Decision Trees

The Art of Winning an Unfair Game, by Michael Lewis, brought sports analytics into the mainstream and reached a wider audience with its film adaptation, *Moneyball*. Since then, the field has evolved dramatically. Today, collecting and analyzing player statistics, team potential, and external factors like weather and location form the backbone of smarter betting strategies and serious financial upside. As of 2024, the sports analytics industry was estimated to be valued at $4.8 billion.[1]

Today, niche machine learning models are transforming the way we engage with sports data. In this post, we'll learn how a simple decision tree algorithm can predict the winners of European soccer matches.

What is a Decision Tree?

In my last post, in which I discussed common terminology and quotidian practices of ML. You may recall, **decision trees are a supervised predictive algorithm that can handle classification and regression problems**. As *non-parametric models*, decision trees don't assume any specific distribution in the input data - unlike linear or logistic regressions -, making them highly flexible for real-world datasets with complex or nonlinear relationships.

Our dataset comes from Kaggle in the form of a SQL database. It covers European soccer matches and includes features like team and player attributes (sourced from the EA Sports FIFA video game series), betting odds from up to 10 providers, and more. If you're curious about SQL data wrangling and feature engineering, check out the full workflow in the linked code file.

Before we build the model, let's understand the core structure of decision trees and their constraints.

---

[1]Research, Straits. "Sports Analytics Market Size, Share & Growth Forecast by 2033." *Straits Research*, straitsresearch.com/report/sports-analytics-market. Accessed 1 June 2025.

Structure

The structure of a decision tree resembles an upside-down tree and is composed of four elements: the root node, decision nodes, branches, and leaf nodes. A visual might help make this a bit more intuitive:

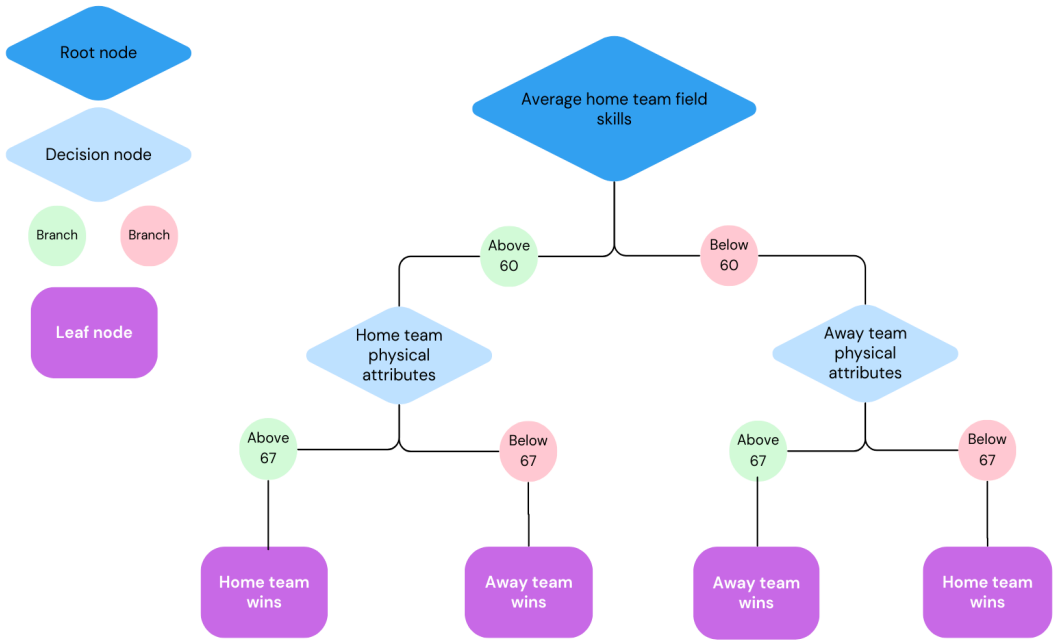


Figure 1

- The root node represents the entire dataset and the starting point for splitting.
- Decision nodes are points in a decision tree where the algorithm evaluates a feature and splits the data into branches based on specific conditions.
- Nodes are connected by branches that represent the outcome of the split in the dataset. Each branch corresponds to a possible value or range of the feature being split on.
- Leaf nodes express the final prediction.

