# Chapter 11

# Classification and Regression Trees

Decision trees are a powerful prediction method and extremely popular. They are popular because the final model is so easy to understand by practitioners and domain experts alike. The final decision tree can explain exactly why a specific prediction was made, making it very attractive for operational use.

Decision trees also provide the foundation for more advanced ensemble methods such as bagging, random forests and gradient boosting. In this tutorial, you will discover how to implement the Classification And Regression Tree algorithm from scratch with Python. After completing this tutorial, you will know:

- How to calculate and evaluate candidate split points in a data.
- How to arrange splits into a decision tree structure.
- How to apply the classification and regression tree algorithm to a real problem.

Let's get started.

## 11.1 Descriptions

This section provides a brief introduction to the Classification and Regression Tree algorithm and the Banknote dataset used in this tutorial.

### 11.1.1 Classification and Regression Trees

Classification and Regression Trees or CART for short is an acronym introduced by Leo Breiman to refer to Decision Tree algorithms that can be used for classification or regression predictive modeling problems. We will focus on using CART for classification in this tutorial. The representation of the CART model is a binary tree. This is the same binary tree from algorithms and data structures, nothing too fancy (each node can have zero, one or two child nodes).

A node represents a single input variable (`X`) and a split point on that variable, assuming the variable is numeric. The leaf nodes (also called terminal nodes) of the tree contain an output variable (`y`) which is used to make a prediction. Once created, a tree can be navigated with a new row of data following each branch with the splits until a final prediction is made.

Creating a binary decision tree is actually a process of dividing up the input space. A greedy approach is used called recursive binary splitting. This is a numerical procedure where all the values are lined up and different split points are tried and tested using a cost function. The split with the best cost (lowest cost because we minimize cost) is selected. All input variables and all possible split points are evaluated and chosen in a greedy manner based on the cost function.

- **Regression**: The cost function that is minimized to choose split points is the sum squared error across all training samples that fall within the rectangle.
- **Classification**: The Gini cost function is used which provides an indication of how pure the nodes are, where node purity refers to how mixed the training data assigned to each node is.

Splitting continues until nodes contain a minimum number of training examples or a maximum tree depth is reached.

### 11.1.2 Banknote Dataset

In this tutorial we will use the Banknote Dataset. This dataset involves the discrimination between authentic and inauthentic banknotes. The baseline performance on the problem is approximately 50%. You can learn more about it in Appendix A, Section A.6. Download the dataset and save it into your current working directory with the filename `data_banknote_authentication.csv`.

## 11.2 Tutorial

This tutorial is broken down into 5 parts:

1. Gini Index.
2. Create Split.
3. Build a Tree.
4. Make a Prediction.
5. Banknote Case Study.

These steps will give you the foundation that you need to implement the CART algorithm from scratch and apply it to your own predictive modeling problems.

### 11.2.1 Gini Index

The Gini index is the name of the cost function used to evaluate splits in the dataset. A split in the dataset involves one input attribute and one value for that attribute. It can be used to divide training patterns into two groups of rows.

A Gini score gives an idea of how good a split is by how mixed the classes are in the two groups created by the split. A perfect separation results in a Gini score of 0, whereas the worst

case split that results in 50/50 classes in each group results in a Gini score of 0.5 (for a 2 class problem).

Calculating Gini is best demonstrated with an example. We have two groups of data with 2 rows in each group. The rows in the first group all belong to class 0 and the rows in the second group belong to class 1, so it's a perfect split. We first need to calculate the proportion of classes in each group.

$$proportion = \frac{count(class\_value)}{count(rows)} \tag{11.1}$$

The proportions for this example would be:

$$\begin{aligned} \text{group\_1\_class\_0} &= \frac{2}{2} = 1 \\ \text{group\_1\_class\_1} &= \frac{0}{2} = 0 \\ \text{group\_2\_class\_0} &= \frac{0}{2} = 0 \\ \text{group\_2\_class\_1} &= \frac{2}{2} = 1 \end{aligned} \tag{11.2}$$

Gini is then calculated for each child node as follows:

$$\begin{aligned} \text{gini\_index} &= \sum_{i=1}^{n}(proportion_i \times (1.0 - proportion_i)) \\ &= 1 - \sum_{i=1}^{n} proportion_i^2 \end{aligned} \tag{11.3}$$

