Conception report

Machine Learning
Mushroom dataset

Performance comparison of a manually implemented decision tree with Gini impurity and entropy

---

Author:
Vincent Lalanne

April 5, 2024

1 Dataset

As an example, The Mushroom Data Set (`https://archive.ics.uci.edu/dataset/73/mushroom`) will be used to train and evaluate the classifier. It has 8124 instances and 22 features, related to the mushroom's characteristics, such as the cape's appearance, it's odour, population or habitat. The target is the edible nature of the mushroom, with two possible values: edible (`e`), or poisonous (`p`). Two instances of the dataset are shown in Fig. 1. The dataset includes 2480 rows with missing values. Those rows will be removed beforehand.

```
cap-shape                 f
cap-surface               y
cap-color                 y
bruises                   f
odor                      f
gill-attachment           f
gill-spacing              c
gill-size                 b
gill-color                h
stalk-shape               e
stalk-root                b
stalk-surface-above-ring  k
stalk-surface-below-ring  k
stalk-color-above-ring    p
stalk-color-below-ring    n
veil-type                 p
veil-color                w
ring-number               o
ring-type                 l
spore-print-color         h
population                v
habitat                   g
Name: 5207, dtype: object
poisonous    p
Name: 5207, dtype: object
```

```
cap-shape                 k
cap-surface               s
cap-color                 e
bruises                   t
odor                      n
gill-attachment           f
gill-spacing              c
gill-size                 b
gill-color                w
stalk-shape               e
stalk-root              NaN
stalk-surface-above-ring  s
stalk-surface-below-ring  s
stalk-color-above-ring    w
stalk-color-below-ring    w
veil-type                 p
veil-color                w
ring-number               t
ring-type                 e
spore-print-color         w
population                c
habitat                   w
Name: 5720, dtype: object
poisonous    e
Name: 5720, dtype: object
```

Figure 1: Dataset instances

2 Tree splitting

Let `X` be the features vector, created as a Pandas DataFrame. It's shape is (5644, 22) after removal of the rows with missing values. `y` is the target vector, of shape 5644, 1. First, let us create a function to compute the Gini impurity (eq. 1) and the entropy (eq. 2) of a given set of labels. They are defined as such, with `k` the number of classes and p_i the probability of samples to belong to a class at a given node.

$$\text{Gini(labels)} = 1 - \sum_{i=1}^{k} p_i^2 \tag{1}$$

$$\text{Entropy} = -\sum_{i}^{k} p_i \log_2(p_i) \tag{2}$$

A possible Python implementation is:

```python
def gini_impurity(labels):
    impurity = 1.0
    label_counts = Counter(labels)
    num_labels = len(labels)
    for label in label_counts:
        prob_label = label_counts[label] / num_labels
        impurity -= prob_label ** 2
    return impurity

def entropy(labels):
    entropy = 0.0
    label_counts = Counter(labels)
    num_labels = len(labels)
    for label in label_counts:
        prob_label = label_counts[label] / num_labels
        entropy -= prob_label * math.log2(prob_label)
    return entropy
```

Here, I import `Counter` from `collection` to calculate the probability of each label and `math` to calculate the log_2. To find the best feature and threshold, an option is to iterate over all features using a `for` loop for `feature_index` in `range(num_features)`. Each feature value is then extracted, and for each value, the gain is computed. The maximum gain is saved as well as its associated feature and threshold. The splitting criterion is either the Gini gain (eq. 3) or the information gain (eq. 4), which are defined as such:

$$\text{Gini Gain} = \text{parent Gini impurity} - \frac{n_{yes}}{n} \cdot \text{Gini impurity}\,(y_{yes}) - \frac{n_{no}}{n} \cdot \text{Gini impurity}\,(y_{no}) \tag{3}$$

$$\text{Information Gain} = \text{parent entropy} - \frac{n_{yes}}{n} \cdot \text{entropy}\,(y_{yes}) - \frac{n_{no}}{n} \cdot \text{entropy}\,(y_{no}) \tag{4}$$

With:

- n_{yes} the number of examples satisfying the rule (number of elements in the left instance)
- n_{no} the number of elements in the right instance
- y_{yes} is the left instance
- y_{no} is the right instance

