C4.5 Decision Tree Algorithm

Contents

C4.5 Decision Tree

Google Colab link to C4.5 implementation

Introduction

This report will detail the logic and implementation of a method of machine learning classification called the C4.5 Decision Tree algorithm. This algorithm can be used to effectively make classification predictions about incomplete data samples. However, it must first be fitted using a large dataset of completed data samples. During the fitting process the algorithm designs a series of questions to ask about incomplete data samples which it can then use to make a prediction about the incomplete data samples.

Algorithm Implementation

Decision Tree Predictions

Once fitted to a sample of a dataset, a decision tree (DT) uses a series of decisions in order to make a classification prediction. Two main components of a fitted DT are decision nodes and leaf nodes. Decision nodes contain a threshold that is compared against a feature (aka attribute) of a given data sample. If the features of the data sample meets the threshold conditions, it will then be directed to the node on the right (right child) for further processing. Otherwise, it will be directed to the node on the left (left child) for further processing. This process is repeated until a leaf node is reached. A leaf node has no children, but instead contains a decision value which is what the DT returns as it's predicted classification for that data sample. An illustration of a DT is represented in Figure 1. In this example, x_i presents the i^{th} feature in the sample (decision node), and y^{pred} represents a decision made by the algorithm based on the features in the data sample (leaf node). At the first node (aka root node) the threshold is 5. If the value of feature x_1 is less than 5, then the algorithm continues on to the left node. Otherwise, if it is greater than or equal to 5, the algorithm will continue on to the right node.

Image Source

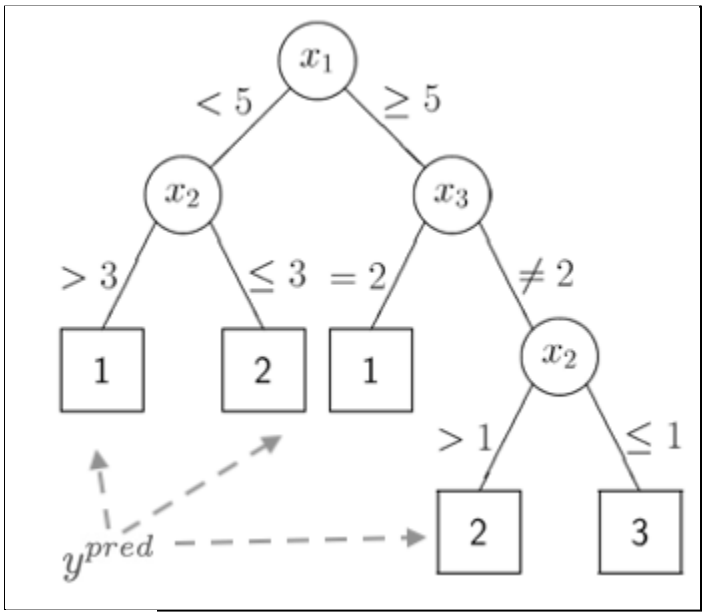

Figure 1

Any prediction by a DT is impossible without first *'fitting'* the DT with thresholds, decision nodes, and leaf nodes. To fit a DT, we give it samples of a dataset containing features and labels (classification results). Using these completed data samples, the decision tree algorithm can assess which feature values result in each classification. In other words, the algorithm learns which ranges of certain features in a data sample will result in each classification (label). The result of this fitting process is a DT similar to the one shown in Figure 1, which can then be used to make predictions about incomplete data samples which have not yet been classified (has no label value).

Decision Tree Fitting

In this section we will investigate how the DT 'learns' which feature values result in each classification.

Calculating Information Gain

When fitting a DT the dataset is split into two at each decision node. The aim is to continue to split the training dataset until there is only one label category (for example 'versicolor' or 'virginica') present in the remaining samples of the training dataset. At that point a leaf node can be placed with a classification decision.

A large dataset can be split in a number of different ways with different samples on either side of the split. For example, for the below data we could have a split of *'Sample 1'* and *'Sample 2'* on

the left side, and then ‘*Sample 3’* on the right side. Alternatively we could have *‘Sample 1’* and *‘Sample 3’* on the left and *‘Sample 2’* on the right. It is intuitively clear that the number of possible splits has an exponential relationship with the number of samples in the training dataset.

At each decision node the selected split should maximise the occurrence percentage of any one label on both sides of the split. For example, using a training dataset with labels **[*‘versicolor’, virginica’, ‘versicolor’, ‘virginica’*]** the best split would be:

[*‘versicolor’, ‘versicolor’*] and [*‘virginica’, ‘virginica’*]

With this split we can say with a level of certainty that the samples on the left side are classified as *‘versicolor’* and the samples on the right are *‘virginica’*. An example of a less optimal split would be:

[*‘versicolor’, ‘virginica’*] and [*‘versicolor’, ‘virginica’*]

With this split we can not say with certainty that samples on either side are either *‘versicolor’* or *‘virginica’*.