The Gini index for each group must then be weighted by the size of the group, relative to all of the samples in the parent, e.g. all samples that are currently being grouped. We can add this weighting to the Gini calculation for a group as follows:

$$\text{gini\_index} = (1 - \sum_{i=1}^{n} proportion_i^2) \times \frac{group\_size}{total\_samples} \tag{11.4}$$

In this example the Gini scores for each group are calculated as follows:

$$\begin{aligned} Gini(\text{group\_1}) &= (1 - (1 \times 1 + 0 \times 0)) \times \frac{2}{4} \\ Gini(\text{group\_1}) &= 0.0 \times 0.5 \\ Gini(\text{group\_1}) &= 0.0 \\ Gini(\text{group\_2}) &= (1 - (0 \times 0 + 1 \times 1)) \times \frac{2}{4} \\ Gini(\text{group\_2}) &= 0.0 \times 0.5 \\ Gini(\text{group\_2}) &= 0.0 \end{aligned} \tag{11.5}$$

The scores are then added across each child node at the split point to give a final Gini score for the split point that can be compared to other candidate split points. The Gini for this split point would then be calculated as $0.0 + 0.0$ or a perfect Gini score of 0.0.

Below is a function named `gini_index()` that calculates the Gini index for a list of groups and a list of known class values. You can see that there are some safety checks in there to avoid a divide by zero for an empty group.

```
# Calculate the Gini index for a split dataset
def gini_index(groups, classes):
  # count all samples at split point
  n_instances = float(sum([len(group) for group in groups]))
  # sum weighted Gini index for each group
  gini = 0.0
  for group in groups:
    size = float(len(group))
    # avoid divide by zero
    if size == 0:
      continue
    score = 0.0
    # score the group based on the score for each class
    for class_val in classes:
      p = [row[-1] for row in group].count(class_val) / size
      score += p * p
    # weight the group score by its relative size
    gini += (1.0 - score) * (size / n_instances)
  return gini
```

Listing 11.1: Function To Calculate the Gini Index of a Dataset split.

We can test this function with our worked example above. We can also test it for the worst case of a 50/50 split in each group. The complete example is listed below.

```
# Example of calculating Gini index

# Calculate the Gini index for a split dataset
def gini_index(groups, classes):
  # count all samples at split point
  n_instances = float(sum([len(group) for group in groups]))
  # sum weighted Gini index for each group
  gini = 0.0
  for group in groups:
    size = float(len(group))
    # avoid divide by zero
    if size == 0:
      continue
    score = 0.0
    # score the group based on the score for each class
    for class_val in classes:
      p = [row[-1] for row in group].count(class_val) / size
      score += p * p
    # weight the group score by its relative size
    gini += (1.0 - score) * (size / n_instances)
  return gini

# test Gini values
print(gini_index([[[1, 1], [1, 0]], [[1, 1], [1, 0]]], [0, 1]))
print(gini_index([[[1, 0], [1, 0]], [[1, 1], [1, 1]]], [0, 1]))
```

Listing 11.2: Example of Calculating Gini Index on a Contrived Dataset.

Running the example prints the two Gini scores, first the score for the worst case at 0.5 followed by the score for the best case at 0.0.

```
0.5
0.0
```

Listing 11.3: Example Output of Calculating Gini Index.

Now that we know how to evaluate the results of a split, let's look at creating splits.

### 11.2.2 Create Split

A split is comprised of an attribute in the dataset and a value. We can summarize this as the index of an attribute to split and the value by which to split rows on that attribute. This is just a useful shorthand for indexing into rows of data. Creating a split involves three parts, the first we have already looked at which is calculating the Gini score. The remaining two parts are:

1. Splitting a Dataset.
2. Evaluating All Splits.

Let's take a look at each.

#### Splitting a Dataset

Splitting a dataset means separating a dataset into two lists of rows given the index of an attribute and a split value for that attribute. Once we have the two groups, we can then use our Gini score above to evaluate the cost of the split. Splitting a dataset involves iterating over each row, checking if the attribute value is below or above the split value and assigning it to the left or right group respectively. Below is a function named `test_split()` that implements this procedure.

```
# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
  left, right = list(), list()
  for row in dataset:
    if row[index] < value:
      left.append(row)
    else:
      right.append(row)
  return left, right
```

Listing 11.4: Function To Split a Dataset Based on a Split Point.