A possible Python implementation is:

```python
def gain(data, labels, feature_index, threshold, gain_type):

    # Split the dataset based on the given feature and threshold
    left_indices = np.where(data[:, feature_index] <= threshold)[0]
```

```
    right_indices = np.where(data[:, feature_index] > threshold)[0]

    num_left = len(left_indices)
    num_right = len(right_indices)
    total_instances = num_left + num_right

    if gain_type == "gini impurity":
        # Calculate Gini impurity of the parent node
        parent_impurity = gini_impurity(labels)
        # Calculate the weighted average of child node impurities
        left_impurity = gini_impurity(labels[left_indices])
        right_impurity = gini_impurity(labels[right_indices])

        child_impurity = (num_left / total_instances) * left_impurity
        ↪  + (num_right / total_instances) * right_impurity

        # Calculate Gini gain
        gini_gain = parent_impurity - child_impurity

        return gini_gain

    elif gain_type == "entropy":
        # Calculate entropy of the parent node
        parent_entropy = entropy(labels)
        # Calculate the weighted average of child node entropies
        left_entropy = entropy(labels[left_indices])
        right_entropy = entropy(labels[right_indices])

        child_entropy = (num_left / total_instances) * left_entropy +
        ↪  (num_right / total_instances) * right_entropy

        # Calculate information gain
        information_gain = parent_entropy - child_entropy

        return information_gain


def find_best_split(data, labels):
    num_features = data.shape[1]
    best_gain = 0.0
    best_feature_index = -1
    best_threshold = None

    for feature_index in range(num_features):
        feature_values = data[:, feature_index]
        unique_values = np.unique(feature_values)
        for threshold in unique_values:
```

```
50              gain = gini_gain(data, labels, feature_index, threshold) #
                ↪  or information gain
51              if gain > best_gain:
52                  best_gain = gain
53                  best_feature_index = feature_index
54                  best_threshold = threshold
55
56      return best_feature_index, best_threshold
```

3 Label prediction

At this point, the tree is able to find the features and thresholds that maximise the gain for each node. However, it is not yet able to tell which class would be predicted at a leaf. A prediction is needed for each leaf so that the tree can actually be evaluated. For instance, if the tree depth is one, the splitting feature is `spore-print-color` and the threshold is `h`, then the tree would separate the labels without knowing which part is edible and which part is poisonous. To fix this, the most intuitive approach is to take the majority class at a leaf as a prediction. If, among the left part, the edible class is predominant, then the prediction would be `e`. A possible Python implementation is:

```
1  def build_tree(data, labels, depth, stopping_depth, gain_type):
2      # Stopping criteria
3      if depth >= stopping_depth:
4          # Reach the maximum depth, create a leaf node
5          leaf_labels = labels
6          majority_class = Counter(leaf_labels).most_common(1)[0][0]
7          return {'is_leaf': True, 'majority_class': majority_class}
8
9      # Find the best split
10     best_feature_index, best_threshold = find_best_split(data, labels,
       ↪  gain_type)
11
12     # No split found (leaf node)
13     if best_feature_index == -1:
14         # Create a leaf node with the majority class
15         majority_class = Counter(labels).most_common(1)[0][0]
16         return {'is_leaf': True, 'majority_class': majority_class}
17
18     # Split the dataset
19     left_indices = np.where(data[:, best_feature_index] <=
       ↪  best_threshold)[0]
20     right_indices = np.where(data[:, best_feature_index] >
       ↪  best_threshold)[0]
21
22     # Recursively build left and right subtrees
23     left_child = build_tree(data[left_indices], labels[left_indices],
       ↪  depth + 1, stopping_depth, gain_type)
```

```
    right_child = build_tree(data[right_indices],
    ↪  labels[right_indices], depth + 1, stopping_depth, gain_type)

    # Create a node representing the best split
    feature_name = mushroom_data.columns[best_feature_index]
    node = {
        'feature_name': feature_name,
        'threshold': best_threshold,
        'left_child': left_child,
        'right_child': right_child
    }

    return node

def predict(node, instance):
    if 'is_leaf' in node and node['is_leaf']:
        # If the node is a leaf, return the majority class
        return node['majority_class']

    feature_name = node['feature_name']
    threshold = node['threshold']
    feature_index = mushroom_data.columns.get_loc(feature_name)

    # Check if the instance value for the current feature is less than
    ↪  or equal to the threshold
    if instance[feature_index] <= threshold:
        # Recursively follow the left child
        return predict(node['left_child'], instance)
    else:
        # Recursively follow the right child
        return predict(node['right_child'], instance)
```