During the fitting process the C4.5 algorithm will assess all possible ways to split the data and choose the most optimal split. The measure of the optimality of a split is called **information gain**. This is done by calculating the probability of each label on each side of a split and finding the split with the most certain overall probability on either side. Using the earlier example:

[*‘versicolor’, ‘versicolor’*] and [*‘virginica’, ‘virginica’*]

There is a 100% probability of a *‘versicolor’* classification on the left side and a 100% probability of a *‘virginica’* classification on the right side. However, with the less optimal example:

[*‘versicolor’, ‘virginica’*] and [*‘versicolor’, ‘virginica’*]

There is a 50/50 probability of classifying *‘versicolor’* or *‘virginica’* on both sides of the split.

C4.5 has two possible methods of calculating the probability on each side of the split; **entropy** and **gini index** (aka gini impurity).

Entropy

Entropy measures the level of *uncertainty* in a dataset using the formula below.

$$\text{Entropy (E)} = \sum_{i=1}^{n} - p_i \ log_2(p_i)$$

$$p_i = probability\ of\ label\ i\ = \frac{no.\ label\ i\ values}{no.\ total\ label\ values}$$

$$n = number\ of\ label\ categories$$

Entropy is considered less efficient than Gini Index as the logarithmic calculation increases the processing power required. However, entropy at times will yield a better DT. For our most optimal example:

[*'versicolor', 'versicolor'*] and [*'virginica', 'virginica'*]

Entropy is equal to $(-\frac{2}{2}log_2(\frac{2}{2})) + (-\frac{2}{2}log_2(\frac{2}{2})) = 0$ on both sides of the split. However, for the less optimal split:

[*'versicolor', 'virginica'*] and [*'versicolor', 'virginica'*]

Entropy is equal to $(-\frac{1}{2}log_2(\frac{1}{2})) + (-\frac{1}{2}log_2(\frac{1}{2})) = 1$ on both sides, as $p_i = \frac{1}{2}$ for each side of the split.

As we have seen, the lower the entropy, the better. However the maximum value the entropy algorithm can yield is dependent on the number label categories. The maximum entropy value is approximately equal to $\sqrt{no.\ label\ categories}$. In the examples discussed we have 2 label categories, therefore the maximum entropy value is approximately 1.4142…

Gini Index

Gini Index measures the purity/impurity of a dataset. A dataset is said to be pure when the majority of its labels are of the same category. The Gini Index will always produce a value between 0 and 1. Gini Index is calculated using the formula below:

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

$$p_i = probability\ of\ label\ i\ = \frac{no.\ label\ i\ values}{no.\ total\ label\ values}$$

$$n = number\ of\ label\ categories$$

For our most optimal example:

[*'versicolor', 'versicolor'*] and [*'virginica', 'virginica'*]

Gini is equal to $1 - (\frac{1}{1})^2 = 0$ on both sides of the split.

For out less optimal example:

[*'versicolor', 'virginica'*] and [*'versicolor', 'virginica'*]

Gini is equal to $1 - (\frac{1}{2})^2 = 0.5$ on both sides of the split.

Information Gain

The Information Gain of a split is calculated using the gain (calculated by either the Entropy or Gini Index) of both sides of the split. Information Gain is calculated using the formula below:

$$Information\ Gain\ (IG) = g(parent)\ - \sum_{i=1}^{2} w_i\ g(child_i)$$

$$w_i = weight_i = \frac{no.\ values\ in\ parent\ node}{no.\ values\ in\ child\ split}$$

$$g = Entropy\ or\ Gini\ Index$$

For our most optimal example:

[*'versicolor', 'versicolor'*] and [*'virginica', 'virginica'*]

When using Entropy, information gain will equal: $1 - ((\frac{4}{2} \times 0) + (\frac{4}{2} \times 0)) = 1$

When using Gini Index, information gain will equal: $0.5 - ((\frac{4}{2} \times 0) + (\frac{4}{2} \times 0)) = 0.5$

For our less optimal example:

[*'versicolor', 'virginica'*] and [*'versicolor', 'virginica'*]

When using Entropy, information gain will equal: $1 - ((\frac{4}{2} \times 1) + (\frac{4}{2} \times 1)) = -3$

When using Gini Index, information gain will equal: $0.5 - ((\frac{4}{2} \times 0.5) + (\frac{4}{2} \times 0.5)) = -1.5$

Thresholds

As discussed in the previous section, a threshold is a value that is compared against a feature value of a given data sample. This is what dictates if the algorithm will continue to the left or right child of a decision node. But how does the algorithm select a threshold value and which feature to apply the threshold to?

These values have a direct impact on how the training data is split. Therefore the threshold and the feature being evaluated should be whichever values yield the split with the best information gain (the best split). To ensure this the algorithm iterates over each of the features. For each feature it establishes all of the possible thresholds as all occurring values for each feature of the current feature. For example, all possible thresholds of feature 3 in the table below are 1.4 and 1.3. We then try each possible threshold and find which threshold yields the best split.