Not much to it. Note that the right group contains all rows with a value at the index above or equal to the split value.

#### Evaluating All Splits

With the Gini function above and the test split function we now have everything we need to evaluate splits. Given a dataset, we must check every value on each attribute as a candidate split, evaluate the cost of the split and find the best possible split we could make. Once the best split is found, we can use it as a node in our decision tree.

This is an exhaustive and greedy algorithm. We will use a dictionary to represent a node in the decision tree as we can store data by name. When selecting the best split and using it as a new node for the tree we will store the index of the chosen attribute, the value of that attribute by which to split and the two groups of data split by the chosen split point.

Each group of data is its own small dataset of just those rows assigned to the left or right group by the splitting process. You can imagine how we might split each group again, recursively as we build out our decision tree. Below is a function named `get_split()` that implements this procedure. You can see that it iterates over each attribute (except the class value) and then each value for that attribute, splitting and evaluating splits as it goes. The best split is recorded and then returned after all checks are complete.

```
# Select the best split point for a dataset
def get_split(dataset):
  class_values = list(set(row[-1] for row in dataset))
  b_index, b_value, b_score, b_groups = 999, 999, 999, None
  for index in range(len(dataset[0])-1):
    for row in dataset:
      groups = test_split(index, row[index], dataset)
      gini = gini_index(groups, class_values)
      if gini < b_score:
        b_index, b_value, b_score, b_groups = index, row[index], gini, groups
  return {'index':b_index, 'value':b_value, 'groups':b_groups}
```

Listing 11.5: Function To Find the Best Split Point in a Dataset.

We can contrive a small dataset to test out this function and our whole dataset splitting process.

```
X1          X2            Y
2.771244718  1.784783929   0
1.728571309  1.169761413   0
3.678319846  2.81281357    0
3.961043357  2.61995032    0
2.999208922  2.209014212   0
7.497545867  3.162953546   1
9.00220326   3.339047188   1
7.444542326  0.476683375   1
10.12493903  3.234550982   1
6.642287351  3.319983761   1
```

Listing 11.6: Small Contrived Dataset For Testing CART.

We can plot this dataset using separate colors for each class. You can see that it would not be difficult to manually pick a value of `X1` (x-axis on the plot) to split this dataset.

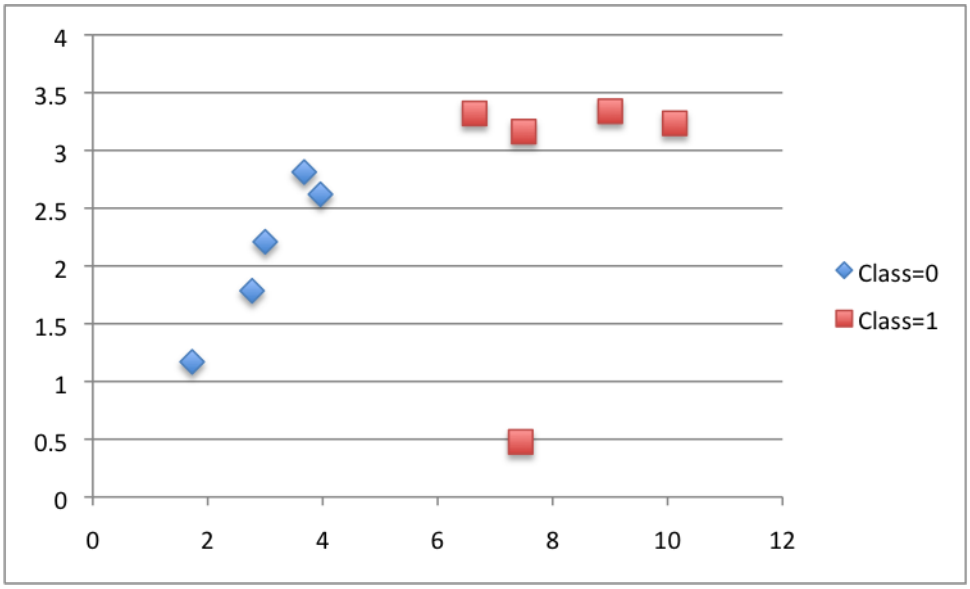


Figure 11.1: Plot of Small Contrived Dataset for Testing CART.

