University of Bergen

Introduction to machine learning
INF264

Project 1: Implementing decision trees

Student :
Arthur Gautier

September 18, 2023

Contents

1 Implement a decision tree learning algorithm, from scratch

1.1 Node and Tree class

I choosed to use object-oriented programming to implement the decision tree. In this implementation the tree is composed of two classes *TreeNode* and *DecisionTree*. The first class contained several attribute detailed below :

TreeNode :

- *feature* : to select a feature for the split.
- *threshold* : threshold value to split the feature.
- *left* : left subtree (lower or equal values).
- *right* right subtree (upper or equal values).
- *info_gain* : information gain of the split.
- *value* : value predicted by the leaf (for leaf nodes).

DecisionTree :

- *root* : related to the root node of the tree
- *max_depth* : recursion stopping condition
- *min_samples_split* : recursion stopping condition

1.2 Entropy and information gain

We know that

$$H(X) = -\sum_{x\in\mathcal{X}} P(x)\log_2 P(x)$$

Based on this formula and using pandas array properties, we get this entropy method.

```python
def entropy(self, y):
    """Calculate the entropy of a variable Y"""
    p = y.value_counts() / y.shape[0]
    return -np.sum(p*np.log2(p))
```

Same thing for the information gain,

$$\mathrm{IG} = H(Y) - H(Y|X)$$

```
def information_gain(self, y, mask):
    """Calculate the information gain of a variable Y using a mask"""
    cardinal = mask.shape[0]
    m_true = sum(mask)
    m_false = cardinal - m_true

    if m_true == 0 or m_false == 0:
        ig = 0
    else:
        ig = self.entropy(y) - m_true/cardinal * self.entropy(y[mask])
            - m_false/cardinal* self.entropy(y[-mask])
        # y[mask] and y[-mask] are respectively two vectors containing
        the values of y where the mask
        is true and false.
    return ig
```

1.3 Building the tree

We can now calculate the Information Gain for all variables. Next, we choose the split that generates the highest Information Gain as our splitting criterion. To do that, we can search the split that maximize the information gain using the method *max_information_gain*. We can apply this function to every feature of the dataset and find where is the best split using *get_best_split*. This means we select the variable that provides the most valuable information for making decisions at each node of the tree. Lastly, we repeat this process recursively until at least one of the conditions set by hyperparameters of the algorithm is not fulfilled. All the tree is store in the root attribute of the decision tree class.

```
def build_tree(self, data, current_depth=0, depth=[]):
    """recursive function to build the tree"""
    # Print the current_depth
    if current_depth not in depth:
        depth.append(current_depth)
        print(current_depth)
    #  If all data points have the same label :
    y = data.iloc[:, -1]
    if y.nunique() == 1:
        leaf_value = self.calculate_leaf_value(y)
        #  return a leaf with that label
        return TreeNode(value=leaf_value)

    # Else, if all data points have identical feature values:
    all_rows_same = data.duplicated().sum() == 0
    if all_rows_same is True:
        leaf_value = self.calculate_leaf_value(y)
        return TreeNode(value=leaf_value)
```

```python

        # Else, choose a feature that maximizes the information gain :
        num_samples, num_features = data.shape
        if num_samples >= self.min_samples_split and current_depth <= self.max_depth:
            best_split_label, best_split_value,
                                        best_split_ig = self.get_best_split(y, data)
            # We check if the split is pure
            if best_split_ig is not None and best_split_ig > 0:
                data_left, data_right = self.make_split(best_split_label,
                                                        best_split_value, data)
                left_subtree = self.build_tree(data_left, current_depth+1, depth)
                right_subtree = self.build_tree(data_right, current_depth + 1, depth)
                return TreeNode(best_split_label, best_split_value, left_subtree,
                                right_subtree, best_split_ig)
            else:
                leaf_value = self.calculate_leaf_value(y)
                return TreeNode(value=leaf_value)
            # compute leaf nodel
        leaf_value = self.calculate_leaf_value(y)
        # return leaf node
        return TreeNode(value=leaf_value)
```

2 Add Gini index

$$G(X) = \sum_{x \in \mathcal{X}} P(x)(1 - P(x))$$

```python
def gini(self, y):
    """Gini index"""
    p = y.value_counts()/y.shape[0]
    return np.sum(p*(1-p))
```

We can modify a bit the *information_gain* method by replacing the *entropy* function by a variable *function* in order to choose which cost function to use in the decision tree.

