# INF264
# Project 1:
# Implementing decision trees

Alina Artemiuk

Fall 2023

## 1. Decision Tree Model Implementation

### 1.1 Overview

The decision tree is a supervised machine learning algorithm that is used for both classification and regression tasks. It works by recursively splitting the data based on feature values to create a tree structure. The end nodes (leaves) of this tree are the predictions. The process starts at the root of the tree and moves to the leaves by following decision rules derived from the data.

### 1.2. Model Architecture

#### 1.2.1 `DecisionNode` Class

The `DecisionNode` class represents a node in the decision tree. Each node contains:

- `feature_index`: Index of the feature this node splits on.
- `threshold`: The threshold value for this split.
- `value`: For leaf nodes, this contains the predicted class or value.
- `left_branch` and `right_branch`: Pointers to the left and right child nodes.

#### 1.2.2 `DecisionTree` Class

The `DecisionTree` class encapsulates the decision tree algorithm and its functionalities.

- The `_entropy` and `_gini` methods compute the impurity of a dataset, using entropy and Gini impurity measures respectively.
- The `_information_gain` method calculates the information gain for a particular split. This metric measures how much the impurity decreases when we split the dataset using a certain feature and threshold.
- The `_best_split` method finds the best feature and threshold to split the dataset on, iterating over each feature and each unique value of that feature.
- The `_learn` method is a recursive function that constructs the decision tree. If a stopping criterion (like maximum depth) is reached, or if no further improvement can be made, it creates a leaf node with the most common label. Otherwise, it splits the dataset and calls itself on the two subsets.
- The `predict` method is used to predict the class of a single data point by traversing the tree from the root to a leaf.
- The `_accuracy` method calculates the prediction accuracy of the model on a given dataset.
- The `_prune` method applies reduced-error pruning. It simplifies the tree by replacing a subtree with a leaf node if doing so does not decrease the accuracy on a validation dataset.

- The `learn` method serves as the main entry point to construct the decision tree from data. If pruning is enabled, it first splits the dataset into training and validation sets and then prunes the tree after it's built.
- The `compute_feature_importance` method computes the importance of each feature based on how much information gain it provides when used in splits.
- The `print_tree` method displays the decision tree in a readable format, which helps in understanding the structure and rules of the constructed tree.

### 1.3. Building The Tree

1. The tree construction begins with the entire dataset at the root.
2. The best feature and threshold for splitting are found using the `_best_split` method.
3. If a stopping criterion (like maximum depth) is reached, or no further improvement can be made by splitting, a leaf node is created with the most common label of the data points in that node.
4. Otherwise, the data is split based on the chosen feature and threshold. This creates two new subsets of the data.
5. The process is then recursively repeated for each subset, resulting in a left branch and a right branch of the tree.
6. The recursion continues until all data points are perfectly classified or until reaching other stopping criteria.

### 1.4. Pruning

Once the tree is fully grown, it may overfit the training data. Pruning is a technique to simplify the tree, reduce its depth, and thereby increase its generalization to unseen data. The current project uses reduced-error pruning, which involves:

1. Splitting the dataset into a training set and a pruning set.
2. Building the tree using the training set.
3. Iterating through the nodes of the tree, starting from the leaves, and attempting to replace each node with a leaf node of the majority class of that subtree.
4. If replacing the node does not decrease accuracy on the pruning set, the node is pruned (replaced by the leaf node).

### 1.5. Prediction

For a new data point, prediction starts at the root node. The tree is traversed based on the features of the data point until a leaf node is reached. The value of this leaf node is the predicted class.

## 2. Data Exploration

The main task is to classify wines into one of two categories: white or red, based on five specific numerical measurements. The data provides five continuous features: *citric acid*, *residual sugar*, *pH*, *sulphates*, and *alcohol*. An additional column, *type*, serves as the categorical class label. In this column, a value of 0 indicates that the wine is a white wine, and 1 indicates that it is a red wine.