The example below puts all of this together.

```
# Example of getting the best split

# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
  left, right = list(), list()
  for row in dataset:
    if row[index] < value:
      left.append(row)
    else:
      right.append(row)
  return left, right

# Calculate the Gini index for a split dataset
def gini_index(groups, classes):
  # count all samples at split point
  n_instances = float(sum([len(group) for group in groups]))
  # sum weighted Gini index for each group
  gini = 0.0
  for group in groups:
    size = float(len(group))
    # avoid divide by zero
    if size == 0:
      continue
    score = 0.0
    # score the group based on the score for each class
    for class_val in classes:
      p = [row[-1] for row in group].count(class_val) / size
      score += p * p
    # weight the group score by its relative size
    gini += (1.0 - score) * (size / n_instances)
  return gini

# Select the best split point for a dataset
```

```
def get_split(dataset):
  class_values = list(set(row[-1] for row in dataset))
  b_index, b_value, b_score, b_groups = 999, 999, 999, None
  for index in range(len(dataset[0])-1):
    for row in dataset:
      groups = test_split(index, row[index], dataset)
      gini = gini_index(groups, class_values)
      print('X%d < %.3f Gini=%.3f' % ((index+1), row[index], gini))
      if gini < b_score:
        b_index, b_value, b_score, b_groups = index, row[index], gini, groups
  return {'index':b_index, 'value':b_value, 'groups':b_groups}

# Test getting the best split
dataset = [[2.771244718,1.784783929,0],
  [1.728571309,1.169761413,0],
  [3.678319846,2.81281357,0],
  [3.961043357,2.61995032,0],
  [2.999208922,2.209014212,0],
  [7.497545867,3.162953546,1],
  [9.00220326,3.339047188,1],
  [7.444542326,0.476683375,1],
  [10.12493903,3.234550982,1],
  [6.642287351,3.319983761,1]]
split = get_split(dataset)
print('Split: [X%d < %.3f]' % ((split['index']+1), split['value']))
```

Listing 11.7: Example of Calculating The Best Split on a Contrived Dataset.

The `get_split()` function was modified to print out each split point and it's Gini index as it was evaluated. Running the example prints all of the Gini scores and then prints the score of best split in the dataset of $X1 < 6.642$ with a Gini Index of 0.0 or a perfect split.

```
X1 < 2.771 Gini=0.444
X1 < 1.729 Gini=0.500
X1 < 3.678 Gini=0.286
X1 < 3.961 Gini=0.167
X1 < 2.999 Gini=0.375
X1 < 7.498 Gini=0.286
X1 < 9.002 Gini=0.375
X1 < 7.445 Gini=0.167
X1 < 10.125 Gini=0.444
X1 < 6.642 Gini=0.000
X2 < 1.785 Gini=0.500
X2 < 1.170 Gini=0.444
X2 < 2.813 Gini=0.320
X2 < 2.620 Gini=0.417
X2 < 2.209 Gini=0.476
X2 < 3.163 Gini=0.167
X2 < 3.339 Gini=0.444
X2 < 0.477 Gini=0.500
X2 < 3.235 Gini=0.286
X2 < 3.320 Gini=0.375
Split: [X1 < 6.642]
```

Listing 11.8: Example Output of Fining the Best Split.

Now that we know how to find the best split points in a dataset or list of rows, let's see how we can use it to build out a decision tree.

### 11.2.3 Build a Tree

Creating the root node of the tree is easy. We call the above `get_split()` function using the entire dataset. Adding more nodes to our tree is more interesting. Building a tree may be divided into 3 main parts:

1. Terminal Nodes.
2. Recursive Splitting.
3. Building a Tree.

#### Terminal Nodes

We need to decide when to stop growing a tree. We can do that using the depth and the number of rows that the node is responsible for in the training dataset.

- **Maximum Tree Depth**. This is the maximum number of nodes from the root node of the tree. Once a maximum depth of the tree is met, we must stop adding new nodes. Deeper trees are more complex and are more likely to overfit the training data.
- **Minimum Node Records**. This is the minimum number of training patterns that a given node is responsible for. Once at or below this minimum, we must stop splitting and adding new nodes. Nodes that account for too few training patterns are expected to be too specific and are likely to overfit the training data.