3 Add reduced-error pruning

We recursively traverse the nodes of the tree, for each node we replace the node with a leaf. We calculate the precision on the plume dataset, if the precision is better than the accuracy compute on the training dataset we keep the leaf, otherwise we cancel the modifications while keeping the original node.

```python
def prune(self, node, data, y, data_prune, y_prune, accuracy):
    """Prune function to cut the useless branches of the tree"""
    # If it's not a leaf :
```

```python
    if node.value is None:
        # We save the two subtrees
        node_save_left = node.left
        node_save_right = node.right
        # If the subtree are actually two leafs with the same value :
        if node_save_left.value is not None and node_save_right.value is not None:
            if node_save_left.value == node_save_right.value:
                # We replace the node with a leaf
                return TreeNode(value=node.left.value)
        data1 = data[data[node.feature] < node.threshold]
        data2 = data[data[node.feature] >= node.threshold]

        y1 = data1.iloc[:, -1]
        y2 = data2.iloc[:, -1]

        value1 = y1.mode().iloc[0]
        value2 = y2.mode().iloc[0]
        # We build to new leaf for the actual node :
        node.left = TreeNode(value=value1)
        node.right = TreeNode(value=value2)

        # Compute the accuracy with this configuration
        result = self.prediction(data_prune)

        # if the new accuracy is not improved :
        if accuracy_score(y_prune, result) < accuracy:
            # We continue to explore the tree with the original node
            node.left = self.prune(node_save_left, data1, y1, data_prune,
                                   y_prune, accuracy)
            node.right = self.prune(node_save_right, data2, y2, data_prune,
                                    y_prune, accuracy)
        # otherwise we return the new node
        return node
    # if it's a leaf :
    return node
```

4 Evaluate your algorithm

We start by importing the data from the file *wine_ dataset.csv*. Then, we need to split the dataset in a training and test set. We proceed as below :

```python
def load_data():
    # Load the dataset :
    data_types = {
        "citric acid": float,
        "residual sugar": float,
        "sugar": float,
        "sulphates": float,
        "alcohol": float,
        "type": int
    }
    columns = ["citric acid", "residual sugar", "pH", "sulphates", "alcohol", "type'
    data = pd.read_csv("wine_dataset.csv", header=0, dtype=data_types, names=column

    return data

def trainTest(data):
    """Devide the data in train and test datasets"""
    y = data.iloc[:, -1]
    data_train, data_test, y_train, y_test = model_selection.train_test_split(
        data, y, test_size=0.30, random_state=42)
    return data_train, data_test, y_train, y_test
```

Then we can train our tree using the previous explained functions and print the results. Results with the decision tree implemented from scratch :