4 Tree evaluation and performance comparison

To evaluate the tree, I use a classic accuracy measure on a test set. It requires splitting the dataset in two sets of random samples. I also compute the accuracy on the train set to detect any overfitting. The accuracy is computed by comparing the prediction and the actual class of each instance in a leaf. In the evaluation process, I use a 80%/20% train/test split and respectively name the sets `X_train`, `X_test`, `y_train`, `y_test`. The simple evaluation process can be coded as following:

```
def evaluate_tree(tree_root, X_test, y_test):
    correct_predictions = 0
    total_instances = len(X_test)

    for i in range(total_instances):
        instance = X_test.iloc[i]
```

```
        true_label = y_test.iloc[i, 0]
        predicted_label = predict(tree_root, instance)

        if predicted_label == true_label:
            correct_predictions += 1

    accuracy = correct_predictions / total_instances
    return accuracy
```

The graphs of accuracies on train and test sets for both Gini impurity and entropy based splits are given in Fig. 2. The results are very similar, although a few differences can be noticed. First, it can be noted that the first and second splits appear to be the same, and the maximum accuracy (1.0) on the train set is reached at the fifth split in both cases. However, the Gini impurity-based split appears to have slightly lower capacities of generalisation, as the average gap between train and test accuracies is higher than with the entropy-based split. Also, it can be noted that the running time is slightly shorter when the impurity criterion is used. The two trees giving a perfect accuracy are given in Fig. 3 and 4. As previously noticed, the third split is different, causing accuracies to slightly differ in favour of the entropy-based tree. However, the general assessment remains that no method is dramatically better than the other. The best practice would be to test both methods on the specific dataset one works on, and choose the most suited to the needs.