How does a decision tree choose features to split on in decision trees? There are several methods to choose from, here are three of the most important ones:

1. Gini impurity

Gini impurity is a way to measure how mixed up the classes are in a group of data. If all the items belong to the same class, the Gini impurity is zero—meaning the group is perfectly "pure." But if there's a mix of different classes, the impurity goes up. Technically, it represents the expected probability of misclassifying a randomly chosen observation. To quantify this, gini impurity uses pairwise comparisons, specifically, the chance of two randomly selected items belonginf to the same class (calculated by squaring class probabilities). Subtracting this from 1 measures impurity. The Gini impurity is calculated using the formula:

$$1 - \sum_k p_k^{\,2}$$

Where p_k is the proportion of observations of class k in the node.

Decision trees use this formula to select the feature that reduces impurity the most at each split. Scikit-learn's implementation of decision trees uses the CART (Classification and Regression Trees) algorithm by default, which chooses splits based on the lowest Gini impurity.

2. Information Gain

Information Gain is another criterion used to evaluate the quality of a feature for a split in decision trees. It is based on the concept of entropy, which is just a fancy word for "disorder", and it measures the amount of disorder or uncertainty in a dataset. A dataset with mixed classes has higher entropy, while one with mostly or entirely one class has lower entropy.

Information Gain quantifies the reduction in entropy attained by splitting a dataset on a given feature. In other words, it measures how much "information" a feature provides about the class label. The more a split reduces entropy, the higher the information gain, and the more useful the feature is for classification. Entropy is calculated using the formula:

$$-\sum (p_k) \cdot log_2(p_k)$$

Information Gain of a feature is then computed as:

Entropy(parent) − Σ (weighted entropy of children).

Decision trees like ID3 use information gain to select the optimal split at each node. However, one limitation is that information gain tends to favor features with many unique values.

Both Gini impurity and entropy help decision trees figure out the best way to split the data, but they do it a little differently. Gini is based on squared probabilities and so is a bit simpler and faster while entropy comes from information theory and is more sensitive to changes in the data.

3. Gain ratio

Gain Ratio is a refinement of information gain that addresses one of its key weaknesses: favoring features with many unique values. For example, a feature like "ID" or "timestamp" could have high information gain simply because it uniquely splits nearly every row, despite offering no real predictive value.

To fix this issue, Gain Ratio introduces *intrinsic value*, which measures how broadly a feature splits the data. By calculating the entropy of the distribution of samples across the resulting child nodes, intrinsic value penalizes features with many distinct values that would otherwise cause the tree to split data too specifically. As a result, Gain Ratio favors features that lead to more meaningful, generalizable splits. The gain ratio is calculated as:

Information Gain / Intrinsic Value

Where the Intrinsic Value is: $-\sum (|S_i| / S) \times log_2(|S_i| / S)$

Here, S_i represents the number of observations in each child node, and S is the total number of observations at the parent node.

Gain Ratio was introduced in the C4.5 algorithm, which builds upon ID3 by improving split selection and tree pruning. It tends to produce more balanced and generalizable trees, especially in datasets with categorical features that have many unique values.

When Should You Use Decision Trees?

Decision trees are incredibly versatile tools to solve predictive problems. They are so versatile, that they can handle both regression and classification problems, adapting their approach depending on the type of target variable. For classification, decision trees split data by choosing features that reduce uncertainty about class labels—using metrics like Gini impurity or entropy—and assign the most common class in each leaf node. For regression, they focus on minimizing prediction errors by splitting based on variance reduction or mean squared error, with leaf nodes predicting the average value of the target variable in that subset. While both types share the same tree-like structure of nodes and branches, classification aims for pure groups of categories, whereas regression seeks accurate numerical predictions.

