# COMPUTING DECISION TREES WITH A SINGLE SORT

Tara Mirmira and Ramesh Subramonian

**Abstract**

A typical computational strategy for building decision trees (as evidenced in scikit-learn) is as follows. Each feature is sorted and then traversed in ascending order to determine the best split point. The best split point over all features is selcted and used to partition the data into two. This process is repeated recursively until some stopping criterion is reached e.g., the number of instances is too small. The contribution of this paper is to provide a novel indexing strategy that requires a single sort at the beginning. After that, maintaining the sorted order is accomplished by a linear scan of the data.

## 1 Introduction

**TO BE COMPLETED**

### 1.1 Motivation

Notwithstanding the growing popularity of neural networks, decision trees and random forests continue to be a widely used machine learning technique. Efficient training of decision trees becomes important when

1. the data set sizes become large
2. there is a large hyper-parameter space over which to tune
3. a random forest may employ a large number of decision trees

This calls for an efficient algorithm and implementation for the kernel of the computation. This can be summarized as

1. scanning all the distinct values of all the features to see how they partition the data in terms of the goal attribute
2. computing the metric of interest (gini impurity, entropy, . . . ) for each of the candidate split points

## 2 Algorithm

### 2.1 Assumptions

Our current implementation makes the following assumptions. These are purely for the sake of convenience — these are not conceptual limitations inherent in the approach.

1. The goal attribute can be encoded as 0 or 1
2. The values of the attributes used to build the decision tree are ordered and can be represented as floating point numbers
3. $n \leq 2^{32}$, where $n$ is the number of instances
4. The number of unique values for each instance is $< 2^{31}$
5. There are no missing values.

### 2.2 Data Structures

1. Let $n$ be the number of instances
2. Let $m$ be the number of features.
3. Let $X[m][n]$ be the input data. $X[j]$, also denoted as $X_j$, is a column vector containg the values of feature $j$
4. Let $g[n]$ be the values of the goal attribute
5. Let $Y[m][n]$ be the transformed data where input data has been "position-encoded" by its position in the sort order (ascending). A sample mapping from X-values to Y-values is shown below.

   $$[11, 32, 47, 11, 17, 28, 32, 55] \Rightarrow [1, 4, 5, 1, 2, 4, 6]$$

   Further, for efficiency, each element of Y is encoded as a 64-bit integer where

   - bits $[0..30]$ represent the Y-value itself. We refer to this as $Y_j[i].y$
   - bit 31 represent the goal value We refer to this as $Y_j[[i].g$
   - bits $[32..63]$ represent the **from** value, explained later We refer to this as $Y_j[i].f$

   $Y_j$ is sorted so that $Y_j[i].y \leq Y_j[i+1].y$ In order to record the original position of this value, we use the **from** field. The inter-relationship is specified as follows:

   - $x = X_j[k]$

- $k = Y_j[i].f$
- $y = Y_j[i].y$
- Then, $y$ is the position-encoded value of $x$

6. Let $T[m][n]$ be a data structure used to record the "to" indexing. Intuitively, it tells us **to** which position a datum in the original set has been permuted. Its interlinking with the **from** field is best explained in Invariant 1

**Invariant 1** *Let* $p = Y_j[i].f$. *Then* $T_j[p] = i$

The algorithm is motivated with a simple example where

1. $n = 16$
2. $m = 2$
3. The $x$-values and their corresponding position-encoded $y$-values are the same
4. Column **P** represents position
5. Column **L** is a label (not used by the algorithm)
6. Column $F_1$ is the values of feature 1
7. Column $F_2$ is the values of feature 2
8. Column **G** is the values of the goal
9. Column $Y_1$ is the sorted, encoded values of $F_1$. Values in the column are a tuple $(f, g, y)$, where
   (a) $f$ — the position of this value in the original data set, F1.
   (b) $g$ — the value of the goal feature for this instance
   (c) $y$ — the position-encoded value for this feature
10. Column $Y_2$ is the sorted, encoded values of $F_2$
11. Column $T_1$ is the **to** data structure for $F_1$
12. Column $T_2$ is the **to** data structure for $F_2$

