Decision Tree Binary Classification

Hossein Akrami

Declaration

Introduction

In this report, I will explain the fundamentals of the Decision Tree algorithm, which is a basic machine learning algorithm. The focus of this report is on a specific dataset, which can be accessed through the following link:

`https://archive.ics.uci.edu/dataset/848/secondary+mushroom+dataset`

This report is designed to be accessible without requiring advanced mathematical knowledge—basic high school-level mathematics will suffice.

The report is divided into two chapters. In the first chapter, I provide an intuitive explanation of the Decision Tree model and the motivation behind the project. The second chapter details the implementation of the model using Python. The code is built from scratch without relying on external libraries, so only fundamental coding skills in Python are necessary.

What is a Decision Tree Model?

A decision tree is a type of **supervised machine learning** used to categorize or make predictions based on how a previous set of questions were answered.

What does it mean?

To answer this question, I am focusing on one specific example, which is the mushroom dataset. In this focus, we assume that the reader does not have any previous experience with the Decision Tree algorithm and hence, I developed it step by step.

Imagine that we have a mushroom dataset which contains different records about various mushrooms. The dataset appears as a two-dimensional table. Each row contains data about each mushroom, and each column contains the **specifications (features)** of each mushroom.

Mushroom ID	Size	Location	Color	Class
1	Small	Switzerland	Red	Poisonous
2	Medium	USA	Brown	Edible
3	Large	USA	Yellow	Edible
4	Small	USA	Red	Poisonous
5	Medium	Switzerland	Yellow	Poisonous

Table 1: Sample Mushroom Dataset

What are we going to do?

We are going to create a **model** that is able to predict whether a mushroom is edible or poisonous.

But how? Because there is no function or simple criterion to tell us if a mushroom is edible or not. Additionally, we are not nutritionists and do not have sufficient knowledge about this science. So, the idea is to create a **machine** model which has the ability to **learn** from the data what the correlation is between the independent features and the target variable, which is called in simple terms a **machine learning model**.

For instance, imagine that we have 100 mushrooms, with 50 in Switzerland and 50 in the United States. If all of the 50 mushrooms in Switzerland are poisonous while all the 50 in the US are edible, it is clear that there is a high correlation between the location and the target variable. In this case, the desired machine learning model should learn this. On the other hand, if all the Swiss mushrooms have 25 poisonous and 25 edible, and the same for the US, it is clear that there is no relation between the location and the target variable. So, in this case, the machine should be able to learn it. In other words, the machine needs to learn from the data.

What are the other features that a machine needs to learn?

Imagine that we have a feature that is not categorical but instead numerical, for example, the size of the mushroom. For one mushroom, it is 1 inch, while for another, it is 3.4 inches. The model should be able to learn what the threshold for classifying the data is. For example, imagine that we have 100 mushrooms and each one greater than 3 inches is poisonous (the reason could be that as mushrooms grow, they become poisonous). So, in this case, the model needs to learn not only the size feature but also the threshold.

The approach in this project is a **Decision Tree** model, which is a very famous ML model. If you read the rest of this chapter, you will understand the Decision Tree fundamentals and also how I implemented it in this project.

What is a Decision Tree and How Does It Work in This Project?

Imagine a tree diagram. There is one **node**, which is then divided into two or many nodes. Each child node is then divided into multiple nodes, and this continues for the next generations and repeats. At the first node, we have all the mushrooms. In the next generation, we split the data into two **nodes** (as it is a binary classification; however, it can be multiple). Then we are interested in splitting each child node into two **grandchild nodes**. We do the splits based on the most important feature at each level. So, if the split at the first generation is location and the split at the second generation is size, it indicates that the location has more **correlation** to the target variable than the size.

In summary, the model needs to learn three things: firstly, the **splitting feature**, the **threshold**, and the **left** and **right nodes**.

Additionally, the model needs to learn if a node is a leaf node (final node). For example, if all of the mushrooms in one node are poisonous, then we need to stop the decision tree and label this node (P).