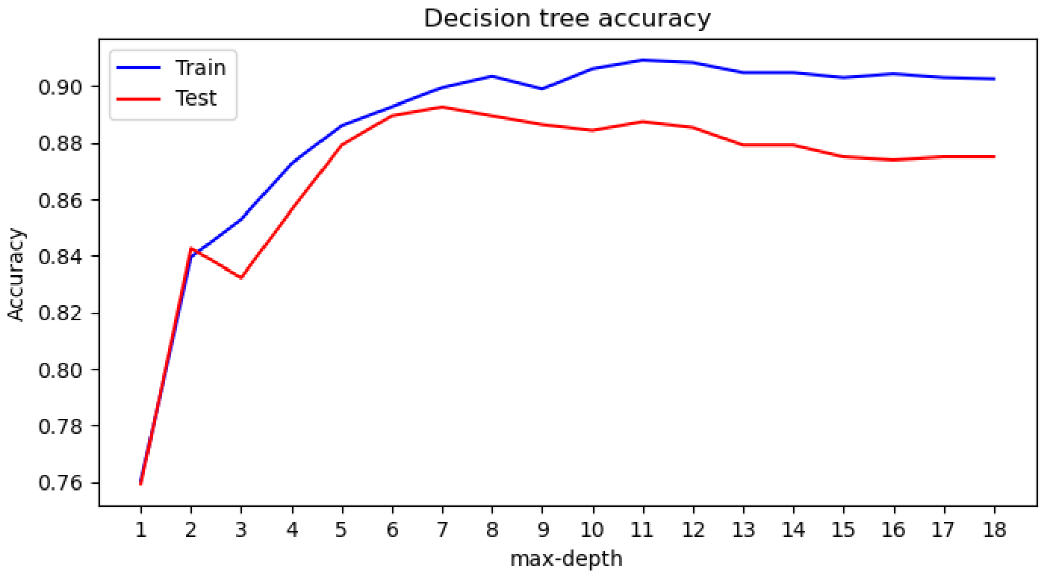

Figure 1: Implemented decision tree accuracy, impurity_measure='entropy'

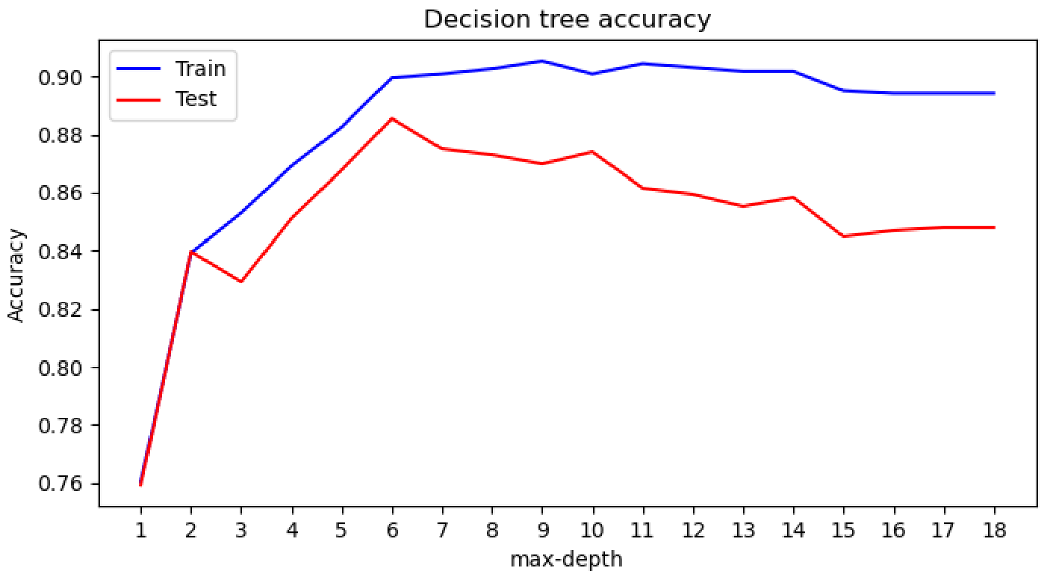

Figure 2: Implemented decision tree accuracy, impurity_measure='gini'