Can We Build It? Yes, We Can!

With Python, initializing and training a decision tree model is remarkably straightforward—it takes just two lines of code:

```
model = DecisionTreeClassifier(parameter_1,…,parameter_n)
```
or
```
model = DecisionTreeRegressor(parameter_1,…,parameter_n)
model.fit(X, y)
```

The first line sets up the model, and the second line uses the fit() function to learn the relationship between all the independent variables and the price. While you can create a decision tree with default settings like `DecisionTreeClassifier()`, scikit-learn offers several parameters to help control the tree's complexity and prevent overfitting:

1. max_depth: Limits the maximum depth (height) of the tree, i.e., the number of decision nodes from the root to any leaf. For example, in figure 1 the tree's height is 2.
2. min_samples_split: Minimum number of samples needed to split a node.
3. min_samples_leaf: Minimum number of samples required in a leaf node.
4. max_leaf_nodes: Sets the maximum number of leaves in the tree.
5. max_features: Limits the number of features considered when looking for the best split.
6. min_weight_fraction_leaf: Similar to min_samples_leaf but expressed as a fraction of the total sample weights.

7. min_impurity_decrease: A node will be split only if the impurity decrease is above this threshold.

These parameters afforded to us by Python's Scikit-learn library make it pretty easy to control the tree's size and shape, but what's going on inside the 'black box'? In machine learning, when we call something a 'black box', we mean that the internal process is hidden from us. Though scikit-learn makes it easy to use this function, we still want to understand what's going on inside.

Math in the Black Box

The goal of decision trees is to predict a continuous variable or a class of future observations given multiple features and a label (Y). In our European soccer match example, we're tackling a binary classification problem: predicting whether the home team wins (1) or doesn't win (0). Unlike linear models that use coefficients, decision trees like CART rely on recursive partitioning to split data into pure subsets.

Let's explore a decision tree that uses CART for feature selection. At its heart is Gini impurity, a measure of how mixed the classes are in a node. The formula is:

$$G = 1 - \sum_k p_k^2$$

To identify the best feature for a decision node, the tree evaluates all possible splits for each feature. For continuous variables, it considers various threshold values, while for categorical variables, it groups categories to maximize purity. For each candidate split, the tree calculates the Gini impurity for the resulting left and right child nodes, and then computes the *weighted average Gini impurity.* Weighted average Gini impurity combines the Gini impurity of each child node with the proportion of observations in each split, summing the resulting weighted Gini values. The mathematical denotation is:

$$G_w = \frac{n_L}{n} * G_L + \frac{n_R}{n} * G_R$$

Where n_L and n_R are the number of observations in the left and right child node, and G_L G_R are the gini impurity of the left and right child node respectively.

The tree repeats this process for all features and possible split points, selecting the split that produces the lowest weighted average Gini impurity. This feature and split point are used to partition the data at that node. The decision tree continues recursively, splitting the data further until a stopping criterion is met (such as maximum depth, minimum samples per leaf, etc), the nodes are pure, or all samples in a node have identical feature values.

Computational Mathematics

Manually building a decision tree is exciting because it is a predominantly recursion and logic-based coding problem that uses simple arithmetic. Let's walk through the process gradually.