Decision Tree Diagram

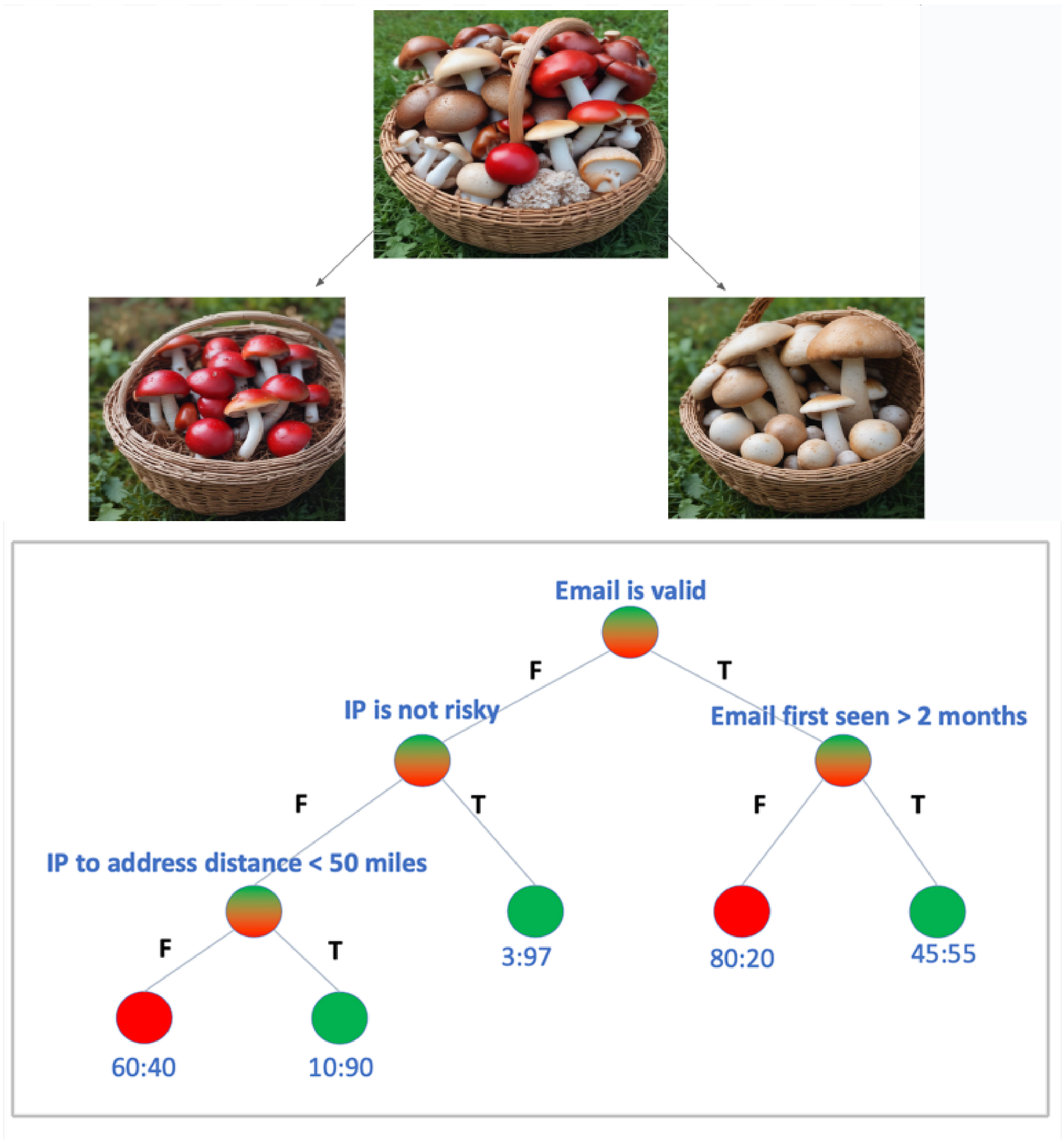

Figure 1: Decision Tree Diagram, First one for mushroom and second for the email classificatin

Criteria for Splitting: Gini Impurity, Entropy, and Information Gain

In the previous sections, we discussed the essential features that our decision tree model needs to learn during the training process. These include identifying the splitting feature, determining the threshold, defining the left and right nodes, and recognizing leaf nodes along with their corresponding values.