Index	*feature 1*	*feature 2*	*feature 3*	*feature 4*	*label*
Sample 1 Values	5.1	3.5	1.4	0.2	'versicolor'
Sample 2 Values	4.9	3.0	1.4	0.2	'versicolor'
Sample 3 Values	4.7	3.2	1.3	0.2	'virginica'

If we assess feature 1 for the best split, the results will be as follows.

Threshold = 5.1
Decision = is the feature 1 value >= 5.1?
Split = ['versicolor'] and ['versicolor', 'virginica']

Information Gain (entropy) ≈ 0.2516

Threshold = 4.9
Decision = is the feature 1 value >= 4.9?
Split = [‘versicolor’, ‘versicolor’] and [‘virginica’]
Information Gain (entropy) ≈ 0.9183

Threshold = 4.7
Decision = is the feature 1 value >= 4.7?
Split = [‘versicolor’, ‘versicolor’, ‘virginica’] and [*empty*]
Information Gain (entropy) ≈ 0

We have found that for feature 1 the best threshold is 4.9 as it yields the best (highest) information gain. Threshold 4.9 for feature 1 will be pinned as our best split so far. This process will then be repeated for feature 2, 3, and 4. If a better feature threshold pair is found (a pair that produces a better information gain than our current best), then it will become the new best threshold and feature pair.

For further clarity, this process is outlined in recursive steps below:

1. View first/next feature
2. View first/next possible threshold in the current feature
3. Calculate information gain of current threshold. If it is the best information gain so far, store the current feature and the current threshold.
4. Go back to step 2 until all thresholds in the current feature have been viewed.
5. Go back to step 1 until all features have been viewed.

Steps to Fit a Decision Tree

1. If there is 1 sample in the training dataset OR if all labels are the same, generate a leaf node with the remaining label as the decision and skip steps 2, 3, and 4.
2. Otherwise, find the best split (by finding the threshold and feature pair with the best information gain).
3. Generate a decision node with the threshold and feature used to generate the best split.
4. For each side of the split, start from step 1.

Hyperparameters

When fitting a DT, hyper-parameters can be given in order to influence the resulting DT. These hyper-parameters are listed below.

- **max_depth:** The maximum depth the DT can grow to. Depth refers to the number of nodes that must be passed in order to reach the lowest part of the tree. The root node has depth zero, and each child of a root node has depth 1. For example the DT shown in Figure 1 has a maximum depth of 3. By default, (no max_depth setting) the tree continues to grow until it runs out of samples to split.

- **max_features:** The maximum number of features from the training dataset to be used in the fitting (training) process. For example, if a training dataset has 4 features, but max_features = 1, then a random three of the four features will be rejected and only one feature will be used for fitting. By default, (no max_features setting) all features are used. This hyperparameter can take five different types of inputs.

    - **'Sqrt'**: max_features will equal the square root of the total number of features in the training dataset.
    - **'auto'**: max_features = 'sqrt'
    - **'log2'**: max_features will equal log_2 of the total number of features in the training dataset.
    - **float value**: max_features will equal the float value divided by the number of features. Note, this float value represents a fraction.
    - **integer value**: max_features will equal the integer value.

- **min_samples_split:** The minimum number of samples required to make a split (decision node). Default value is 2.

- **min_samples_leaf:** The minimum number of samples required to be at a leaf node. Default value is 1.

- **info_gain_method:** A specification of which algorithm to use when calculating information gain ('gini' or 'entropy'). Default value is 'entropy'.

Steps to Fit a Decision Tree with **max_depth**

1. If there is 1 sample in the training dataset OR if all labels are the same, generate a leaf node with the remaining label as the decision and skip steps 2, 3, 4, and 5.
2. Otherwise, if the current depth is equal to the max_depth value, generate a leaf node using the most popular of the remaining labels as the decision.
3. Otherwise, find the best split (by finding the threshold and feature pair with the best information gain) using entropy.
4. Generate a decision node with the threshold and feature used to generate the best split.
5. For each side of the split, start from step 1 using the split as the training dataset.

Steps to Fit a Decision Tree with **max_features**

1. If the number of features in each sample of the training dataset is greater than the max_features value, then randomly select max_feature number of features and reject any remaining features.
2. If there is 1 sample in the training dataset OR if all labels are the same, generate a leaf node with the remaining label as the decision and skip steps 3, 4, and 5.
3. Otherwise, find the best split (by finding the threshold and feature pair with the best information gain) using entropy.
4. Generate a decision node with the threshold and feature used to generate the best split.
5. For each side of the split, start from step 2 using the split as the training dataset.

Steps to Fit a Decision Tree with **min_samples_split**

1. If there is 1 sample in the training dataset OR if all labels are the same, generate a leaf node with the remaining label as the decision and skip steps 2, 3, 4 and 5.
2. Otherwise, if the number of samples in the training dataset is less than min_samples_split, generate a leaf node using the most popular of the remaining labels as the decision.
3. Otherwise, find the best split (by finding the threshold and feature pair with the best information gain) using entropy
4. Generate a decision node with the threshold and feature used to generate the best split.
5. For each side of the split, start from step 1 using the split as the training dataset.