These two approaches will be user-specified arguments to our tree building procedure. There is one more condition; it is possible to choose a split in which all rows belong to one group. In this case, we will be unable to continue splitting and adding child nodes as we will have no records to split on one side or another.

Now we have some ideas of when to stop growing the tree. When we do stop growing at a given point, that node is called a terminal node and is used to make a final prediction. This is done by taking the group of rows assigned to that node and selecting the most common class value in the group. This will be used to make predictions. Below is a function named `to_terminal()` that will select a class value for a group of rows. It returns the most common output value in a list of rows.

```
# Create a terminal node value
def to_terminal(group):
  outcomes = [row[-1] for row in group]
  return max(set(outcomes), key=outcomes.count)
```

Listing 11.9: Function To Create a Terminal Node.

### Recursive Splitting

We know how and when to create terminal nodes; now we can build our tree. Building a decision tree involves calling the above developed `get_split()` function over and over again on the groups created for each node. New nodes added to an existing node are called child nodes. A node may have zero children (a terminal node), one child (one side makes a prediction directly) or two child nodes. We will refer to the child nodes as left and right in the dictionary representation of a given node.

Once a node is created, we can create child nodes recursively on each group of data from the split by calling the same function again. Below is a function that implements this recursive procedure. It takes a node as an argument as well as the maximum depth, minimum number of patterns in a node and the current depth of a node. You can imagine how this might be first called passing in the root node and the depth of 1. This function is best explained in steps:

1. Firstly, the two groups of data split by the node are extracted for use and deleted from the node. As we work on these groups the node no longer requires access to these data.
2. Next, we check if either left or right group of rows is empty and if so we create a terminal node using what records we do have.
3. We then check if we have reached our maximum depth and if so we create a terminal node.
4. We then process the left child, creating a terminal node if the group of rows is too small, otherwise creating and adding the left node in a depth first fashion until the bottom of the tree is reached on this branch.
5. The right side is then processed in the same manner, as we rise back up the constructed tree to the root.

```
# Create child splits for a node or make terminal
def split(node, max_depth, min_size, depth):
  left, right = node['groups']
  del(node['groups'])
  # check for a no split
  if not left or not right:
    node['left'] = node['right'] = to_terminal(left + right)
    return
  # check for max depth
  if depth >= max_depth:
    node['left'], node['right'] = to_terminal(left), to_terminal(right)
    return
  # process left child
  if len(left) <= min_size:
    node['left'] = to_terminal(left)
  else:
    node['left'] = get_split(left)
    split(node['left'], max_depth, min_size, depth+1)
  # process right child
  if len(right) <= min_size:
    node['right'] = to_terminal(right)
  else:
    node['right'] = get_split(right)
```

```
    split(node['right'], max_depth, min_size, depth+1)
```

Listing 11.10: Function To Create Split Points Recursively.

### Building a Tree

We can now put all of the pieces together. Building the tree involves creating the root node and calling the `split()` function that then calls itself recursively to build out the whole tree. Below is the small `build_tree()` function that implements this procedure.

```
# Build a decision tree
def build_tree(train, max_depth, min_size):
  root = get_split(train)
  split(root, max_depth, min_size, 1)
  return root
```

Listing 11.11: Function To Create a Decision Tree.