Starting easy, we'll apply the logic that computes gini impurity using its formula:

```
classes, counts = np.unique(arr, return_counts=True)
probabilities = counts / counts.sum()
gini_impurity = 1 - np.sum(probabilities ** 2)
```

And do the same to compute the weighted average Gini impurity:

```
gini_left = gini_impurity(y_left)
gini_right = gini_impurity(y_right)
n_left = len(y_left)
n_right = len(y_right)
weighted_gini = (n_left / n) * gini_left + (n_right / n) * gini_right
```

Now, let's build a function that executes a decision tree's feature selection logic. Decision trees order the observations in each feature and retain a set of unique values. Once we have these unique values, thresholds for splitting are calculated as midpoints between every pair of consecutive values. For example, if a set {1,4,6,7,8} exists, its corresponding split thresholds would be 2.5,5,6.5,7.5. In code, you can find these thresholds using:

```
 thresholds = (unique_vals[:-1] + unique_vals[1:]) / 2
```

For each threshold, we'll split the feature into left and right sides (called mask in the code), based on whether the feature value is less than or equal to the threshold or greater than it. The weighted average Gini impurity is calculated for each split to evaluate the effectiveness of the split, and the algorithm chooses the split with the lowest weighted Gini impurity.

Once we have our supporting functions ready, it's time to build the core function that drives the decision tree logic. At each node, we store the node's Gini impurity as a baseline and keep track of the majority class, which will be used if we decide to stop splitting and create a leaf node. We also need to set up stopping criteria to prevent the tree from growing forever. The tree will stop splitting if it reaches the maximum allowed depth, if the node has fewer samples than a minimum threshold, or if the node is already pure. Here's what checking the stopping criteria might look like in code:

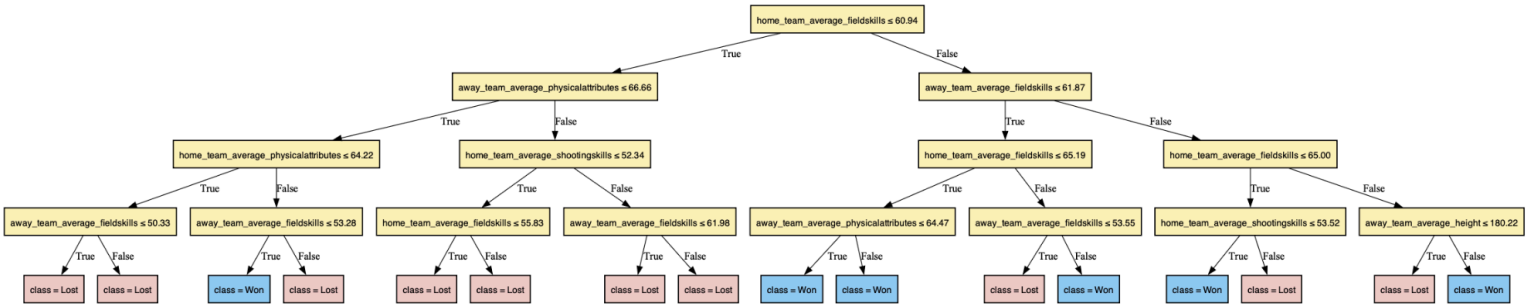
```
if (max_depth is not None and depth >= max_depth) or \
   (n_samples <= min_samples_leaf) or (current_gini == 0):
   return {'type': 'leaf', 'class': majority_class}
```

With these checks in place, the function can now proceed to select the best split. It iterates over all features and possible split points, using the best_gini_split function to find the split that leads to the lowest weighted Gini impurity. If no split improves the purity, the node becomes a leaf. Otherwise, the data is split into left and right subsets using the best feature and threshold, and the tree recursively builds subtrees for each side, increasing the depth by one each time:

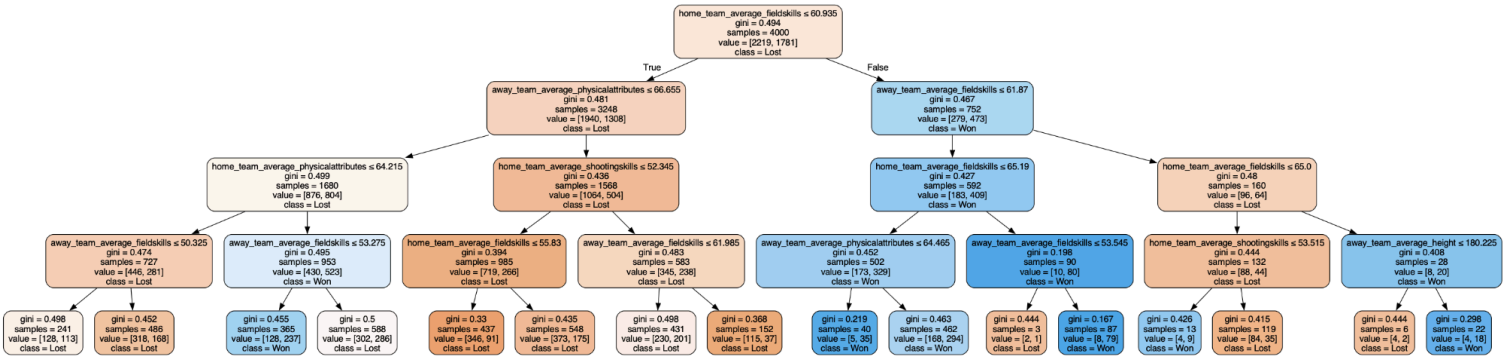
```
left_mask = X[:, best_feature] <= best_threshold
right_mask = ~left_mask
left_subtree = build_decision_tree(X[left_mask], y[left_mask], max_depth,
min_samples_leaf, depth + 1)
right_subtree    =    build_decision_tree(X[right_mask],    y[right_mask],
max_depth, min_samples_leaf, depth + 1)
```

The tree that results from this process is stored as a nested dictionary, where each node contains the best feature and threshold for splitting, and each leaf node contains the class prediction. If we visualized this manually built tree, it would show each decision point and

outcome just like the trees generated by Scikit-learn. While Scikit-learn's visualizations are more detailed, the underlying logic is exactly the same: split the data step by step to make the best possible predictions.



This is what the Scikit-learn decision tree looks like; the visualization has more detail but the tree essentially work in the same way:



Predictions

Now that our manual decision tree is built, how do we actually use it to predict the outcomes of new soccer matches? The process is straightforward: each new observation (or match) starts at the root of the tree and works its way down, following the decision rules at each node, until it reaches a leaf node that gives the final prediction.

To make this happen, we write a prediction function. At each decision node, the function checks which feature is being used for the split and compares the observation's value for that feature to the node's threshold. If the value is less than or equal to the threshold, the observation moves to the left child node; otherwise, it goes to the right child node. This process repeats—moving left

or right at each node—until the observation finally lands in a leaf node, which holds the predicted class.

Here's what this logic looks like in code:

```
while node['type'] != 'leaf':
        feature = node['feature']
        threshold = node['threshold']
        if sample[feature] <= threshold:
            node = node['left']
        else:
            node = node['right']
    return node['class']
```

Evaluation

The model has given us its predictions, let's see how it did. We use key evaluation metrics to measure its success: accuracy rate, precision, recall, and F1 score (view my previous post to learn more about these metrics.)

Here is the confusion matrix for our decision tree model predicting whether the home team would win a match:

[[476 82]
[270 172]]

This matrix tells us that out of 1,000 matches:

- The model correctly predicted 476 home losses (true negatives).
- It correctly identified 172 home wins (true positives).
- There were 82 cases where the model predicted a home win when it was actually a loss (false positives).
- There were 270 cases where the model predicted a home loss when the home team actually won (false negatives).

While the model demonstrates some ability to distinguish between home wins and losses, there is a notable imbalance in its predictions. The high number of false negatives (270) suggests

that the model often fails to identify home wins, while the 82 false positives indicate some over-prediction of home victories.

Let's break down the key metrics:

1. Accuracy: 65%

With an accuracy of 65%, the model correctly predicts the match outcome about two-thirds of the time. While this result is better than random guessing, it also signals that there is significant room for improvement, especially in capturing home wins.