Steps to Fit a Decision Tree with **min_samples_leaf**

1. If there is 1 sample in the training dataset OR if all labels are the same, generate a leaf node with the remaining label as the decision and skip steps 2, 3, 4, and 5.
2. Otherwise, find the best split (by finding the threshold and feature pair with the best information gain).
3. Generate a decision node with the threshold and feature used to generate the best split while also meeting the following condition.
    a. When assessing all possible splits, disregard any split that holds less than min_samples_leaf number of splits on either side of the split.
4. If the number of samples in either the left or right split is less than min_samples_leaf, generate a leaf node using the most popular of the remaining labels as the decision.
5. Otherwise, for each side of the split, start from step 1 using the split as the training dataset.

Data Pre-processing

Reusability was a key focus behind the design of this decision tree classifier. As such, it is required that all input feature values are preprocessed (if necessary) to be in numerical form. However, the labels can be any data type. When passing in a completed sample for training, the features and labels must be passed in separately. These values must be contained in a list or array. An example of a completed training sample is shown below.

Features

```
[
  [5.1, 3.5, 1.4, 0.2],
  [4.9, 3.0, 1.4, 0.2],
  [4.7, 3.2, 1.3, 0.2],
  ...,
]
```

Index	*0*	*1*	*2*	*3*
Sample 1 Values	5.1	3.5	1.4	0.2
Sample 2 Values	4.9	3.0	1.4	0.2
Sample 3 Values	4.7	3.2	1.3	0.2
...	**...**	**...**	**...**	**...**

Labels

```
[
  ['versicolor'],
  ['versicolor'],
  ['virginica'],
  ...,
]
```

Index	*0*
Sample 1 Value	'versicolor'
Sample 2 Value	'versicolor'
Sample 3 Value	'virginica'
...	**...**

Code Implementation

Node class

The first step to developing a reusable DT classifier is establishing the properties of the nodes. To do this a node class is created to handle both decision and leaf nodes.

Decision Node

When generating a decision node, the following properties must be passed as parameters.

- feature_index: The index of the feature that will be evaluated at this node.
- threshold: The condition that will be tested against the feature value.
- left_subtree: The chain of nodes on the left side of this node.
- right_subtree: The chain of nodes on the right side of this node.
- info_gain: The information gain at this decision node.
- depth: The depth of this node in the DT.

Leaf Node

When generating a leaf node, the following properties must be passed as parameters.

- leaf_decision: The classification decision of this leaf node.
- depth: The depth of this node in the DT.

The main feature of this Node class is that all Nodes will store all Nodes beneath them in the tree. This will be useful for traversing through the tree when making predictions. Note, this means that the root node (first node) will store the entire DT.

```
class Node():
  def __init__(self, feature_index=None, threshold=None, left_subtree=None,
               right_subtree=None, info_gain=None, leaf_decision=None, depth=None):
    self.feature_index = feature_index
    self.threshold = threshold
    self.left_subtree = left_subtree
    self.right_subtree = right_subtree
    self.info_gain = info_gain
    self.depth = depth

    self.value = leaf_decision  # for lead nodes

    if leaf_decision == None:
      self.node_type = 'decision'
    else:
      self.node_type = 'leaf'
```

Decision Tree class

__init__

```
class DecisionTree():
  def __init__(self, min_samples_split=2, min_samples_leaf=1, max_features=None, info_gain_method='entropy', max_depth=float('inf')):

    self.min_samples_split = min_samples_split # the minium number of samples required to make a split | default = 2
    self.min_samples_leaf = min_samples_leaf # the minimum number of samples required to be at a leaf node | default = 1
    self.max_depth = max_depth # default = infinity
    self.max_features = max_features # default = None | when None all features will be used
    self.info_gain_method = info_gain_method # default = 'entropy'

    self.root_node = None
    self.highest_depth = 0
```

```
IrisClassifier = DecisionTree(min_samples_leaf=5, max_features=4, max_depth=5, min_samples_split=2, info_gain_method='entropy')
```

When initialising a decision tree class hyperparameters can be tuned. If a hyperparameter is not given a value, default values will be given.

The initialiser also has a root_node variable for storing the root node of the DT once it has been fitted. When initialising the DT, it has not yet been fitted and so it has no root node, therefore root_node is assigned a None value when initialised.

The highest_depth is a value which is updated during the fitting process, every time a new depth is reached by a node. Again, when initialising the DT has not been fitted, so the highest_depth is given a value of 0 as there will always be at least one node in a DT, therefore there will always be a depth of at least 0.