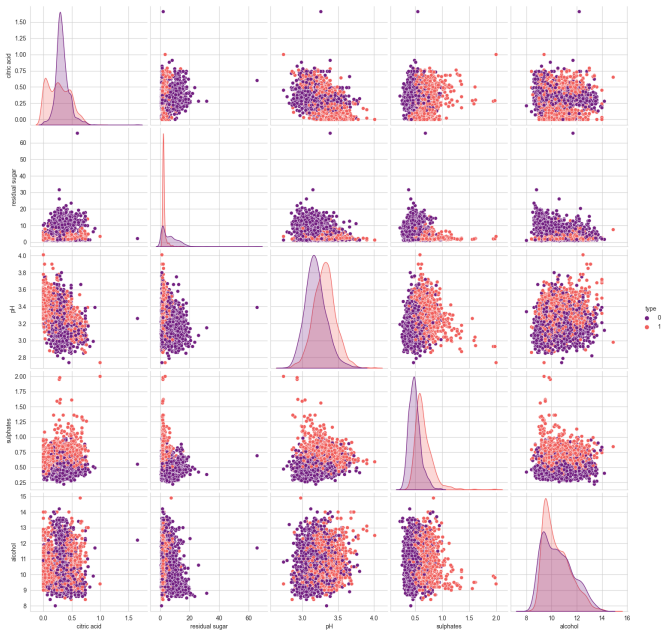


Figure 1: Feature relationships differentiated by wine type.

## 2.1 Data Overview

The dataset contains 3198 unique wine samples. Both classes — white and red wines — are equally represented, each with 1599 samples.

## 2.2 Descriptive Statistics

The following conclusions can be drawn from the statistical analysis:

- The feature *citric acid* has values ranging from 0 to 1.66, with an average of around 0.30.
- *residual sugar* exhibits a wide range, with the highest being 65.8 and the lowest at 0.6. The average value hovers around 4.45.
- The *pH* values range from 2.74 to 4.01, and the median value is 3.24.
- *sulphates* have an average value of approximately 0.57, while *alcohol* averages around 10.46.

## 2.3 Class Distribution

The dataset shows a balanced class distribution. Both white and red wines have 1599 samples each, which means a 50/50 split.

### 2.4 Feature Correlation

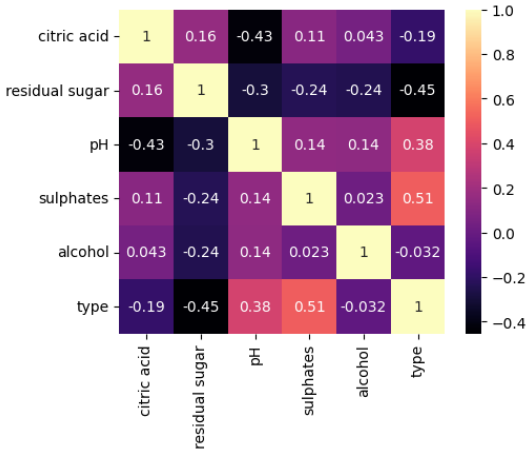


Figure 2: Wine Dataset Correlation Heatmap.

Analyzing the correlations between features provides the following insights:

- *citric acid* has a negative correlation with *type*, suggesting that wines with higher citric acid levels might be more commonly white wines.
- *residual sugar* showcases a strong negative correlation with *type*, indicating that wines with higher sugar levels are likely to be white wines.
- *pH* values are positively correlated with *type*, implying that wines with higher *pH* values tend to be red.
- *sulphates* also display a positive correlation with *type*, suggesting that higher sulphate levels are associated with red wines.
- The correlation between *alcohol* and *type* is weak, suggesting that the alcohol content is not the main factor distinguishing between white and red wines.

## 3. Hyperparameter Tuning

The goal of hyperparameter tuning is to identify the best combination of hyperparameters for a decision tree model. This includes both a custom `DecisionTree` model and sklearn's `DecisionTreeClassifier`. The performance metric used to assess model quality is accuracy.