Now, we delve into how the decision tree makes splits at each level. Specifically, we'll explore the criteria used to evaluate the quality of these splits: **Gini Impurity**, **Entropy**, and **Information Gain**. These metrics help the model determine the most effective feature to split the data, ensuring that the resulting child nodes are as pure as possible.

Why Do We Need Splitting Criteria?

Consider our mushroom dataset example. Suppose we have 100 mushrooms, with 50 originating from Switzerland (all poisonous) and 50 from the USA (all edible). Intuitively, the location feature is a perfect criterion for splitting since it perfectly separates poisonous from edible mushrooms. However, computers lack this innate intuition. They require a systematic method to evaluate and determine which features best split the data to maximize classification accuracy.

Splitting criteria like Gini Impurity, Entropy, and Information Gain provide the mathematical foundation for making these decisions, enabling the model to quantify the "purity" of a split and choose the most effective feature accordingly.

Gini Impurity

Gini Impurity measures the likelihood that a randomly chosen element from a set would be incorrectly classified if it were randomly labeled according to the distribution of labels in the set.

The formula for Gini Impurity is:

$$Gini = 1 - \sum_{i=1}^{n} p_i^2$$

Where p_i is the proportion of samples belonging to class i.

A lower Gini Impurity indicates a purer node, meaning that the node predominantly contains elements of a single class.

Entropy

Entropy is a measure of uncertainty or disorder within a set. It quantifies how mixed the classes are within a node. High entropy means the node has a high level of disorder, with multiple classes mixed together. Conversely, low entropy indicates that the node is more homogenous.

The formula for Entropy is:

$$Entropy = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Where p_i is the proportion of samples belonging to class i.

Entropy ranges from 0 (completely pure) to 1 (completely impure for binary classification).

Information Gain

Information Gain measures the reduction in entropy or impurity after a dataset is split on a particular feature. It essentially quantifies the improvement in purity by making a specific split.

The formula for Information Gain is:

$$Information\ Gain = Entropy(\text{parent node}) - \sum_{j=1}^{k} \left(\frac{n_j}{n} \times Entropy(\text{child node}_j) \right)$$

Where n_j is the number of samples in child node j, and n is the total number of samples in the parent node.

A higher Information Gain indicates a more effective feature for splitting the data, leading to purer child nodes.

Choosing the Best Split

When constructing the decision tree, the algorithm evaluates each feature using Gini Impurity, Entropy, and Information Gain to determine which feature provides the best split. The goal is to select the feature that results in the lowest impurity or entropy after the split, leading to the most homogeneous child nodes.

Hyperparameters in Decision Trees

In building a decision tree, there are several hyperparameters that play a critical role in controlling the complexity and performance of the model. These hyperparameters need to be carefully tuned to achieve the best balance between underfitting and overfitting. Below, we discuss some of the most important hyperparameters.

Maximum Depth

Maximum Depth is one of the key hyperparameters in a decision tree. It controls the maximum number of splits or levels that the tree can have from the root node to the farthest leaf node.

Intuition:

Imagine a scenario where we have only two features: location and whether the mushroom is poisonous. If all poisonous mushrooms are in Switzerland and all edible mushrooms are in the USA, the decision tree can easily split the data based on location. However, if we allow the tree to grow too deep, it might start to "learn" patterns that are actually just noise in the data, rather than general rules. For example, the tree might find that within Switzerland, certain smaller regions have slightly different ratios of poisonous to edible mushrooms and use this to further split the data. But this may not be a generalizable pattern and could lead to overfitting—where the model performs well on the training data but poorly on new, unseen data.

Setting a maximum depth helps to prevent this by limiting how much the tree can "learn" from the data. A shallow tree might underfit (missing important patterns), while a very deep tree might overfit (capturing noise rather than signal).

Minimum Number of Samples for Splitting

Minimum Number of Samples for Splitting is another crucial hyperparameter. It specifies the minimum number of samples required to make a split at a node. This parameter helps control the tree's growth by preventing it from making splits that are based on very few data points, which are more likely to be outliers or noise.