The decision to not include the fitting process in the initialiser was made so that the user of this class could easily re-fit using different training samples without the need for inputting the hyperparameters again.

fit

```
122   def fit(self, features, labels):
123     if self.max_features != None:
124       features = self.adjustToMaxFeatures(features)
125     dataset = np.concatenate((features, labels), axis=1)
126     self.root_node = self.buildTree(dataset)
2 IrisClassifier.fit(X_train, Y_train)
```

Before the fitting begins, this function first checks if a max_features value has been given. If no setting has been given the max_features will hold its default value of None and the fitting will commence using all features. Otherwise, if a max_features setting has been given the features will be adjusted to exclude the features that will not be used in the fitting process.

The features and labels will then be concatenated to create a completed dataset. The fitting process is then commenced and the root node of the DT is returned and set as the value of root_node.

adjustToMaxFeatures

```
129  def adjustToMaxFeatures(self, features):
130    num_samples, num_features = np.shape(features)
131    selected_features = np.array([])
132
133    if num_features > self.max_features:
134      if isinstance(self.max_features, str) and self.max_features == 'auto' or self.max_features == 'sqrt':
135        # sqrt
136        self.max_features = np.sqrt(num_features)
137      elif isinstance(self.max_features, str) and self.max_features == 'log2':
138        # log2
139        self.max_features = np.log2(num_features)
140      elif isinstance(self.max_features, float):
141        # fraction
142        self.max_features = self.max_features * num_features
143      # chose 'max_features' number of random features
144      while len(selected_features) < self.max_features:
145        random_index = np.random.randint(num_features)
146        if random_index not in selected_features:
147          selected_features = np.append(selected_features, random_index)
148      # return the features at the selected feature indexes
149      features = features[:, selected_features.astype(int)]
150    return features
```

Before adjusting the features, first the number of features in the dataset is checked against the max_features value. If the number of features in the dataset is less than or equal to the max_features, there is no need to adjust the features.

If there are more features in the dataset than the max_features value, then we check for the type of input that was given and calculate the maximum number of features. For example if the max_features is ‘sqrt’ and we have 4 features in the dataset, $\sqrt{4} = 2$ features will be used.

If the max_features is not a string or a float, it is assumed that the value is an integer, in which case the number of features to be used is directly specified as the integer value.

Once we have the number of features that will be used as our max_features value, we choose max_features number of features at random to be used. These random features are stored in a selected_features array. The selected features are then returned.

buildTree

```
92  def buildTree(self, dataset, curr_depth=0):
93    if curr_depth > self.highest_depth: self.highest_depth = curr_depth #update highest depth value
94
95    features, labels = dataset[:, :-1], dataset[:,-1] #separate the dataset into features and labels
96    num_samples, num_features = np.shape(features)
97    num_labels = np.unique(labels)
98    best_split = self.getBestSplit(dataset, num_features)
99
100   if num_samples < self.min_samples_split or curr_depth >= self.max_depth or len(best_split['left_split']) < self.min_samples_leaf \
101    or len(best_split['right_split']) < self.min_samples_leaf or num_samples == self.min_samples_leaf or best_split['info_gain'] <= 0:
102     # leaf_node
103     leaf_most_prob = self.calcLeafProb(labels)
104     return Node(leaf_decision = leaf_most_prob, depth=curr_depth)
105   # if all labels in the labels array are the same then we can pass any value from the labels array as the leaf_decision
106   elif len(num_labels) == 1:
107     # leaf_node
108     return Node(leaf_decision = labels[0], depth=curr_depth)
109   # otherwise, keep generating decision nodes
110   else:
111     #decision_node
112     #recurse left side
113     left_subtree = self.buildTree(best_split['left_split'], curr_depth+1)
114     #recurse right side
115     right_subtree = self.buildTree(best_split['right_split'], curr_depth+1)
116
117     #return decision Node with it's left and right subtree
118     return Node(best_split['feature_index'], best_split['threshold'],
119                 left_subtree, right_subtree, best_split['info_gain'], depth=curr_depth)
```

This is a recursive method which handles the fitting process. The best split (the split with the most information gain) is found and then a series of checks are done against hyperparameter values and information gain values to decide whether a leaf or decision node should be placed.

Check min_samples_split:

Here we check if the number of samples processed at this node is less than min_samples_split. If this is true, then there are not a sufficient number of samples available to make a split resulting in a decision node, therefore a leaf node will be placed.

Check max_depth:

Here we check if the current depth of this node is greater than or equal to the max_depth. If this is true, no nodes should be generated at a depth below the current depth, therefore a leaf node is placed because a leaf node can have no children. Therefore, the depth of any node cannot exceed max_depth.

Check min_samples_leaf:

Here we check if the potential left and right children of the current node will hold less samples than min_samples_leaf. If either potential child will not hold enough samples, then a leaf node is generated so that the child nodes will never be generated. We also check if the number of samples at the current node is equal to min_samples_leaf. If this case is true any further split will be less than min_samples_leaf and so a leaf node must be generated at the current node position.