### 3.1 Evaluated Hyperparameters

The following hyperparameters are considered:

- **Criterion**: The function to measure the quality of a split. Options are:
  - `Entropy`
  - `Gini`
- **Max Depth**: The maximum depth of the decision tree. This is defined from 1 to 15, with `None` representing no maximum depth (the tree can grow until all leaves are pure).
- **Pruning**: Whether or not to prune the tree to avoid overfitting.
  - `False`: No pruning
  - `True`: Apply pruning
- **Pruning Strategy**:

– For the custom `DecisionTree` model: The ratio of data to use for pruning, ranging from 0.1 to 0.5.
– For sklearn's `DecisionTreeClassifier`: The `ccp_alpha` parameter is used for Minimal cost complexity pruning. The best value for this parameter is determined dynamically for each fold using the `find_best_ccp_alpha` function.

### 3.2 Evaluation Method

The dataset is first split into a training-validation set and a test set using an 80/20 ratio. The `StratifiedKFold` method with 5 splits is then applied to ensure that each fold of the dataset maintains the same proportion of observations for each class label.

## 4. Exploring Hyperparameter Configurations

This section presents plots illustrating the effects of different hyperparameters on model performance. These parameters include the tree's depth, pruning, pruning ratio, and impurity measure (Gini vs. Entropy).

### 4.1 Custom `DecisionTree`

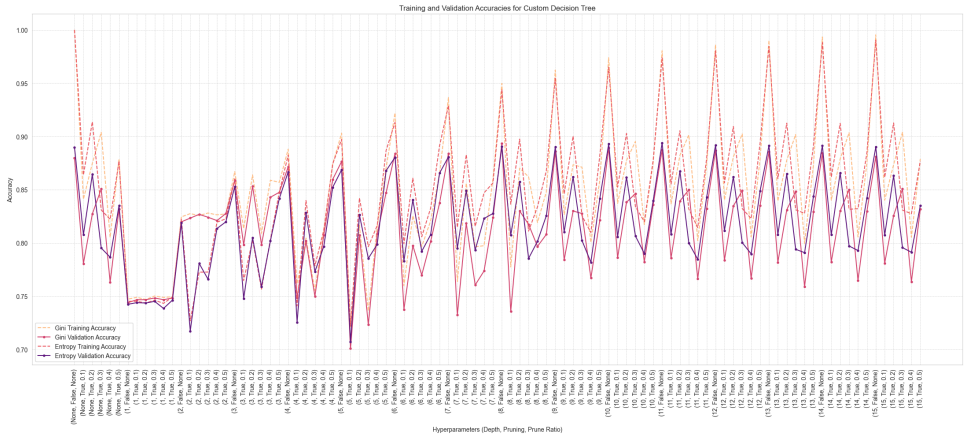


Figure 3: Training and validation accuracies for the custom `DecisionTree` model.

The custom `DecisionTree` model shows differences in accuracy when switching between the Gini and Entropy criteria. The performance of the decision tree is also affected by variations in pruning and the associated ratio.

### 4.2 Sklearn's `DecisionTreeClassifier`

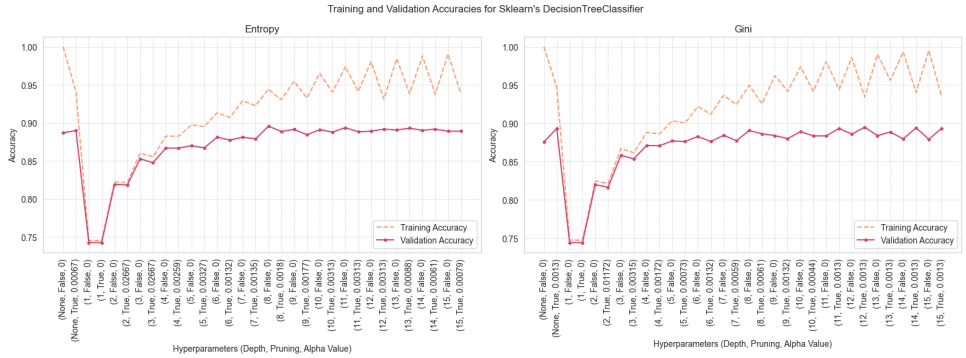


Figure 4: Training and validation accuracies for Sklearn's `DecisionTreeClassifier`.