2. Precision: 67.7%
    - Home Loss (class 0): 64%
    - Home Win (class 1): 68%

Precision for home wins is 68%, meaning that when the model predicts a home win, it is correct about two-thirds of the time. This is a reasonable starting point, but there is a nontrivial rate of false positives.

3. Recall: 38.9%
    - Home Loss (class 0): 85%
    - Home Win (class 1): 39%

The recall for home wins is only 39%. This means the model misses a majority of actual home wins, which is a major limitation if our goal is to reliably identify matches where the home team will succeed.

4. F-1 score: 49.4%
    - Home Loss (class 0): 0.73
    - Home Win (class 1): 0.49

The F1-score for home wins is 0.49, reflecting the model's struggle to balance precision and recall for this class. The macro average F1-score is 0.61, and the weighted average is 0.63, both indicating moderate overall performance.

Interpretation:

From evaluating our metrics, it's clear that the model is more effective at predicting home losses (high recall and F1-score for class 0), it struggles to identify home wins, as shown by the lower recall and F1 for class 1. In practical terms, this means the model is cautious in predicting home wins and tends to err on the side of predicting losses. This behavior could be due to class imbalance or features that are not strongly predictive of home victories.

In many real-world prediction tasks, not all mistakes carry the same consequences—a concept known as *asymmetric costs of errors*. For example, in our match outcome model, predicting a home win when the team actually loses (a false positive) may have very different implications compared to missing a true home win (a false negative), especially if these predictions inform betting, ticket sales, or team strategy. Traditional evaluation metrics like accuracy or mean squared error treat all errors equally, but in practice, the cost of overestimating and underestimating outcomes can be quite different. By recognizing and, when appropriate, incorporating asymmetric error costs into model evaluation or training, you can better align your models with the specific risks and objectives of the application, ultimately leading to more effective and context-aware decision-making.

Overall, these metrics suggest that while the decision tree offers some predictive value, especially for home losses, there is considerable opportunity to enhance its ability to detect home wins. Future improvements could include engineering additional features, tuning tree parameters, or experimenting with ensemble methods to boost recall and balanced accuracy for both classes.

Conclusion

Decision trees offer a transparent and intuitive approach to classification problems, making them a popular choice for both exploratory analysis and predictive modeling. In our attempt to forecast home team victories, the decision tree provided a clear framework for understanding how different features contribute to match outcomes. While the model achieved moderate accuracy and demonstrated reasonable precision when predicting home wins, it

struggled to capture a large portion of actual victories, as reflected in its lower recall and F1-score.

This evaluation highlights the importance of not relying solely on overall accuracy, especially in scenarios where class imbalance or asymmetric costs of errors exist. By examining the confusion matrix and key metrics, we gained valuable insight into the model's strengths - such as its reliability in predicting home losses - and its limitations, notably in identifying true home wins.

Ultimately, the process of building, evaluating, and interpreting a decision tree model deepens our understanding of both the data and the predictive task at hand. It also underscores the value of iterative improvement: through feature engineering, parameter tuning, or exploring ensemble methods, we can strive to enhance the model's ability to generalize and deliver actionable insights. Mastering decision trees not only strengthens our analytical toolkit but also lays a solid foundation for tackling more sophisticated machine learning challenges in the future.

In this post, we've learned:

- What decision trees are
- The structure of a tree
- The core concepts involved a decision tree's feature selection process
- How to build a decision tree using the scikit-learn library in Python
- How to build a decision tree manually in Python
- How to predict values using scikit and manually in Python
- How to evaluate the model using a confusion matrix, accuracy, precision, recall, and F1 metrics with Scikit-learn
- How to interpret model evaluations
- As a bonus: How to create the visualize decision trees

That's a wrap on decision trees! Next, we'll study the combined power of multiple decision trees or *ensemble learning techniques* - stay tuned!