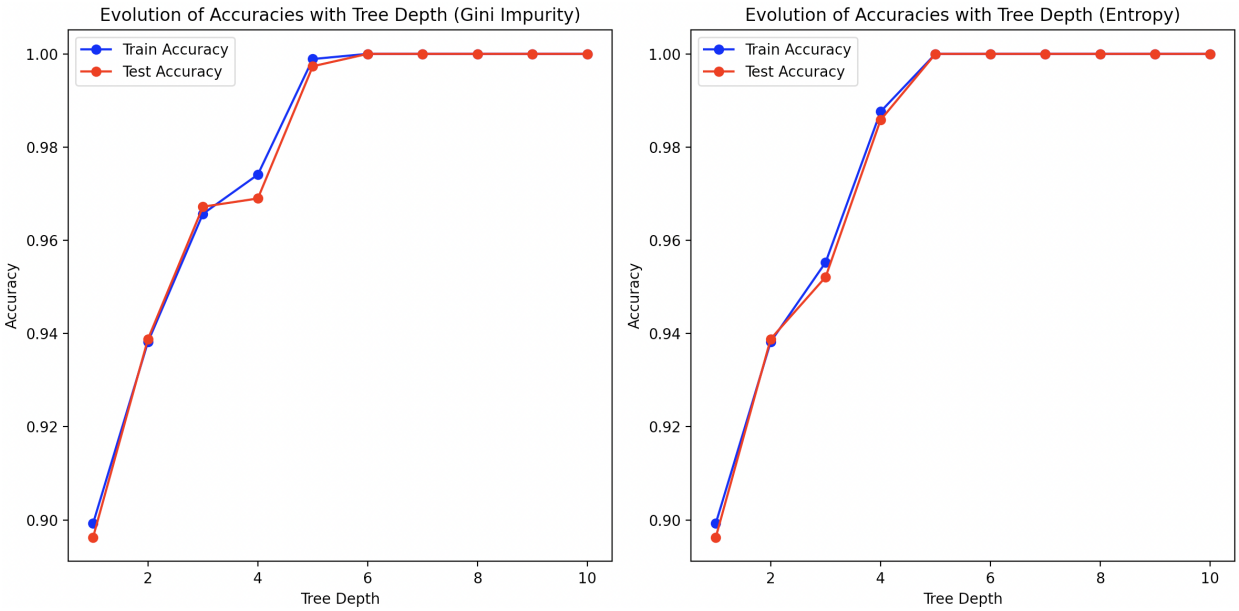

Figure 2: Accuracies

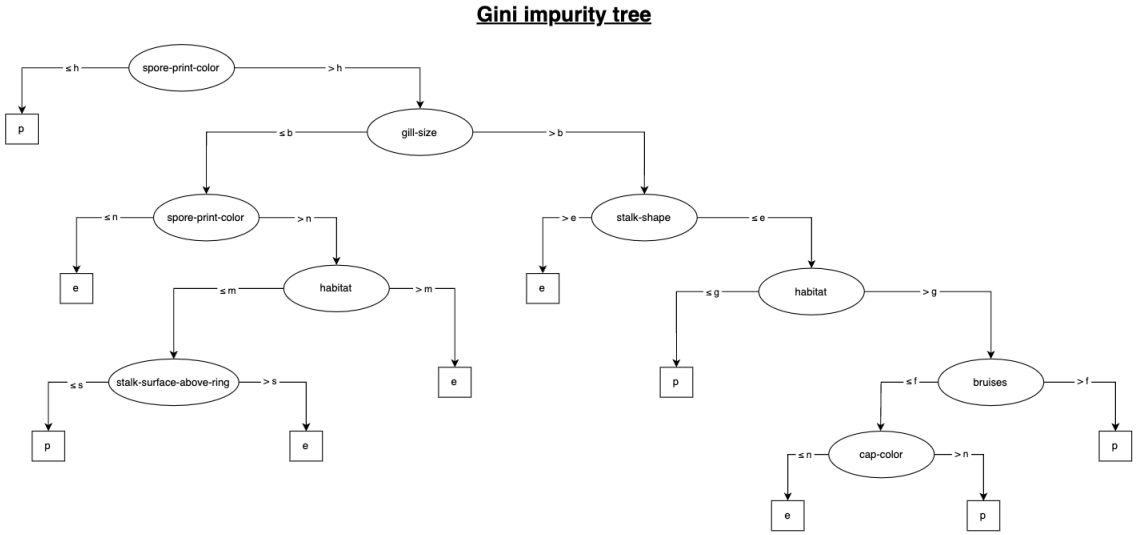

Figure 3: Tree generated with Gini impurity

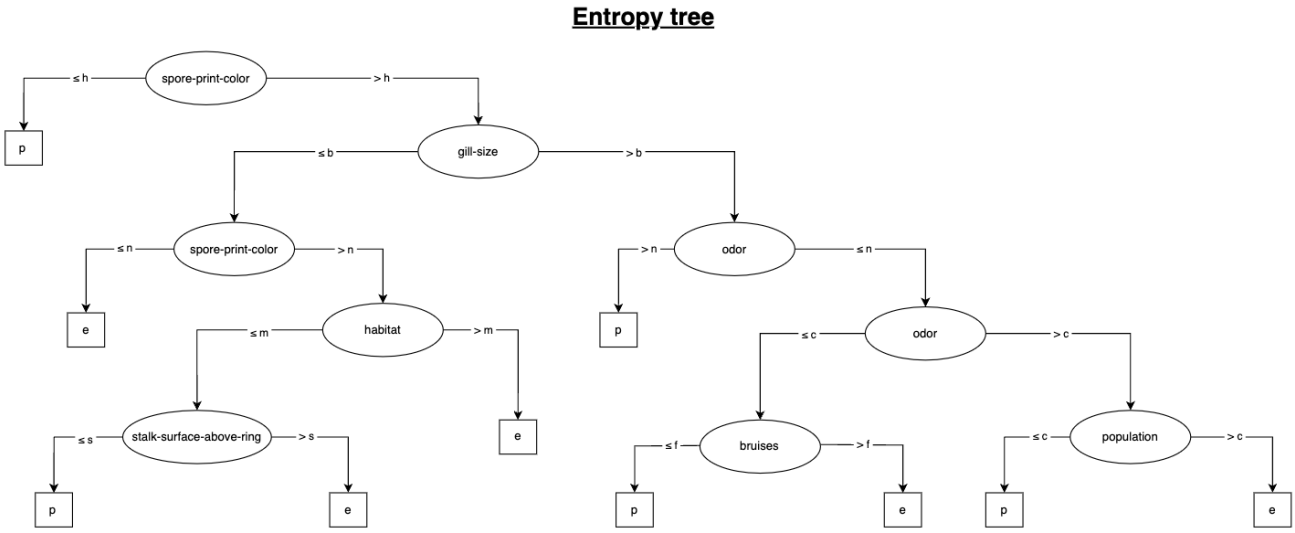

Figure 4: Tree generated with entropy