| **P** | **L** | $F_1$ | $F_2$ | **G** | $Y_1$ | $Y_2$ | $T_1$ | $T_2$ | $Y_2'$ | $T_2'$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | A | 8 | 7 | 1 | (3 , 0 , 1) | (13 , 1 , 1) | 7 | 6 | (13 , 1 , 1) | 2 |
| 1 | B | 3 | 2 | 0 | (15 , 1 , 2) | (1 , 0 , 2) | 2 | 1 | (1 , 0 , 2) | 1 |
| 2 | C | 12 | 11 | 0 | (1 , 0 , 3) | (14 , 0 , 3) | 11 | 10 | (0 , 1 , 7) | 13 |
| 3 | D | 1 | 14 | 0 | (13 , 0 , 4) | (7 , 0 , 4) | 0 | 13 | (8 , 1 , 8) | 6 |
| 4 | E | 10 | 9 | 1 | (5 , 1 , 5) | (11 , 1 , 5) | 9 | 8 | (15 , 1 , 10) | 12 |
| 5 | F | 5 | 16 | 1 | (8 , 1 , 6) | (12 , 1 , 6) | 4 | 15 | (9 , 0 , 12) | 7 |
| 6 | G | 13 | 15 | 1 | (9 , 1 , 7) | (0 , 1 , 7) | 12 | 14 | (3 , 0 , 14) | 15 |
| 7 | H | 11 | 4 | 0 | (0 , 0 , 8) | (8 , 1 , 8) | 10 | 3 | (5 , 1 , 16) | 9 |
| 8 | I | 6 | 8 | 1 | (10 , 1 , 9) | (4 , 1 , 9) | 5 | 7 | (14 , 0 , 3) | 3 |
| 9 | J | 7 | 12 | 0 | (4 , 0 , 10) | (15 , 1 , 10) | 6 | 11 | (7 , 0 , 4) | 5 |
| 10 | K | 9 | 13 | 1 | (7 , 1 , 11) | (2 , 0 , 11) | 8 | 12 | (11 , 1 , 5) | 14 |
| 11 | L | 14 | 5 | 1 | (2 , 1 , 12) | (9 , 0 , 12) | 13 | 4 | (12 , 1 , 6) | 10 |
| 12 | M | 15 | 6 | 1 | (6 , 1 , 13) | (10 , 1 , 13) | 14 | 5 | (4 , 1 , 9) | 11 |
| 13 | N | 4 | 1 | 1 | (11 , 1 , 14) | (3 , 0 , 14) | 3 | 0 | (2 , 0 , 11) | 0 |
| 14 | O | 16 | 3 | 0 | (12 , 0 , 15) | (6 , 1 , 15) | 15 | 2 | (10 , 1 , 13) | 8 |
| 15 | P | 2 | 10 | 1 | (14 , 1 , 16) | (5 , 1 , 16) | 1 | 9 | (6 , 1 , 15) | 4 |

### 2.3 Example

Let us assume that the best split is using feature $F_1$ such that the first $i_L = 8$ 8 elements are assigned to the left sub-tree and the remaining to the rest. In other words, $F_1 \leq 8$ selects the data for the left child, with the balance to the right.

- The instances for the left sub-treee have labels $\{D, P, B, N, F, I, J, A\}$
- The instances for the right sub-treee have labels $\{K, E, H, C, G, L, M, O\}$

When we move to processing the left and right sub-trees, no additional processing needs to be done for feature $F_1$ because it already in the the correct sorted order. However, $F_2$ needs to be re-arranged so that it is in sorted order. This is done as follows

1. traverse the $Y_2$ column in order. Use $p = Y_2[i].f$ to figure out where this item came *from*. Use $q = T_1[p]$ to decide where this item went **to**. If $q \leq i_L = 8$, then we know that it goes to the left sub-tree; else it goes to the right sub -tree. Since the $Y_2$ column was sorted, and we build the left and right trees by scanning $Y_2$ in order, the left and right trees continue to be in sorted order. This allows us to build the $Y_2'$ column.

2. There is one last detail to consider. Which is that the $T_2'$ field needs to be updated to reflect the reordering of the values of $F_2$ in $Y_2'$.

   This is done as follows. Assume $Y_2[i]$ is to be moved to positon $k$. Let $f = Y_2[i].f$. Then, $T_2'[f] \leftarrow k$

Note that we need to keep $Y_2, T_2$ until we have built $Y_2', T_2'$ at which point we can replace them with $Y_2', T_2'$

As a quick sanity check, notice that the values of $F_2$ in left and right sub-trees are sorted as:

**left** $[1, 2, 7, 8, 10, 12, 14, 16]$, see rows 0 to 7 of column $Y_2'$.

**right** $[3, 4, 5, 6, 9, 11, 13, 15]$ , see rows 8 to 15 of column $Y_2'$.

### 2.4 The Algorithm

The key insight that drives the algorithm is that preserving the sorted order of the features *not* chosen for the current split does *not* require a re-sort. Instead, as long as we can identify whether an element should be assigned to the left or right sub-tree, then we can scan the elements in order and add them to the left/right sub-trees. This preserves the sorted order and allows us to maintain the doubly-indexed data structures and sets us up for a recursive solution.

## 3 Conclusion

**TO BE COMPLETED**