Sklearn's `DecisionTreeClassifier` highlights the influence of the `ccp_alpha` parameter, which determines the extent of pruning. In certain hyperparameter setups, there is potential overfitting when the training accuracy is higher than the validation accuracy. Conversely, some configurations may indicate underfitting, where both training and validation accuracy are relatively low

# 5. Results and Analysis

## 5.1 Custom `DecisionTree` Model

The custom `DecisionTree` model achieved the following results:

### 5.1.1 Best Hyperparameter Setting

The model performed best with the following hyperparameter settings:

- Impurity measure: Entropy
- Max depth: 11
- Prune: False
- Ratio: None

### 5.1.2 Training and Validation Accuracy

The model achieved a high training accuracy of 97.39%, indicating a good fit to the training data. The validation accuracy was also strong at 89.41%, suggesting that the model generalizes well to unseen data.

### 5.1.3 Test Accuracy

When tested on the independent test dataset, the model achieved an accuracy of 91.88%, confirming its ability to make accurate predictions on new data.

### 5.1.4 Feature Importance

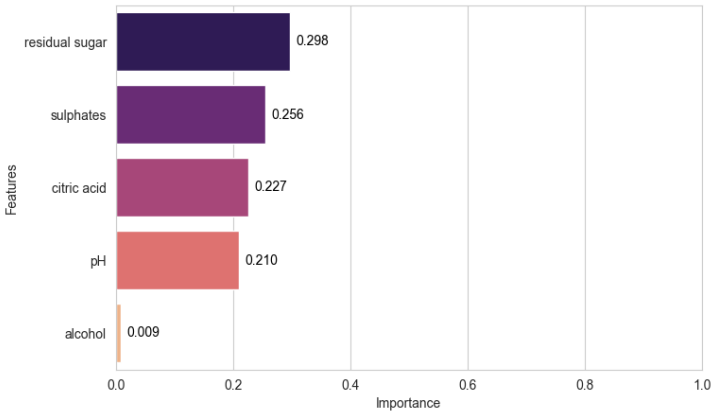


Figure 5: Custom `DecisionTree` Feature Importance.

The feature importance analysis revealed that *residual sugar* was the most influential feature, followed by *sulphates*, *citric acid*, *pH*, and *alcohol*. This information helps us understand which features are most relevant for making predictions.

### 5.1.5 Confusion Matrix

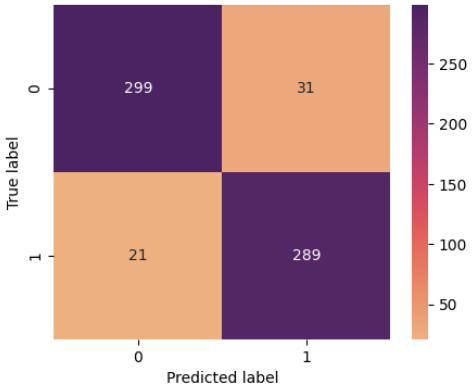


Figure 6: Custom `DecisionTree` Confusion Matrix.

The confusion matrix showed that the model had 299 true positives, 31 false positives, 21 false negatives, and 289 true negatives. This indicates that the model performed well in correctly classifying both classes, with a low number of misclassifications.

## 5.2 Scikit-learn's `DecisionTreeClassifier` Model

The scikit-learn's `DecisionTreeClassifier` model achieved the following results:

### 5.2.1 Best Hyperparameter Setting

The model's best hyperparameter configuration is as follows:

- Impurity measure: Entropy
- Max depth: 8
- Prune: False
- ccp_alpha: 0

### 5.2.2 Training and Validation Accuracy

The model achieved a training accuracy of 94.42% and a validation accuracy of 89.56%. These results indicate that the model performed well during training and generalizes effectively to new data.

