Decision Trees For Classification in Finance: A Beginner's Guide

Univeristy of Essex Equity Fund Quantitative Research Division
Amjad Saidam
email amjadsaidama@gmail.com

March 2025

Contents

Outline

This short paper covers the basics of decision trees for classification from a data-science and finance perspective. We cover the the general mathematics behind their algorithms, a base python implementation and then finish with building a trading strategy which we back-test from our fitted tree

1 What is a Decision Tree

This is a non-parametric (no parameter estimates are required such as weights and biases which are recursively updated in a neural network) **supervised machine learning** method.

"The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features." Bibliography [1]

1.1 Advantages and Disadvantages

Advantages of decision trees, not limited to the following include

- **Easy interpretability**. Decision trees are a "White box" model, as the boolean nature of decision tree makes interpretation of model outputs intuitive.
- **Fast model training**. Decision tree cost in prediction is logarithmic in the number of data points used to train the model, Bibliography [1].

Disadvantages of decision trees include, but are not limited too

- **Un-optimality**, optimisation Algorithms used to split data cannot guarantee to return a globally optimal tree.
- **Sensitive to data**. If a class label dominates, then may lead to a bias tree. It is therefore recommended to balance class labels before fitting.

2 Maths Behind Decision Trees

Given some set of n dimensional feature vectors $x_i \in R^n$ and a response or "label" vector $y \in R^l$ where $l = n$. A decision-tree will recursively split the feature space (the data) such that samples with the same labels are grouped together.

The decision tree algorithm in laymen terms will at each stage look for the feature and threshold that best splits the data labels, with the later being determined by some loss function that calculates the "purity" of the split. This process is recursive in nature as it is repeated until some boundary / stopping condition is met such as max tree depth or min samples per leaf node is met (number of samples in node m, n_m is less than or equal to some integer value).

2.1 Decision Tree Maths

At each node m the dataset is represented as Q_m, and each node Q_m has N_m samples.

For each candidate split (only one of the computed splits is used - that is the optimal split) $\theta = (j, t_m)$, with j = feature vector index and t_m = some arbitrary threshold value, we split the dataset in to two subsets $Q_m^{left}(\theta)$ and $Q_m^{right}(\theta)$ where

$$Q_m^{left}(\theta) = \{(x, y) \,|\, x_j \leq t_m\} \tag{1}$$

$$Q_m^{right}(\theta) = Q_m(\theta) \setminus Q_m^{left}(\theta) \tag{2}$$

Note that "\" means take all the data which is not in $Q_m^{left}(\theta)$. We have now computed our data set splits for some feature j. However to insure we choose the best j and t_m that splits the data, such that we have the best partition of labels, we apply some loss function $H(Q)$ to each $Q_m^{\{left,right\}}$ and then calculate the weighted average loss of this split $G(Q_m, \theta)$ often referred to as the **impurity**. The parameter combination $\theta = (j, t_m)$ that yield the lowest loss are our "optimal" parameters. We can formalise this procedure and define an optimisation problem θ^*.

$$\theta^* = \arg\min_\theta G(Q_m, \theta) \tag{3}$$

This optimisation problem can be though of as finding the θ value, i.e. the feature and corresponding threshold that minimises our split loss function G

2.2 Loss Functions

For binary classification of outcome k we define $H(Q_m)$ as the **Gini** function, where

$$H(Q_m^{\{left,right\}}) = \sum_{k\in\{0,1\}} p_{mk}(1 - p_{mk}) = 1 - \sum_{k=\{0,1\}} p_{mk}^2 \tag{4}$$

Where

$$p_{mk} = \frac{1}{n_m} \sum_{x_i \in Q_m^{\{left,right\}}}^{n_m} \mathbb{1}\{y_i = k\} \tag{5}$$

So that p_{mk} is the fraction of label k in some node m. We can now define are models prediction which it forms in sample, for some terminal node, $m = M$, our models prediction at that "leaf" will be $\arg\max_k\{p_{mk}\}$. *note that p_{mk} is computed of each of the two nodes in a split, so p_{mk} can also be written as $p_{mk}^{\{left,right\}}$*

To compare the loss function value, $H(Q)$, for some level m in our tree we calculate a weighted loss value, $G(H(Q_m))$ which we define here.

$$G(Q_m^{left}, Q_m^{right}, \theta) = \frac{n_m^{left}}{N_m} H(Q_m^{left}) + \frac{n_m^{right}}{N_m} H(Q_m^{right}) \tag{6}$$

We can now redefine our optimisation function, [3], for the classification case

$$\min_{\theta=(j,t)} \left[G(Q_m^{left}, Q_m^{right}, \theta)\right]. \tag{7}$$

2.3 Prediction

Given some unseen observation $\tilde{x}_i \in \tilde{Q}$, where $\tilde{Q}$ represents the unseen / validation data set, our fitted decision tree will output as follows

$$\hat{f}(\tilde{x}_i) = \sum_{m=1}^{M} c_m \mathbb{1}\{\tilde{x}_i \in R_m\} \tag{8}$$

Where

- c_m = The constant prediction value, in the classification case c_m will be the majority class probability for some node m in our tree.
- R_m = The leaf nodes of our tree.