We can test out this whole procedure using the small dataset we contrived above. Below is the complete example. Also included is a small `print_tree()` function that recursively prints out nodes of the decision tree with one line per node. Although not as striking as a real decision tree diagram, it gives an idea of the tree structure and decisions made throughout.

```
# Example of building a tree

# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
  left, right = list(), list()
  for row in dataset:
    if row[index] < value:
      left.append(row)
    else:
      right.append(row)
  return left, right

# Calculate the Gini index for a split dataset
def gini_index(groups, classes):
  # count all samples at split point
  n_instances = float(sum([len(group) for group in groups]))
  # sum weighted Gini index for each group
  gini = 0.0
  for group in groups:
    size = float(len(group))
    # avoid divide by zero
    if size == 0:
      continue
    score = 0.0
    # score the group based on the score for each class
    for class_val in classes:
      p = [row[-1] for row in group].count(class_val) / size
      score += p * p
    # weight the group score by its relative size
    gini += (1.0 - score) * (size / n_instances)
  return gini
```

```
# Select the best split point for a dataset
def get_split(dataset):
  class_values = list(set(row[-1] for row in dataset))
  b_index, b_value, b_score, b_groups = 999, 999, 999, None
  for index in range(len(dataset[0])-1):
    for row in dataset:
      groups = test_split(index, row[index], dataset)
      gini = gini_index(groups, class_values)
      if gini < b_score:
        b_index, b_value, b_score, b_groups = index, row[index], gini, groups
  return {'index':b_index, 'value':b_value, 'groups':b_groups}

# Create a terminal node value
def to_terminal(group):
  outcomes = [row[-1] for row in group]
  return max(set(outcomes), key=outcomes.count)

# Create child splits for a node or make terminal
def split(node, max_depth, min_size, depth):
  left, right = node['groups']
  del(node['groups'])
  # check for a no split
  if not left or not right:
    node['left'] = node['right'] = to_terminal(left + right)
    return
  # check for max depth
  if depth >= max_depth:
    node['left'], node['right'] = to_terminal(left), to_terminal(right)
    return
  # process left child
  if len(left) <= min_size:
    node['left'] = to_terminal(left)
  else:
    node['left'] = get_split(left)
    split(node['left'], max_depth, min_size, depth+1)
  # process right child
  if len(right) <= min_size:
    node['right'] = to_terminal(right)
  else:
    node['right'] = get_split(right)
    split(node['right'], max_depth, min_size, depth+1)

# Build a decision tree
def build_tree(train, max_depth, min_size):
  root = get_split(train)
  split(root, max_depth, min_size, 1)
  return root

# Print a decision tree
def print_tree(node, depth=0):
  if isinstance(node, dict):
    print('%s[X%d < %.3f]' % ((depth*' ', (node['index']+1), node['value'])))
    print_tree(node['left'], depth+1)
    print_tree(node['right'], depth+1)
  else:
    print('%s[%s]' % ((depth*' ', node)))
```

```
dataset = [[2.771244718,1.784783929,0],
  [1.728571309,1.169761413,0],
  [3.678319846,2.81281357,0],
  [3.961043357,2.61995032,0],
  [2.999208922,2.209014212,0],
  [7.497545867,3.162953546,1],
  [9.00220326,3.339047188,1],
  [7.444542326,0.476683375,1],
  [10.12493903,3.234550982,1],
  [6.642287351,3.319983761,1]]
tree = build_tree(dataset, 1, 1)
print_tree(tree)
```

Listing 11.12: Example of Creating a Decision Tree From the Contrived Dataset.

We can vary the maximum depth argument as we run this example and see the effect on the printed tree. With a maximum depth of 1 (the second parameter in the call to the `build_tree()` function), we can see that the tree uses the perfect split we discovered in the previous section. This is a tree with one node, also called a decision stump.

```
[X1 < 6.642]
 [0]
 [1]
```

Listing 11.13: Example of a Decision Tree With Depth=1 (Decision Stump).

Increasing the maximum depth to 2, we are forcing the tree to make splits even when none are required. The `X1` attribute is then used again by both the left and right children of the root node to split up the already perfect mix of classes.

```
[X1 < 6.642]
 [X1 < 2.771]
  [0]
  [0]
 [X1 < 7.498]
  [1]
  [1]
```

Listing 11.14: Example of a Decision Tree With Depth=2.

Finally, and perversely, we can force one more level of splits with a maximum depth of 3.

```
[X1 < 6.642]
 [X1 < 2.771]
  [0]
  [X1 < 2.771]
   [0]
   [0]
 [X1 < 7.498]
  [X1 < 7.445]
   [1]
   [1]
  [X1 < 7.498]
   [1]
   [1]
```

Listing 11.15: Example of a Decision Tree With Depth=3.

These tests show that there is great opportunity to refine the implementation to avoid unnecessary splits. This is left as an extension. Now that we can create a decision tree, let's see how we can use it to make predictions on new data.