Check information gain:

Finally, we check that information gain of the proposed best split for the current node is greater than 0. If this case is true, then no information has been gained from the split and the split is serving no purpose. Therefore in this case, a leaf node will be placed because no further effective progress can be made.

If it is decided that a leaf node should be placed, then we return a Node object with the leaf decision and current depth. The leaf decision is the most popular label of the remaining labels in the dataset.
If the hyperparameter conditions are met and the DT has permission to continue to grow, we must first check that impurities still remain in the dataset. If there are no impurities (all labels are of the same category), continuing to grow the DT would be a waste of resources. In such a case, a leaf node is returned with the current depth and the decision being the single label category remaining in the dataset. This means that the tree will be built top-to-bottom and left-to-right.

If there is no justification for generating a leaf node, then a decision node can be generated. The buildTree function is recursively called and given the left side of the split as a dataset. The current depth is also given as the current depth +1 (being the next depth level from the current depth). The same recursive call is then made but with the right side of the split given as a dataset instead.

Note that if no depth value is given, the default depth value is 0. A depth value should only not be given on the first call of the buildTree function which returns the root node, such as in the fit function. On all recursive calls of builtTree a depth value must be given to accurately track the depth of each node.

At the beginning of each recursive pass of buildTree, we check if the depth of the current node is greater than the highest depth. If this case is true, a new depth has been reached and so the current depth is now the new highest depth.

calcLeafProb

```
84  def calcLeafProb(self, remaining_labels):
85      remaining_labels = list(remaining_labels)
86      return max(remaining_labels, key=remaining_labels.count) # returns the most common label
```

This function takes an ndarray of labels and converts it to a list. This list is then processed to find the most popular label category. The most popular label category is then returned.

getBestSplit

```
51  def getBestSplit(self, dataset, num_features):
52    best_split = {
53      'feature_index': None,
54      'threshold': None,
55      'left_split': [],
56      'right_split': [],
57      'info_gain': None
58    }
59    best_info_gain = -float('inf')
60    # iterate through all feature values to find the threshold with the best info gain
61    for feature_index in range(num_features):
62      feature_values = dataset[:, feature_index]
63      all_possible_thresholds = np.unique(feature_values)
64      #generate a split for each threshold and find the split with the best info_gain
65      for threshold in all_possible_thresholds:
66        left_split, right_split = self.split(dataset, feature_index, threshold)
67        #check both splits have at least 'min_samples_leaf' number of samples
68        if len(left_split) >= self.min_samples_leaf and len(right_split) >= self.min_samples_leaf:
69          all_labels, left_split_labels, right_split_labels = dataset[:, -1], left_split[:, -1], right_split[:, -1]
70          #calculate information gain
71          current_info_gain = self.InfoGain(all_labels, left_split_labels, right_split_labels)
72          #update best split if necessary
73          if current_info_gain > best_info_gain:
74            best_split['feature_index'] = feature_index
75            best_split['threshold'] = threshold
76            best_split['left_split'] = left_split
77            best_split['right_split'] = right_split
78            best_split['info_gain'] = current_info_gain
79            best_info_gain = current_info_gain
80
81    return best_split
```

This function will return the split with the best information gain for a given dataset with num_features number of features. First, a best_split dictionary is defined with empty values and a best_info_gain variable is set to a value of negative infinity (so that it will be less than any other information gain value).

We then assess each sample of each feature in the dataset and establish all value occurrences as possible thresholds. The dataset is split at each possible threshold and feature pair.

We check that the number of samples on each side of the split is greater than or equal to min_samples_leaf. Any split that does not fulfill this requirement cannot be used and so any further processing of such a split would be a waste of resources.

For splits that meet the min_samples_leaf conditions are then assessed for information gain. If the information gain is greater than best_info_gain then a new better information gain has been found. When this happens, the best split dictionary is filled in using the variables that contributed to and resulted in the current best split. The value of best_info_gain is then set to the current information gain before the beginning of the next iteration.

Once all iterations have passed, the best_split dictionary will hold all of the relevant information about the best split that was found. The best_split dictionary is returned as the result.

split

```
42  # This fucntion will split a dataset based on the value of one feature of each sample.
43  def split(self, dataset, feature_index, threshold):
44
45    left_split = np.array([sample for sample in dataset if sample[feature_index] <= threshold])
46    right_split = np.array([sample for sample in dataset if sample[feature_index] > threshold])
47
48    return left_split, right_split
```

To split the dataset, we iterate through each sample in the given dataset, and assess the given threshold against the value of the given feature. If the feature value of a sample is less than or equal to the threshold value, then the samples will be stored in the left_split ndarray. Otherwise, the sample will be stored in the right_split ndarray. The left_split and right_split ndarrays are then returned as the result.