Our prediction function $\boxed{8}$ takes data as input and outputs the predicted class k. In the classification case the prediction function simplifies to c_m as the summation collapses to the leaf node m the data is classified in. In this case c_m will tell the $P(k = 1|x_i \in R_m)$ which can simply be computed using $\boxed{5}$, i.e. the confidence with which we classify our unseen data x_i.

2.4 Decision Tree Learning Algorithm

We can generalise our procedure in **section 2.1** and write some pseudo code outlining how the decision tree algorithm works.

Algorithm 1 Decision Tree Learning

```
Require: Training dataset $Q$, impurity function $H()$, minimum samples per node
    min_samples, maximum depth max_depth
Ensure: Decision tree root node
 1: function TRAINTREE($Q$, depth)
 2:     $n_m \leftarrow$ number of samples in $Q$
 3:     if $n_m <$ min_samples or depth $\geq$ max_depth then
 4:         return Leaf Node with majority class (for classification) or mean value (for re-
    gression)
 5:     end if
 6:     $\theta^* \leftarrow \arg\min_\theta G(Q, \theta)$                              ▷ Find best split
 7:     $Q_m^{\text{left}}, Q_m^{\text{right}} \leftarrow$ Partition($Q, \theta^*$)
 8:     Node $N \leftarrow$ Create node with split $\theta^*$
 9:     $N$.left $\leftarrow$ TRAINTREE($Q_m^{\text{left}}$, depth + 1)
10:     $N$.right $\leftarrow$ TRAINTREE($Q_m^{\text{right}}$, depth + 1)
11:     return $N$
12: end function
```

3 Regression Trees

In this brief chapter we will show that decision trees are a simple extension of regression trees, with the only change being in the definition of the splitting function, which we define as follows with respect to some split (here the first) (9.13) Bibliography [4]

$$\min_{\theta=(j,t)} \left[\min_{c_1} \sum_{x_i \in Q_m^{left}(\theta)} (y_i - c_m^{left})^2 + \min_{c_2} \sum_{x_i \in Q_m^{right}(\theta)} (y_i - c_2^{right})^2 \right]. \tag{9}$$

We can see our optimisation problem is no longer a function of the same loss function $H(Q)$, rather $H(Q)$ is now defined as the sum of squares, with c_m being the means of the label for some split.

To solve this problem we proceed with a greedy algorithm that iterates over all values $\theta = (j, t)$. The optimisation problem is solved when we yield the lowest binary partition of sum of squares.

To yield $\boxed{7}$ we make a substitution by replacing the sum of the sum of squares with $\boxed{6}$.

$$\min_{\theta=(j,t)} \left[G(Q_m^{left}, Q_m^{right}, \theta) \right]. \tag{10}$$

4 Base Python Implementation

We now build a decision tree classifier in python from scratch, The algorithm we write is provided in Appendix [1], For the complete code with implementation see Bibliography [2].

For "depth $= 3$" we have 7 leaf nodes. In general a decision tree of depth n will have a maximum of 2^n leaf nodes. A fewer number of nodes suggests that the we split a pure leaf node at a higher level. The total number of nodes in a tree scale according too

$$T(n) = 2^{n+1} - 1$$

To calculate the growth rate of this algorithm (the total number of nodes) we need to conduct a rate of change calculation i.e. differentiate the algorithm.

$$\begin{aligned}\frac{dT}{dn} &= \frac{d}{dn}(2^{n+1} - 1)\\ &= 2^{n+1} ln(2)\end{aligned}$$

The first term dominates so the number of nodes in a decision tree grows exponentially.

