Comparing random forest implementations BDT against MDD from BDT

J.J. ten Wolde
Faculty of Electrical Engineering, Mathematics and Computer Science, Computer Architecture for Embedded Systems group, University of Twente, Enschede

***Abstract*—Multi-valued decision diagrams (MDDs) show a promising path to potentially higher evaluating speed of random forest. Only the Treelite library and by extension the TL2Cgen library have no support for these models. They do however support binary decision trees (BDTs). This paper is an exploration of the conversion of BDTs to MDDs. The main advantage of an MDD is that each feature only needs to be evaluated in one node. Treelite however does not support nodes with more than two edges, which would be needed to create the MDD nodes. They were instead emulated using binary nodes. Comparing the MDD and BDT shows that the MDD is equal to the BDT at best in this implementation. The closest results were achieved by having only a few features per tree. Trees with more features resulted in an increase in tree depth and a decrease in prediction per second.**

I. Introduction

TREELITE is a library that aims to standardize the model for decision trees and random forests. This is to achieve an easy exchange between C++ applications. An extension to this library, TL2cgen, is used to convert tree models from Treelite into C. TL2cgen can provide C for binary decision trees using the If-Else implementation. This paper takes a preliminary look at extending the capability of Treelite and TL2cgen to also support multi-valued decision diagrams next to the binary decision trees.

Binary decision trees (BDT) consist of nodes and edges. Each node contains a condition, based on which the left or right edge is followed. The first node, the root node, is the start of the tree. The edges will point to new nodes with again a condition and two edges. This repeats until a leaf node is found which contains the value that is the output of the whole tree. The features are the input for the tree and each condition is based on one of these features.

A node in the BDT can however only apply one condition to a feature, so choose an edge on a binary decision. With a multi-valued decision diagram (MDD) each node can take a more detailed decision based on the feature and choose between more than two edges. The benefit is that a feature only has to be used by one node along the path to a leaf. This ensures that the path length to a leaf is at most equal to the number of features in the tree.

Treelite does however not support multi-valued Decision diagrams, as these diagrams have a different structure compared to binary decision trees. The MDD needs to be created from the binary decision tree and using TL2cgen converted to C so MDDs can be compared to BDTs.

So in this paper we aim to answer the following question. Can multi-valued decision diagrams converted from binary decision trees, provide an improvement in throughput over binary decision trees in a random forest using the Treelite and TL2cgen library?

II. Theoretical Background

A. MDD

Multi-valued decision diagrams (MDD) have the same function as binary decision trees. Binary decision trees (BDT), as the name suggests, can only discern between true or false (1 or 0). Thus having only two edges from each node.

An MDD can discern between a lot more values, so nodes can have more than two edges. This also means that each path (from the root node to the terminal node) in the MDD only passes each variable once, whereas with the BDT each variable could be passed multiple times. The difference between the BDT and MDD is visualized in Figure 1

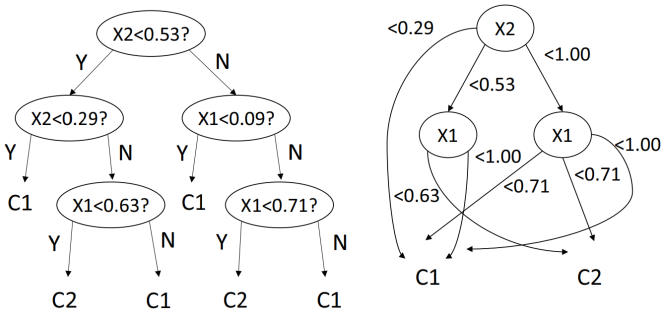

Fig. 1. Visual representation of a BDT (left) and an MDD (right) [1]

With the shorter path length of the MDD compared to the BDT, it has the potential to be evaluated at a higher speed. A disadvantage of the MDD is that as the number of variables (N) increases, the number of nodes increases with $O(2^N)$ [1].

