision	recall	f1-score	support
No	0.88	0.98	0.93	57
Yes	0.92	0.58	0.71	19
accuracy			0.88	76
macro avg	0.90	0.78	0.82	76
weighted avg	0.89	0.88	0.87	76

PRUNING ON FOLD 4
Accuracy on validation (before pruning) = 0.881578947368421
Total Nodes (before pruning) = 216
Accuracy on validation (after pruning) = 0.9473684210526315
Total Nodes (after pruning) = 81
Accurary on training data (after pruning) = 0.9609120521172638

Average accuracy of 5 folds: 86.4354066985646

USING SKLEARN LIBRARY

TRAINING FOR FOLD 1

	precision	recall	f1-score	support
No	0.98	0.90	0.94	58
Yes	0.75	0.95	0.84	19
accuracy			0.91	77
macro avg	0.87	0.92	0.89	77
weighted avg	0.92	0.91	0.91	77

TRAINING FOR FOLD 2

	precision	recall	f1-score	support
No	0.96	0.96	0.96	52
Yes	0.92	0.92	0.92	25
accuracy			0.95	77
macro avg	0.94	0.94	0.94	77
weighted avg	0.95	0.95	0.95	77

TRAINING FOR FOLD 3

	precision	recall	f1-score	support
No	0.96	0.94	0.95	54
Yes	0.88	0.91	0.89	23
accuracy			0.94	77
macro avg	0.92	0.93	0.92	77
weighted avg	0.94	0.94	0.94	77

TRAINING FOR FOLD 4

	precision	recall	f1-score	support
No	0.96	0.98	0.97	54

	precision	recall	f1-score	support
Yes	0.95	0.91	0.93	22
accuracy			0.96	76
macro avg	0.96	0.95	0.95	76
weighted avg	0.96	0.96	0.96	76

TRAINING FOR FOLD 5

	precision	recall	f1-score	support
No	0.92	1.00	0.96	57
Yes	1.00	0.74	0.85	19
accuracy			0.93	76
macro avg	0.96	0.87	0.90	76
weighted avg	0.94	0.93	0.93	76