5 Practical Implementation in Python using Scikit-Learn

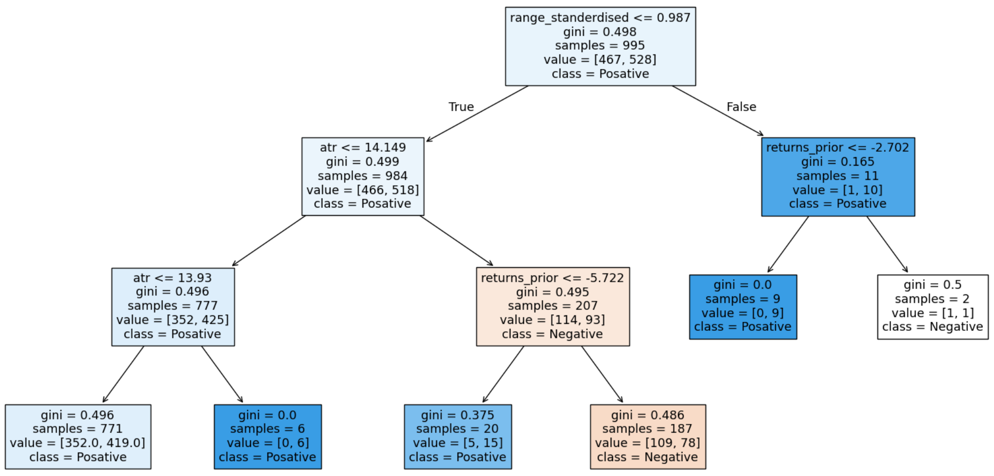

Figure 1: *5 Split Fitted Decision-Tree, tree output, Bibliography [3]*

We are concerned with the financial application of this ML method. One simple and very common example is to use decision trees to predict the state of the market next day, will it rise (positive return day) or go down (negative return day). *See Bibliography* [3] *for google colabortory python notebook*

We can now form a trading strategy using our fitted decision tree in Figure 1. Our decision to buy / sell the asset is formalised by 7. Intuitively our trading strategy is very simple, we pass our features through our fitted tree until we reach the leaf node, this node's majority class serves as our models prediction. Trade exits and position sizing is arbitrary and their is alot of scope for experimentation but for generalisability we will assume 100% account balance investment, re-investment of profits/loss, no-leverage, no stop loss / take profit and state based exists where we exist the position after 1 day (This strategy will always hold a position in the market Bibliography [3]). The resulting strategy return based performance is shown below

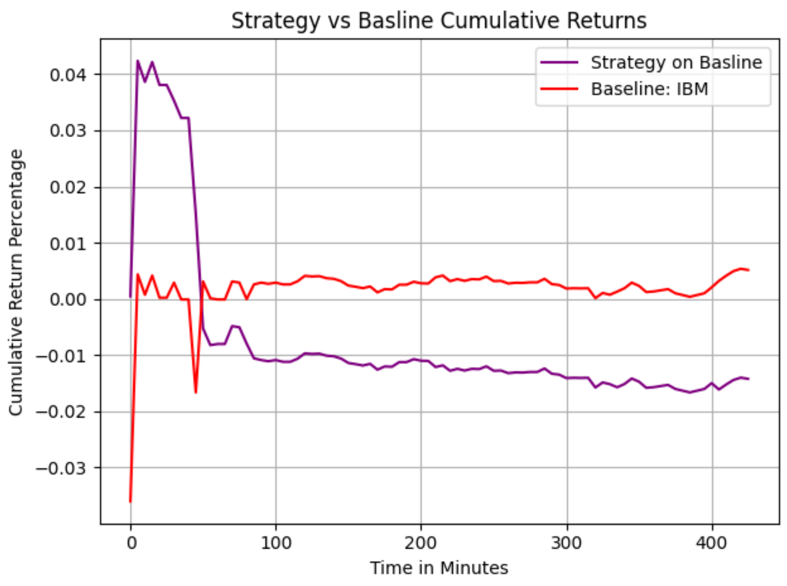

Figure 2: *The trading strategy corresponding to the fitted decision tree in Figure 1 on IBM 5 minute data from "2024-03-28 10:35:00" to "2024-03-28 19:55:00" in Figure 1. Bibliography [3]*

We can see our strategy has underperformed its underlying in the given back-tested time frame. It is also important to note this strategy was test on 5 minute data as oppose to daily data which was used to fit the decision tree. This is unlikely to be an issue due the fractal nature of price data being a stylised fact in finance.

5.1 Conclusion

We have defined decision trees for classification as a supervised machine learning method that is used to predict class labels. We have outlined the general mathematics behind this process and understood that decision tress work by recursivelly splitting our data based on some loss function and continue to do so until a boundary condition is true, the model then forms its prediction by outputting the class label with the majority vote. We then proceeded to write a decision tree classifier in base python and using sickit-learn. We finish by presenting a quantitative application in which we construct a simple trading strategy. We conclude with saying that decision trees are a useful machine learning technique with much scope for application, including in trading strategy building, and therefore should be a skill in every quants toolbox.