In [1] the MDD implementation and BDT implementation are compared for multiple datasets. The results show that the path length for the BDT is 2.0 to 6.6 times longer. However, in the datasets with a higher feature count, from 19 to 34, the number of nodes in the MDD compared to the BDT is 140.1 to 623.7 times as high. But with a feature count of 10 or less, it has at most 13.7 times as many nodes, and from 5 features and less, the MDD has fewer nodes than the BDT.

B. Tree Implementations

In [2] three BDT implementation methods are evaluated for a CPU.

1) Naïve Implementation: The first implementation is the naïve one. [2] mentions the close relation between decision trees and binary search trees. To perform a prediction using the tree, it starts at index zero of the array (root node) and checks if the current node is a leaf (terminal node), and if so returns a prediction. Otherwise, it compares the feature variable value to the split value, and depending on that moves to the left or right child node. In Figure 2 the full implementation is shown.

```
1:  function PREDICT(X, tree)
2:      i ← load(0)                                          ▷ 1
3:      r1 ← load(tree[0].isLeaf)                            ▷ 1
4:      while cmp_eq_false(r1) do                            ▷ 2
5:          r1 ← load(tree[i].feature)                       ▷ 1
6:          r1 ← load(x[r1])                                 ▷ 1
7:          r2 ← load(tree[i].split)                         ▷ 1
8:          if cmp_le(r1, r2) then                           ▷ 2
9:              i ← load(tree[i].leftChild)                  ▷ 1
10:         else
11:             i ← load(tree[i].rightChild)                 ▷ 1
12:         end if
13:         r1 ← load(tree[i].isLeaf)                        ▷ 1
14:     end while
15:     return tree[i].prediction                            ▷ 1
16: end function
```

Fig. 2. Implementation of a Naïve tree [2]

2) If-Else Implementation: The load instructions in the Naïve implementation are quite time-consuming. The second implementation addresses this by unfolding the tree into an if-else structure. The feature and split value do not have to be loaded for each node and only the comparison needs to be run.

3) SIMD Implementation: This implementation looks like the Naïve implementation, except it is doing the comparison for multiple nodes at the same time by using vectors. This can be seen in Figure 3. For each node it is not known beforehand which branch will be taken, therefore a choice must be made on how to decide which nodes to include in each vector. The two suggested strategies are Depth-first and Breadth-first.

Depth-first includes the paths from the initial node to leaf nodes that have the highest probability of being used. For each case, the resulting performance of this method differs a lot, depending on the variability of the paths being used.

Breadth-first includes all nodes closest to the initial node. Preferably all the nodes are taken from each layer until there is not enough space left in the vector for a whole new layer. The remaining spots are filled up with the most probable nodes from the next layer up.

4) Random Forest: As the trees in the random forest are not dependent on each other they can be processed in parallel. Each tree's prediction is stored as a single bit, making a vector of bits. Calculating the hamming weight of this vector and comparing this to the number of trees divided by two gives the majority vote of all the trees, this is the final prediction of the random forest.

```
1:  function PREDICT(X, tree)
2:      i ← load(0)                                          ▷ 1
3:      r1 ← load(tree[0].isLeaf)                            ▷ 1
4:      while cmp_eq_false(r1) do                            ▷ 2
5:          v1 ← load(tree[i].splits)                        ▷ 1
6:          v2 ← load(tree[i].features)                      ▷ 1
7:          v2 ← gather(x[r2])                               ▷ 1
8:          v2 ← compare(v1, v2)                             ▷ 1
9:          mask ← extract(v2)                               ▷ 1
10:         r1 ← tree[i].nextChildren                        ▷ 1
11:         i ← load(r1[mask])                               ▷ 1
12:         r1 ← load(tree[i].isLeaf)                        ▷ 1
13:     end while
14:     return tree[i].prediction                            ▷ 1
15: end function
```

Fig. 3. Implementation of a SIMD tree [2]

TABLE I
THROUGHPUT, POWER CONSUMPTION AND EFFICIENCY COMPARISON FOR VARIOUS IMPLEMENTATION OF DECISION TREES ON FPGAS [2]

	Throughput $[elem/ms]$	Power $[W]$	Power per Element $[nJ/elem]$
If-Else	29000	1.58	52.76
Naïve	14500	1.58	105.51

5) Result: Resulting of the experiment (Table I) the If-Else tree is the fastest implementation, with the Naïve tree being around 50% slower. The total power consumption is the same for borth implementations, 1.3 MB, but as the If-Else implementation is a lot faster than the other, it uses half of the power per element.