Other Hyperparameters

Similar to Maximum Depth and Minimum Number of Samples for Splitting, there are other important hyperparameters, including:

- **Minimum Number of Samples for a Leaf Node**: This controls the minimum number of samples that a leaf node must contain. It helps in preventing the model from creating leaf nodes with very few samples, which might not represent the overall data distribution well.
- **Maximum Number of Features**: This limits the number of features that the model can consider at each split. By reducing the number of features, you can decrease the likelihood of overfitting.
- **Maximum Leaf Nodes**: This parameter controls the maximum number of leaf nodes in the tree. Limiting the number of leaf nodes can help simplify the model and improve generalization.

These hyperparameters can be tuned using techniques like grid search or random search to find the best combination that yields the most effective and generalizable model.

Implementation of the Decision Tree in Python from Scratch

Introduction

This chapter focuses on the implementation of a decision tree classifier in Python from scratch. While basic knowledge of Python programming is required to follow along, the concepts and code are explained in a straightforward manner. We will translate the ideas discussed in the previous chapter into functional Python code, demonstrating how a decision tree can be constructed without the use of specialized libraries such as `scikit-learn`.

About the Dataset

The dataset used in this implementation contains 61,069 mushroom records, each with 21 different features. These features describe various characteristics of the mushrooms, such as size, color, and location, which will be used to predict whether a mushroom is poisonous or edible.

Data Encoding

Before we can use the data for training a decision tree, we need to encode the categorical features into a numerical format that can be processed by the machine learning algorithm. In Python, this can be done using the `pd.get_dummies()` function from the `pandas` library.

```
data_encoded = pd.get_dummies(data)
```

Explanation:

The `pd.get_dummies()` function converts categorical variables into dummy/indicator variables. This is necessary because machine learning algorithms work with numerical inputs. Encoding converts categories into a format suitable for processing by the decision tree.

Tree Implementation

In Python, there isn't a built-in concept of a decision tree unless you use professional libraries like `scikit-learn`. However, for educational purposes, we will build a decision tree from scratch by defining a class in Python that encapsulates the decision tree logic.

Defining the Decision Tree Class

Below is the initial code for our `DecisionTree` class. This class will include methods for calculating Gini impurity, entropy, and information gain, as well as other functions necessary for constructing and splitting the tree.

```
class DecisionTree:
    def __init__(obj, maximum_depth=None, max_leaf_nodes=None,
                 entropy_threshold=None, split_function=None,
                 minimum_samples_split=2, feature_names=None):
        obj.maximum_depth = maximum_depth
        obj.max_leaf_nodes = max_leaf_nodes
        obj.entropy_threshold = entropy_threshold
        obj.split_function = split_function
        obj.minimum_samples_split = minimum_samples_split
        obj.root = None
        obj.feature_names = feature_names
```

Explanation:

The `__init__` method initializes the decision tree with several parameters: - `max_depth`: The maximum depth of the tree. - `max_leaf_nodes`: The maximum number of leaf nodes. - `entropy_threshold`: The threshold for entropy to decide when to stop splitting. - `split_function`: The function used to determine the best split (e.g., Gini impurity or entropy). - `minimum_samples_split`: Minimum number of samples required to split an internal node. - `feature_names`: Names of the features, useful for interpretability.

Implementing Key Functions

The decision tree relies on several key functions to calculate criteria such as Gini impurity, entropy, and information gain. Below are the implementations of these functions.

```
def _gini_impurity(obj, y):
    hist = np.bincount(y)
    probs = hist / len(y)
    gini = 1.0 - np.sum(probs ** 2)  # gini = np.sum(probs * (1 - probs
       ))
    return gini
```

Explanation:

The `_gini_impurity` function calculates the Gini impurity, which measures the level of disorder or impurity in a set of labels. It helps in determining how well a split separates the classes.

```
def _scaled_entropy(obj, y):
    hist = np.bincount(y)
    probs = hist / len(y)
    scaled_ent = -np.sum([(p / 2) * np.log2(p) for p in probs if p >
       0])
    return scaled_ent
```