### 11.2.4 Make a Prediction

Making predictions with a decision tree involves navigating the tree with the specifically provided row of data. Again, we can implement this using a recursive function, where the same prediction routine is called again with the left or the right child nodes, depending on how the split affects the provided data. We must check if a child node is either a terminal value to be returned as the prediction, or if it is a dictionary node containing another level of the tree to be considered.

Below is the `predict()` function that implements this procedure. You can see how the index and value in a given node is used to evaluate whether the row of provided data falls on the left or the right of the split.

```
# Make a prediction with a decision tree
def predict(node, row):
  if row[node['index']] < node['value']:
    if isinstance(node['left'], dict):
      return predict(node['left'], row)
    else:
      return node['left']
  else:
    if isinstance(node['right'], dict):
      return predict(node['right'], row)
    else:
      return node['right']
```

Listing 11.16: Function To Make a Prediction With a Decision Tree.

We can use our contrived dataset to test this function. Below is an example that uses a hard-coded decision tree with a single node that best splits the data (a decision stump). The example makes a prediction for each row in the dataset.

```
# Example of making predictions

# Make a prediction with a decision tree
def predict(node, row):
  if row[node['index']] < node['value']:
    if isinstance(node['left'], dict):
      return predict(node['left'], row)
    else:
      return node['left']
  else:
    if isinstance(node['right'], dict):
      return predict(node['right'], row)
    else:
      return node['right']

# contrived dataset
dataset = [[2.771244718,1.784783929,0],
  [1.728571309,1.169761413,0],
  [3.678319846,2.81281357,0],
  [3.961043357,2.61995032,0],
```

```
  [2.999208922,2.209014212,0],
  [7.497545867,3.162953546,1],
  [9.00220326,3.339047188,1],
  [7.444542326,0.476683375,1],
  [10.12493903,3.234550982,1],
  [6.642287351,3.319983761,1]]
#  predict with a stump
stump = {'index': 0, 'right': 1, 'value': 6.642287351, 'left': 0}
for row in dataset:
  prediction = predict(stump, row)
  print('Expected=%d, Got=%d' % (row[-1], prediction))
```

Listing 11.17: Example of Making Predictions on the Contrived Dataset.

Running the example prints the correct prediction for each row, as expected.

```
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
```

Listing 11.18: Example Output of Making a Prediction with a Decision Tree.

We now know how to create a decision tree and use it to make predictions. Now, let's apply it to a real dataset.

### 11.2.5 Banknote Case Study

This section applies the CART algorithm to the Bank Note dataset. The first step is to load the dataset and convert the loaded data to numbers that we can use to calculate split points. For this we will use the helper function `load_csv()` to load the file and `str_column_to_float()` to convert string numbers to floats.

We will evaluate the algorithm using $k$-fold cross-validation with 5 folds. This means that $\frac{1372}{5} = 274.4$ or just over 270 records will be used in each fold. We will use the helper functions `evaluate_algorithm()` to evaluate the algorithm with cross-validation and `accuracy_metric()` to calculate the accuracy of predictions.