### 5.2.3 Test Accuracy

When tested on the unseen test dataset, the model achieved an accuracy of 90.16%, confirming its ability to make accurate predictions.

### 5.2.4 Feature Importance

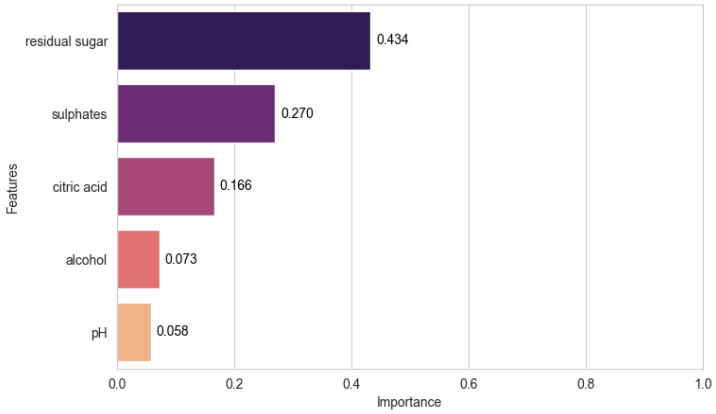


Figure 7: `DecisionTreeClassifier` Feature Importance.

The feature importance analysis indicated that *residual sugar* was the most influential feature, followed by *sulphates*, *citric acid*, *alcohol*, and *pH*. This information provides insights into the key features driving the model's predictions.

### 5.2.5 Confusion Matrix

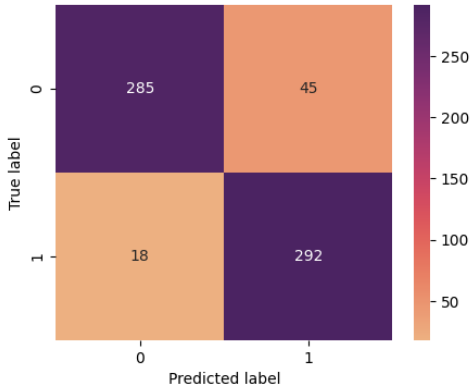


Figure 8: `DecisionTreeClassifier` Confusion Matrix.

The confusion matrix for this model showed 285 true positives, 45 false positives, 18 false negatives, and 292 true negatives. These results demonstrate a good balance between precision and recall.

## 5.3 Model Comparison

When comparing the custom `DecisionTree` model with the scikit-learn's `DecisionTreeClassifier`, the following observations can be made:

- Accuracy:
  - The custom model has a slightly higher training accuracy of 97.39% compared to 94.42% of the scikit-learn model. This might suggest that the custom model is fitting more closely to the training data.
  - Both models showcase similar performance on validation data, with the custom model achieving 89.41% and the scikit-learn version at 89.56%.
  - On the test dataset, the custom model slightly outperformed with an accuracy of 91.88%, compared to the 90.16% from the scikit-learn model.

- Speed:
  - The custom model took 0.55065 seconds to train, while scikit-learn's implementation trained in just 0.00453 seconds. This difference in performance is explained by the fact that scikit-learn has a highly optimized codebase that benefits from various performance improvements, efficient data structures, and algorithmic optimizations.
- Feature Importance:
  - Both models identified similar key features, which means there is a certain level of agreement in how each model interpreted the dataset. However, the importance values may differ slightly between the two.

The differences in accuracy and training time can be attributed to factors such as optimization, pruning strategies, hyperparameters, and code efficiency. Scikit-learn's implementation uses extensive optimization capabilities and efficient algorithms, which leads to shorter training times. The implemented custom model is highly accurate due to the choice of hyperparameters and can benefit from further optimization to improve training efficiency.