Explanation:

The _scaled_entropy function computes the scaled entropy, which measures the amount of uncertainty or randomness in the data. It is used to evaluate the effectiveness of a split.

```
def _information_gain(obj, y, X_column, split_thresh):
    parent_criterion = obj.criterion_func(y)

    left_idxs, right_idxs = obj._split(X_column, split_thresh)
    if len(left_idxs) == 0 or len(right_idxs) == 0:
        return 0

    y_left, y_right = y[left_idxs], y[right_idxs]
    child_criterion = obj._weighted_criterion(y_left, y_right, obj.
       criterion_func)
    gain = parent_criterion - child_criterion
    return gain
```

Explanation:

The _information_gain function calculates the information gain from a split, which measures how much information is gained by partitioning the data at a given threshold. This helps in deciding the best feature and threshold for splitting.

```
def _best_feature_to_split_based_on(obj, X, y, feature_indexes):
    best_information_gain = -1
    split_idx, split_thresh = None, None # split_idx: index of the best
        feature
    for index_counter in feature_indexes: # search for all indexes in
       the list of indexes
        X_column = X[:, index_counter]
        thresholds = np.unique(X_column) # split_thresh the optimal
           threshold for the best feature
        for threshold in thresholds:
            gain = obj._information_gain(y, X_column, threshold)
            if gain > best_information_gain:
                best_information_gain = gain
                split_idx = index_counter
                split_thresh = threshold
    return split_idx, split_thresh
```

Explanation:

The _best_feature_to_split_based_on function identifies the best feature and threshold to split on by evaluating the information gain for each possible split. This helps in choosing the most effective feature for partitioning the data.

Hyperparameter Tuning

Introduction

In this chapter, we explore hyperparameter tuning, a crucial step in optimizing machine learning models. Hyperparameter tuning involves systematically searching for the best combination of hyperparameters to improve the model's performance. This process ensures that the model generalizes well to unseen data and achieves the best possible performance.

Definition of Hyperparameter Grid

The grid search function allows you to define a parameter grid (`param_grid`) that specifies different hyperparameters and their respective values to be tested. This grid is used to evaluate various combinations of hyperparameters.

```
param_grid = [
    {
        'maximum_depth': [40],
        'split_function': ['scaled_entropy', 'gini', 'squared']
    },
    {
        'max_leaf_nodes': [150],
        'split_function': ['scaled_entropy', 'gini', 'squared']
    },
    {
        'entropy_threshold': [0.0001],
        'split_function': ['scaled_entropy', 'gini', 'squared']
    }
]
```

Explanation:

The `param_grid` defines different sets of hyperparameters and their values that will be tested. For example, it includes different values for `max_depth`, `max_leaf_nodes`, and `entropy_threshold` along with various split functions. This allows for a comprehensive search over different combinations.

Generating Parameter Combinations

The `grid_search` function generates all possible combinations of hyperparameters from the `param_grid`.

```
param_combinations = [
    dict(zip(param_dict.keys(), values)) for param_dict in param_grid
       for values in
    itertools.product(*param_dict.values())
]
```

Explanation:

The `param_combinations` list contains all possible combinations of hyperparameters by iterating through the `param_grid`. Each combination is represented as a dictionary of hyperparameters.

Evaluation of Parameter Combinations

For each combination of hyperparameters, the `evaluate_params` function is called to train and evaluate the model.

```
def evaluate_params(params):
    current_scores = []
    depths = []
    leafs = []

    # 80% train, 20% validation
    for train_idxs, val_idxs in cv.split(X_train, y_train):
        X_train_cv, y_train_cv = X_train.iloc[train_idxs], y_train.iloc
           [train_idxs]
        X_val_cv, y_val_cv = X_train.iloc[val_idxs], y_train.iloc[
           val_idxs]

        model = DecisionTree(
            maximum_depth=params.get('maximum_depth'),
            max_leaf_nodes=params.get('max_leaf_nodes'),
            entropy_threshold=params.get('entropy_threshold'),
            split_function=params['split_function'],
            min_samples_split=2,
            feature_names=X_train.columns
        )

        model.fit(X_train_cv.values, y_train_cv.values)
        y_val_pred = model.predict(X_val_cv.values)
        score = scoring_func(y_val_cv, y_val_pred)
        #print(f"zero one loss: {score:.6f} with params: {params}")
        current_scores.append(score)
        depths.append(model.depth)
        leafs.append(model.leaf_count)

    mean_score = np.mean(current_scores)
    mean_depth = np.mean(depths)
    mean_leafs = np.mean(leafs)

    print(f"zero one loss: {mean_score:.5f} with params: {params} \t
       mean depth: {mean_depth:.1f} and mean leafs: {mean_leafs:.1f}")

    return params, mean_score
```