So the If-Else implementation is the best. TL2cgen also uses the If-Else implementation.

III. METHODS

In order to be able to compare the performance of the MDD to the BDT, the models first need to be generated. A couple of datasets are chosen and from the data, the BDT models can be trained. This model is converted to C and used to create the MDD model that also is converted to C. The C-code is compiled, run and measured to compare the two implementations. Each step is explained in more detail below.

A. BDT Generation

The datasets used for the tests can be found in **??** and were retrieved from the UCI Machine Learning Repository except for D9. They were selected based on variety in the feature count, ranging from 4 to 87 features. These datasets are divided into 2 sets, namely a training set of 75% of the original and a testing set of the remaining 25%. The former was used to train the model using the XGBoost library. Nine forests were generated, with each containing about 50 trees. XGBoost requires the number of trees to be a multiple of the number of classes so the amount of trees per dataset ranges

TABLE II
DATASETS

Dataset	#Features	#Instances
Iris (D0)	4	150
Hayes-Roth (D1)	4	160
Statlog Shuttle (D2)	7	58000
Glass Identification (D3)	9	2214
Wine Quality (D4)	11	489
Dry Beans (D5)	16	13611
Automobile (D6)	25	205
Ionosphere (D7)	34	351
Statlog Satellite (D8)	36	6435
Anomaly (D9)	87	262081

from 48 to 51. Each forest has a different max depth value of 2 up to 10. This controls how many nodes can exist along a path from the root to a leaf. The forest is finally converted to Treelite models using the Treelite library.

B. MDD Construction

As an MDD will process each feature one by one, the first step is to extract all features from the BDT and establish a new order for the MDD. This can be done based on a couple of characteristics, for example, based on the feature index or the node count of each feature. In this paper, the feature order was from the most to the least occurring feature, but the order will be established for each tree separately.

A node is created for the first feature according to our feature order, then an edge is created for each threshold that corresponds to this feature in the BDT, including an edge for the values higher than the highest threshold. These edges each lead to a new node with the next feature in the established order. To find the thresholds for the new feature, the threshold of the previous feature is applied to the BDT. This excludes the part of the BDT that does not apply to the current node in the MDD. In the remaining part of the BDT, all thresholds for the next feature are collected. These thresholds will be the new thresholds for the node of the MDD, and the edges will be created again. This whole process is repeated for every feature. When all features have been processed the leaves are created by applying all thresholds leading up to this leaf to the BDT.

The nodes of the MDD often have more than two edges while the Treelite library only supports nodes with two edges. So the MDD nodes are split into multiple nodes with two edges. This can be done based on how much the edge is expected to be used but this is not in the scope of the paper. Instead, the edges are split into two groups based on their threshold value. The lower half goes left and the higher half to the right. When the groups are still too big it is split again until there are enough edges. As MDD nodes cannot be used, all results about the MDD models regarding the depth and node count are expressed in the implemented binary node count instead of the MDD node count.

After the tree is constructed it will be trimmed. If two leafs next to each other have the same value, the tree can be trimmed as this node does not change the result. However, this can only be done in the subtree which contains only the last feature.

Not every BDT forest could be converted to an MDD forest. As the max depth setting increased, the MDD trees became bigger and eventually were stopped after reaching a thousand nodes, as it took too long to compute. This was the case for datasets D4, D5, D8 and D9. They were stopped after reaching a max of 6, 6, 5 and 8, respectively.