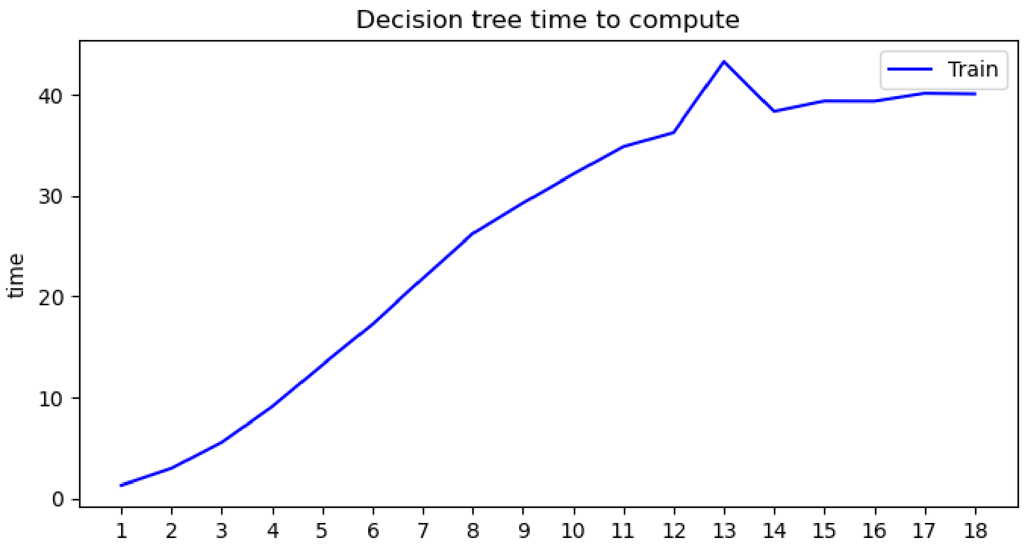

Figure 3: Time to compute in second for the implemented decision tree, with entropy (

5 Compare to an existing implementation

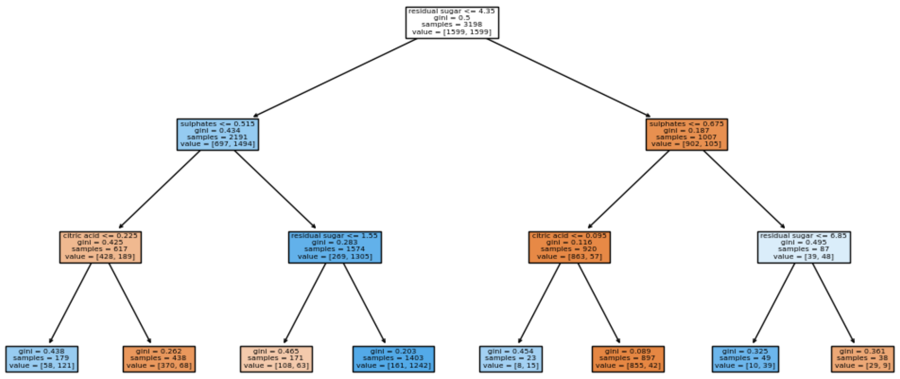

Figure 4: Decision tree from *sklearn* with maxdepth=3

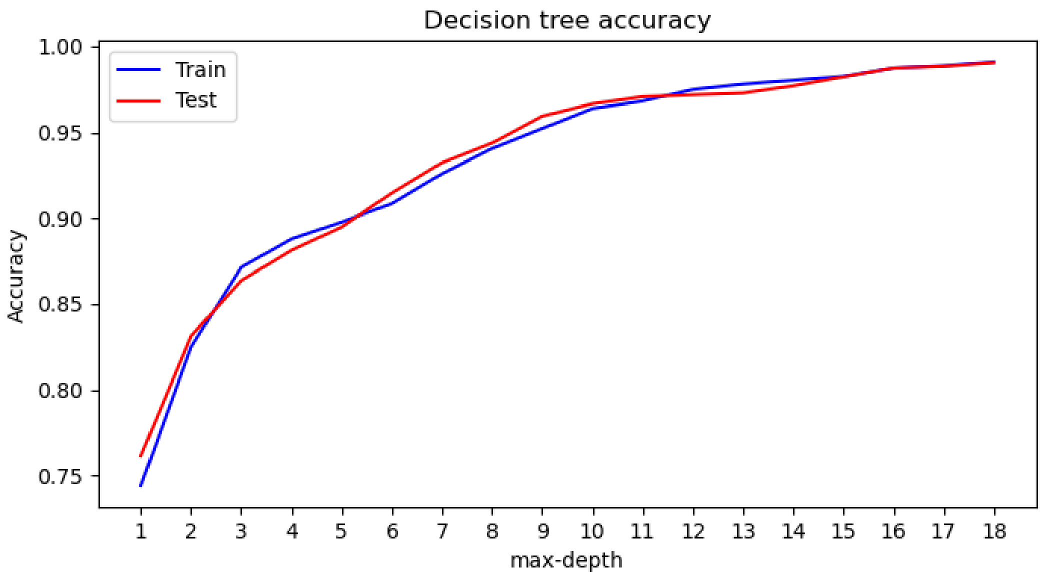

Figure 5: Sklearn decision tree accuracy, impurity_measure='entropy'