InfoGain

```
31  def InfoGain(self, parent_labels, left_labels, right_labels, ):
32    # weight = no. labels in parent node / no. labels in child split
33    left_weight = len(parent_labels) / len(left_labels)
34    right_weight = len(parent_labels) / len(right_labels)
35
36    if self.info_gain_method == 'gini':
37      info_gain = self.giniIndex(parent_labels) - (left_weight * self.giniIndex(left_labels)
38       + right_weight * self.giniIndex(right_labels))
39    else:
40      info_gain = self.entropy(parent_labels) - (left_weight * self.entropy(left_labels)
41       + right_weight * self.entropy(right_labels))
42    return info_gain
```

First, the weight of the left side of the split is calculated as the number of values in the given left_labels ndarray divided by the number of values in the parent_labels ndarray. The same task is performed for the right side of the split.

We then check what information gain method will be used (entropy or gini index). If gini has been specified it calculates the information gain of the split using Gini Index. Otherwise if info_gain_methods has any other value, it will use entropy by default. The result information gain is then returned.

entropy

```
13    # entropy = sum of (-prob of each feature * log2(prob of each feature))
14  def entropy(self, all_labels):
15    all_possible_labels, label_counts = np.unique(all_labels, return_counts=True)
16    entropy = 0
17    for index in range(all_possible_labels.size):
18      # prob = no. occurances of the current label / no. total labels
19      label_prob = label_counts[index] / len(all_labels)
20      entropy += -label_prob * np.log2(label_prob)
21    return entropy
```

First we extract the all label categories and the number of instances of each label category from the given all_labels ndarray. These values are then stored in all_possible_labels and label_counts respectively.

Initially entropy is equal to 0 as there have been no calculations made yet. We then iterate through each possible label and add the probability of the label in the given all_labels ndarray to the entropy variable using the entropy formula.

After all iterations are complete, the final entropy value is returned as the result.

Note, instead of extracting and using label_counts, this calculation could simply be done using only all_possible_labels. However, the decision to use label_counts was made to provide more information while debugging in future updates to this algorithm.

giniIndex

```
23  def giniIndex(self, all_labels):
24    all_possible_labels, label_counts = np.unique(all_labels, return_counts=True)
25    gini = 0
26    for index in range(all_possible_labels.size):
27        label_prob = label_counts[index] / len(all_labels)
28        gini += label_prob**2
29    return 1 - gini
```

This follows the same process as the entropy function, except the gini formula is used and when all iterations have passed, 1 - the gini value is returned as the result as per the formula dictates.

predict

```
167  def predict(self, dataset):
168    predictions = [self.makePrediction(sample, self.root_node) for sample in dataset]
169    return predictions
```

This function is to be called when wanting to use a fitted DT to make a classification prediction about each sample in a given incomplete dataset (samples have no label values).

makePrediction

```
154   def makePrediction(self, sample, tree):
155     if tree.value != None: #if the traversal has reached a leaf
156       return tree.value
157     else:
158       feature_val = sample[tree.feature_index]
159       if feature_val <= tree.threshold:
160         # recurse left side
161         return self.makePrediction(sample, tree.left_subtree)
162       else:
163         # recurse right side
164         return self.makePrediction(sample, tree.right_subtree)
```

This recursive function is used to traverse down the tree and make a prediction about a single sample using a fitted DT (root node). First, the tree.value variable is checked. If this variable holds a value other than none, then a leaf node has been reached. If this case is true, then the tree.value is returned as the prediction.

Otherwise, the value of the feature being assessed by the current node is extracted from the sample and checked against the threshold of the current node. If the feature value is less than or equal to the threshold of the current node, the algorithm will recurse down to the left child. Otherwise, it will recurse to the right child. This process will end when a leaf node is found.

printTree

```
172   def printTree(self, tree=None, indent=" "):
173     if not tree:
174       tree = self.root_node
175     if tree.value is not None:
176       print(tree.value)
177     else:
178       print("X_"+str(tree.feature_index), "<=", tree.threshold, "?", tree.info_gain)
179       print("%sleft:" % (indent), end="")
180       self.printTree(tree.left_subtree, indent + indent)
181       print("%sright:" % (indent), end="")
182       self.printTree(tree.right_subtree, indent + indent)
```

This function is used to produce an illustrated representation of the DT that has been constructed. It can print from a given node onward by assigning the tree parameter a specific node. Otherwise, it will print the entire tree by default.

Starting from the root node or given node, this algorithm recurses through the tree from top-to-bottom and left-to-right, printing out the values stored in each node.

Evaluation

In order to evaluate the DT, we must take a complete dataset, and split it into two parts; one part for training and the other for testing. The training part will be a complete dataset used for fitting the DT. The testing part will be separated into features and labels. This way we can use the features to yield a prediction from the fitted DT and use the actual label values to assess how accurate the prediction was.