Explanation:

The `evaluate_params` function trains and evaluates the model for a given set of hyperparameters. It performs cross-validation to assess the model's performance and calculates the mean score, depth, and number of leaf nodes.

Parallel Execution of Grid Search

The grid search process can be parallelized to speed up the evaluation of parameter combinations.

```
results = Parallel(n_jobs=-1)(delayed(evaluate_params)(params) for
    params in param_combinations)
```

Explanation:

Using `joblib.Parallel`, the grid search process is executed in parallel, which speeds up the evaluation by utilizing multiple CPU cores. Each set of parameters is evaluated concurrently.

Finding the Best Hyperparameters

After evaluating all parameter combinations, the results are sorted, and the best hyperparameters are selected based on the lowest zero-one loss.

```
sorted_results = sorted(results, key=lambda x: x[1])[:10]

print("\nTop 10 Results:")
for rank, (params, mean_score) in enumerate(sorted_results, 1):
    print(f"Rank {rank}: Mean zero one loss: {mean_score:.6f} with
        params: {params}")

return results, sorted_results[0][0], sorted_results[0][1]
```

Explanation:

The results of the grid search are sorted based on the zero-one loss metric. The top 10 parameter sets are displayed, and the best-performing hyperparameters are identified for final model training.

Using Best Hyperparameters

The best hyperparameters are used to create and train the final decision tree model.

```
best_tree = DecisionTree(
    maximum_depth=best_params.get('maximum_depth'),
    max_leaf_nodes=best_params.get('max_leaf_nodes'),
    entropy_threshold=best_params.get('entropy_threshold'),
    split_function=best_params['split_function'],
    min_samples_split=2,  # This is fixed as per the original
        configuration
    feature_names=X_train.columns
)
```

Explanation:

The `best_tree` is created using the best hyperparameters found during the grid search. This final model is then trained with the optimal settings and used for predictions.

Summary

Hyperparameter tuning is a vital process in optimizing machine learning models. The grid search method systematically explores different hyperparameter settings to find the best combination. The process involves:

- Defining a grid of hyperparameters to search.
- Generating all possible combinations of these hyperparameters.
- Training and evaluating models with these combinations using cross-validation.
- Identifying the best combination based on the performance metric (zero-one loss).

The final model is trained with the best-found hyperparameters and evaluated on test data to ensure optimal performance.

Conclusion

In this research project, I introduced the fundamentals of the decision tree algorithm and then implemented the concept using Python code on a specific mushroom dataset. One crucial aspect to consider is that we assumed all features were entirely **independent**, meaning there is no relation between any of them.

The list of all the features is as follows:

- **class**
- **cap-diameter**
- **cap-shape**
- **cap-surface**
- **cap-color**
- **does-bruise-or-bleed**
- **gill-attachment**
- **gill-spacing**
- **gill-color**
- **stem-height**
- **stem-width**
- **stem-root**
- **stem-surface**
- **stem-color**
- **veil-type**
- **veil-color**
- **has-ring**
- **ring-type**
- **spore-print-color**

- **habitat**
- **season**

In machine learning models, it is crucial to ensure that all features are **independent**. Otherwise, we may encounter a problem known as **Bias Variable**. For example, if the color of the mushroom is influenced by the size of the stem, and we attempt to determine the correlation between the class and these two variables, our calculation would be prone to **error**.

In addition, it should be noted that we are estimating the correlation based on the statistical data. In other words, **correlation does not necessarily mean causation**!