A new function named `decision_tree()` was developed to manage the application of the CART algorithm, first creating the tree from the training dataset, then using the tree to make predictions on a test dataset. The complete example is listed below.

```
# Example of CART on the Banknote dataset
from random import seed
from random import randrange
from csv import reader

# Load a CSV file
def load_csv(filename):
  dataset = list()
```

```
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / n_folds)
  for _ in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    accuracy = accuracy_metric(actual, predicted)
    scores.append(accuracy)
  return scores

# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
  left, right = list(), list()
```

```
  for row in dataset:
    if row[index] < value:
      left.append(row)
    else:
      right.append(row)
  return left, right

# Calculate the Gini index for a split dataset
def gini_index(groups, classes):
  # count all samples at split point
  n_instances = float(sum([len(group) for group in groups]))
  # sum weighted Gini index for each group
  gini = 0.0
  for group in groups:
    size = float(len(group))
    # avoid divide by zero
    if size == 0:
      continue
    score = 0.0
    # score the group based on the score for each class
    for class_val in classes:
      p = [row[-1] for row in group].count(class_val) / size
      score += p * p
    # weight the group score by its relative size
    gini += (1.0 - score) * (size / n_instances)
  return gini

# Select the best split point for a dataset
def get_split(dataset):
  class_values = list(set(row[-1] for row in dataset))
  b_index, b_value, b_score, b_groups = 999, 999, 999, None
  for index in range(len(dataset[0])-1):
    for row in dataset:
      groups = test_split(index, row[index], dataset)
      gini = gini_index(groups, class_values)
      if gini < b_score:
        b_index, b_value, b_score, b_groups = index, row[index], gini, groups
  return {'index':b_index, 'value':b_value, 'groups':b_groups}

# Create a terminal node value
def to_terminal(group):
  outcomes = [row[-1] for row in group]
  return max(set(outcomes), key=outcomes.count)

# Create child splits for a node or make terminal
def split(node, max_depth, min_size, depth):
  left, right = node['groups']
  del(node['groups'])
  # check for a no split
  if not left or not right:
    node['left'] = node['right'] = to_terminal(left + right)
    return
  # check for max depth
  if depth >= max_depth:
    node['left'], node['right'] = to_terminal(left), to_terminal(right)
    return
```

```
  # process left child
  if len(left) <= min_size:
    node['left'] = to_terminal(left)
  else:
    node['left'] = get_split(left)
    split(node['left'], max_depth, min_size, depth+1)
  # process right child
  if len(right) <= min_size:
    node['right'] = to_terminal(right)
  else:
    node['right'] = get_split(right)
    split(node['right'], max_depth, min_size, depth+1)

# Build a decision tree
def build_tree(train, max_depth, min_size):
  root = get_split(train)
  split(root, max_depth, min_size, 1)
  return root

# Make a prediction with a decision tree
def predict(node, row):
  if row[node['index']] < node['value']:
    if isinstance(node['left'], dict):
      return predict(node['left'], row)
    else:
      return node['left']
  else:
    if isinstance(node['right'], dict):
      return predict(node['right'], row)
    else:
      return node['right']

# Classification and Regression Tree Algorithm
def decision_tree(train, test, max_depth, min_size):
  tree = build_tree(train, max_depth, min_size)
  predictions = list()
  for row in test:
    prediction = predict(tree, row)
    predictions.append(prediction)
  return(predictions)

# Test CART on Bank Note dataset
seed(1)
# load and prepare data
filename = 'data_banknote_authentication.csv'
dataset = load_csv(filename)
# convert string attributes to integers
for i in range(len(dataset[0])):
  str_column_to_float(dataset, i)
# evaluate algorithm
n_folds = 5
max_depth = 5
min_size = 10
scores = evaluate_algorithm(dataset, decision_tree, n_folds, max_depth, min_size)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

Listing 11.19: Example of CART on the Banknote Dataset.

The example uses the max tree depth of 5 layers and the minimum number of rows per node to 10. These parameters to CART were chosen with a little experimentation, but by no means are they optimal. Running the example prints the average classification accuracy on each fold as well as the average performance across all folds.

You can see that CART and the chosen configuration achieved a mean classification accuracy of about 97% which is dramatically better than the baseline performance of 50% accuracy.

```
Scores: [96.35036496350365, 97.08029197080292, 97.44525547445255, 98.17518248175182,
    97.44525547445255]
Mean Accuracy: 97.299%
```

Listing 11.20: Example Output for CART on the Banknote Dataset.

## 11.3 Extensions

This section lists extensions to this tutorial that you may wish to explore.

- **Algorithm Tuning**. The application of CART to the Bank Note dataset was not tuned. Experiment with different parameter values and see if you can achieve better performance.
- **Cross Entropy**. Another cost function for evaluating splits is cross entropy (logloss). You could implement and experiment with this alternative cost function.
- **Tree Pruning**. An important technique for reducing overfitting of the training dataset is to prune the trees. Investigate and implement tree pruning methods.
- **Categorical Dataset**. The example was designed for input data with numerical or ordinal input attributes, experiment with categorical input data and splits that may use equality instead of ranking.
- **Regression**. Adapt the tree for regression using a different cost function and method for creating terminal nodes.
- **More Datasets**. Apply the algorithm to more datasets on the UCI Machine Learning Repository.

## 11.4 Review

In this tutorial, you discovered how to implement the decision tree algorithm from scratch with Python. Specifically, you learned:

- How to select and evaluate split points in a training dataset.
- How to recursively build a decision tree from multiple splits.
- How to apply the CART algorithm to a real world classification predictive modeling problem.