Implementation

Load Data

```
col_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'type']

url = 'https://gist.githubusercontent.com/Nathan13768609/ae5d8dbf657e3932eeea87be71350313/raw/740e26f90e5aa97541d2109cbdd2a3b269c651fc/iris.csv'
data = pd.read_csv(url, skiprows=1, header=None, names=col_names)
data.head(100)
```

	sepal_length	sepal_width	petal_length	petal_width	type
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
...	...	...	...	...	...
96	5.7	3.0	4.2	1.2	versicolor
97	5.7	2.9	4.2	1.3	versicolor
98	6.2	2.9	4.3	1.3	versicolor
99	5.1	2.5	3.0	1.1	versicolor
100	5.7	2.8	4.1	1.3	versicolor

100 rows × 5 columns

For this evaluation I will use the common Iris dataset shown in the image above. Here, I import the dataset from a csv file on my github where I have stored it for easy access. The pd.read_csv function returns the data as a pd.DataFrame.

Train-Test split

```
X = data.iloc[:, :-1].values
Y = data.iloc[:, -1].values.reshape(-1,1)
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.2, random_state=41)
```

Here we split the data into a training dataset and a testing dataset. First we separate the features and the labels and process them to both be two dimensional ndarrays. Then the train_test_split function is imported from sklearn. The train_test_split function will handle the separation of data. Here we give a fractional value which ensures the size of the test split will be

have (0.2 * total number of samples) number of samples. A random_state value of 41 is given which ensures the data is randomly shuffled 41 times before the splits occur. This ensures the train and test samples have different values almost every time. This can be useful to assess if the DT has been overfitted to any one training sample.

Fit the model

```
IrisClassifier = DecisionTree()
IrisClassifier.fit(X_train, Y_train)
IrisClassifier.printTree()
```

```
X_2 <= 1.9 ? 0.06585013851253163
 left:setosa
 right:X_3 <= 1.5 ? 0.09308152935093505
  left:X_2 <= 4.9 ? 0.17556502585750278
    left:versicolor
    right:virginica
  right:virginica
```

Here we initialise the decision tree class and use the training features and training labels, generated by the test_train_split function, to fit the DT. Then the printTree function is called to produce a visual representation of the DT that has been constructed. We can see that the DT has three decision nodes and four leaf nodes.

Test the model

```
print('info_gain_method =', IrisClassifier.info_gain_method)
print('tree depth =', IrisClassifier.highest_depth)
print('max_depth =', IrisClassifier.max_depth)
print('min_samples_split =', IrisClassifier.min_samples_split)
print('max_features =', IrisClassifier.max_features)
print('min_samples_leaf =', IrisClassifier.min_samples_leaf)

Y_pred = IrisClassifier.predict(X_test)
from sklearn.metrics import accuracy_score
print('accuracy =', accuracy_score(Y_test, Y_pred))
```

```
info_gain_method = entropy
tree depth = 3
max_depth = inf
min_samples_split = 2
max_features = None
min_samples_leaf = 1
accuracy = 0.9
```

Finally, we print all the details about the constructed decision tree. Then we use the DT to produce predictions about the uncompleted samples in the X_test dataset. We then import the accuracy_score from sklearn and pass it the actual values and predicted values. This function

will simply count all of the matching values between the two arrays (as this sum will amount to the total number of correct predictions) and returns the sum as a percentage of the total number of samples.

As we can see by using the default hyperparameters we have achieved an accuracy of 90% with a tree depth of 3.

```
IrisClassifier = DecisionTree(min_samples_leaf=2, max_features=3,
                              max_depth=3, min_samples_split=3, info_gain_method='gini')
IrisClassifier.fit(X_train, Y_train)
IrisClassifier.printTree()
```

```
setosa
```

To test that the hyperparameters do affect the decision tree that is constructed, we can fit the DT using the same dataset except with custom hyperparameters.

```
print('info_gain_method =', IrisClassifier.info_gain_method)
print('tree depth =', IrisClassifier.highest_depth)
print('max_depth =', IrisClassifier.max_depth)
print('min_samples_split =', IrisClassifier.min_samples_split)
print('max_features =', IrisClassifier.max_features)
print('min_samples_leaf =', IrisClassifier.min_samples_leaf)

Y_pred = IrisClassifier.predict(X_test)
from sklearn.metrics import accuracy_score
print('accuracy =', accuracy_score(Y_test, Y_pred))
```

```
info_gain_method = gini
tree depth = 0
max_depth = 3
min_samples_split = 3
max_features = 3
min_samples_leaf = 2
accuracy = 0.3
```

As can be seen, due to the hyperparameter settings, a different DT has been constructed with only one node and an accuracy of 30%.

This DT algorithm has been tested using other datasets. These implementations can be found at the link at the top of page 1.

Conclusion

The C4.5 decision tree machine learning algorithm is an effective tool for making accurate classification predictions on a given dataset of incomplete samples. The algorithm must first be fitted using completed data samples, after which it can make predictions about incomplete data samples. During the fitting process every possible reasonable decision about the data is assessed and the decisions which tell the algorithm the most about the data (yield the most information gain) are used to construct the final decision tree. Decision trees can be effectively evaluated by using completed data to assess a prediction against the actual result. To access the code example shown in this report, click on the link at the top of page 1.