6 Appendix

1.
```
class DecisionTree:
    def __init__(self, data, label, depth=0, max_depth=5,
                 min_data_split=2):
        self.data = data
        self.label = label
        self.depth = depth
        self.max_depth = max_depth
        self.min_data_split = min_data_split
        self.left = None
        self.right = None
        self.is_leaf = False
        self.predicted_label = None
        self.predictions = []  # Store predictions at each node
        self.feature = None
        self.threshold = None

    def __gini(self, labels):
        """Computes the Gini impurity for a set of labels."""
        if len(labels) == 0:
            return 0
        unique_labels, counts = np.unique(labels,
                                          return_counts=True)
        probabilities = counts / len(labels)  # class probabilities
        gini = 1 - np.sum(probabilities ** 2)
        return gini

    def BestSplit(self):
        """Finds the best feature and threshold to split on."""
        if len(self.data) == 0:
            return None, None

        best_feature = None
        best_threshold = None
        best_impurity = float('inf')

        for feature in self.data.columns:
            sorted_values = self.data[feature].sort_values().unique()
            mid_points = (sorted_values[:-1] + sorted_values[1:]) / 2

            for threshold in mid_points:
                left_mask = self.data[feature] <= threshold
                right_mask = self.data[feature] > threshold

                left_labels = self.label[left_mask]
                right_labels = self.label[right_mask]

                if len(left_labels) == 0 or len(right_labels) == 0:
                    continue

                impurity_left = self.__gini(left_labels)
                impurity_right = self.__gini(right_labels)
                weighted_impurity = (
                    (len(left_labels) / len(self.label)) * impurity_left +
                    (len(right_labels) / len(self.label)) * impurity_right
                )

                if weighted_impurity < best_impurity:
                    best_impurity = weighted_impurity
                    best_feature = feature
                    best_threshold = threshold

        if best_feature is None:
            return None, None

        return {'feature': best_feature,
                'threshold': best_threshold}, best_impurity

    def __prediction(self, predictedlabel, nodelabel,
                    treedepth, nodegini):
        # Calculate ratio of 0s to 1s
        unique_labels, counts = np.unique(nodelabel,
                                          return_counts=True)
        label_counts = dict(zip(unique_labels, counts))
        zero_one_ratio = label_counts.get(0, 0) / max(
            label_counts.get(1, 1), 1)  # Avoid division by zero

        # Create prediction dictionary with node metadata
        node_prediction = [{
            'predicted_label': predictedlabel,
            'depth': treedepth,
            'gini': nodegini,
            'zero_one_ratio': zero_one_ratio
        }]
        return node_prediction

    def Train(self):
        """Recursively trains the decision tree and stores
        predictions with metadata at each node."""
        # Early stopping conditions
        if (self.depth >= self.max_depth or
            len(self.data) <= self.min_data_split):
            self.is_leaf = True
            self.predicted_label = self.label.mode().iloc[0]
            self.predictions = self.__prediction(
                self.predicted_label, self.label,
                self.depth, self.__gini(self.label))
            return

        best_split, _ = self.BestSplit()
        if best_split is None:
            self.is_leaf = True
            self.predicted_label = self.label.mode().iloc[0]
            self.predictions = self.__prediction(
                self.predicted_label, self.label,
                self.depth, self.__gini(self.label))
            return

        self.feature = best_split['feature']
        self.threshold = best_split['threshold']

        left_mask = self.data[self.feature] <= self.threshold
        right_mask = self.data[self.feature] > self.threshold

        left_data = self.data[left_mask]
        left_label = self.label[left_mask]
        right_data = self.data[right_mask]
        right_label = self.label[right_mask]

        # Check if split results in empty subset
        if len(left_data) == 0 or len(right_data) == 0:
            self.is_leaf = True
            self.predicted_label = self.label.mode().iloc[0]
            self.predictions = self.__prediction(
                self.predicted_label, self.label,
                self.depth, self.__gini(self.label))
            return

        # Create left and right children
        self.left = DecisionTree(
            left_data, left_label, self.depth + 1,
            self.max_depth, self.min_data_split)
        self.right = DecisionTree(
            right_data, right_label, self.depth + 1,
            self.max_depth, self.min_data_split)

        # Recursively train the children
        self.left.Train()
        self.right.Train()

        # Collect predictions from child nodes
        self.predictions = (self.left.predictions +
                            self.right.predictions)

    def Predict(self, X):
        """Predict labels for input data X."""
        if self.is_leaf:
            return self.predictions

        predictions = []
        for _, row in X.iterrows():
            if row[self.feature] <= self.threshold:
                pred = self.left.Predict(pd.DataFrame([row]))[0]
            else:
                pred = self.right.Predict(pd.DataFrame([row]))[0]
            predictions.append(pred)

        return predictions
```

7 Bibliography

1. `https://scikit-learn.org/stable/modules/tree.html`
2. `https://colab.research.google.com/drive/1gpKGjEQ9N_pULRy9riF81vxsajlBA 47y?usp=sharing`
3. `https://colab.research.google.com/drive/1ycKGmKU4Wbuk3fy1s-IOELKxHybKL 8jK?usp=sharing`
4. `https://hastie.su.domains/ElemStatLearn/`