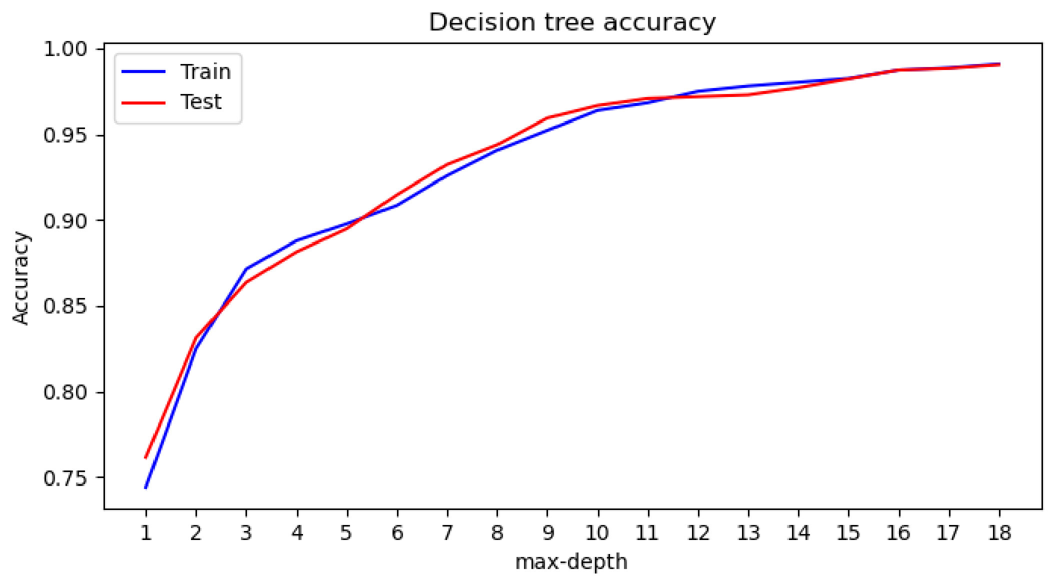

Figure 6: Sklearn decision tree accuracy, impurity_measure='gini'

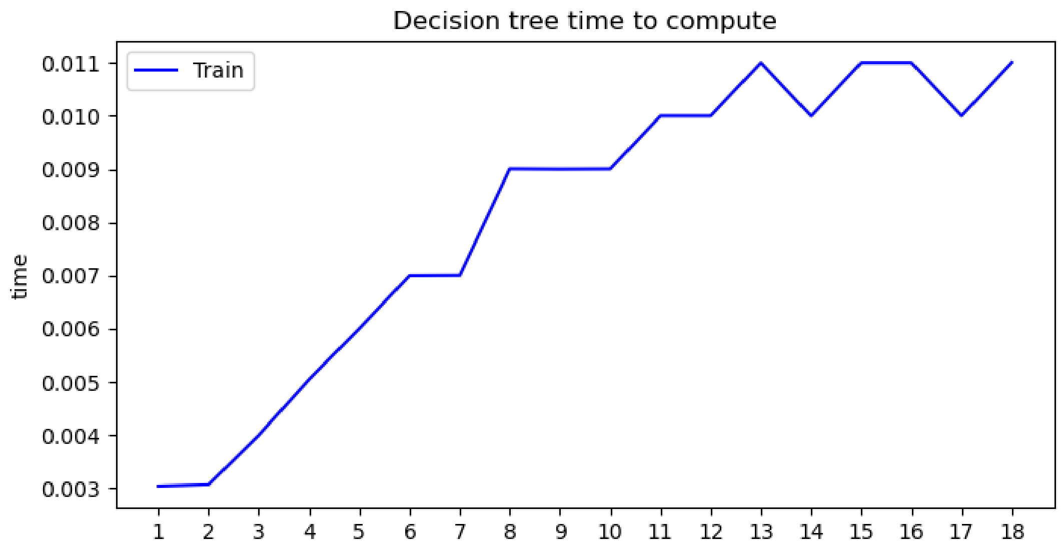

Figure 7: Time to compute in second for the sklearn decision tree, with entropy

As we can see my program is not efficient in comparison to the sklearn one. This difference can be attributed to scikit-learn's extensive codebase, which has undergone years of refinement and optimization by a community of developers.