C. Testing

For all generated BDT and MDD random forest C-code was generated containing the predict function. Extra code was written to run the tests. This code continuously passes the test data through the predict function until 1 second or more has passed and the code has finished passing the whole test set through the predict function. So the amount of predictions done is always a whole multiple of the test set size. As the cycle will rarely end exactly after 1 second, the exact time is also measured. The predictions per second are calculated by dividing the total number of predictions by the measured time.

By dividing the total number of predictions done, through this time, a resulting value in predictions per second is retrieved.

Because some of the test datasets were too big to fit on the RAM of the Esp32, the size of the test data was limited to a thousand instances. For D9 this size was limited to 250 as the dataset has more features per instance.

The tests are compiled for and executed on two systems. The first is an ESP32-S3 part of an SQFMI Watchy, an open-source E-paper Watch. The second is a Linux server, called Anton running 2 Intex Xeon Silver 4216 processors.

IV. RESULTS

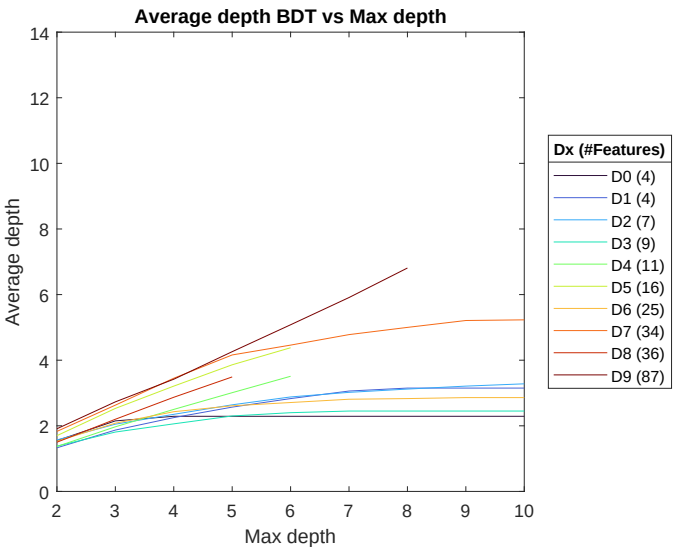

Fig. 4. A plot of average depth of the BDT against the Features of the dataset

After generating the BDT and converting to the MDD some general characteristics can be retrieved from the forests. For each tree, we find the average depth and average this over all trees in the forest. In Figure 5 datasets D4, D5, D7, D8 and D9 already are a bit higher than in Figure 4 at a max depth of

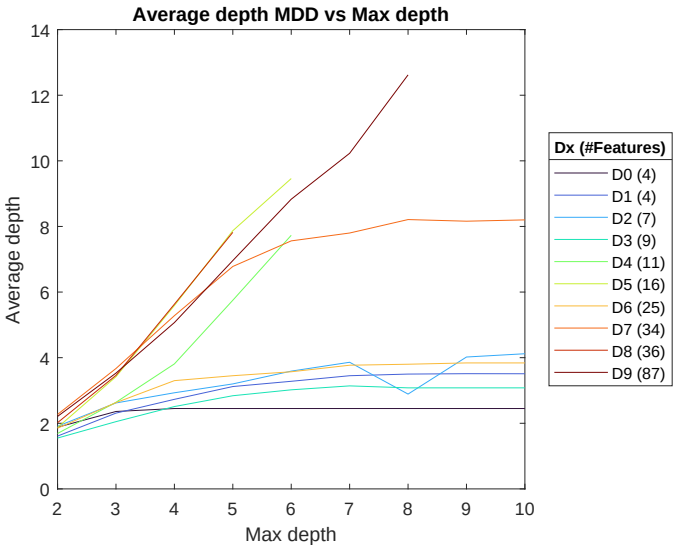

Fig. 5. A plot of average depth of the MDD against the Features of the dataset

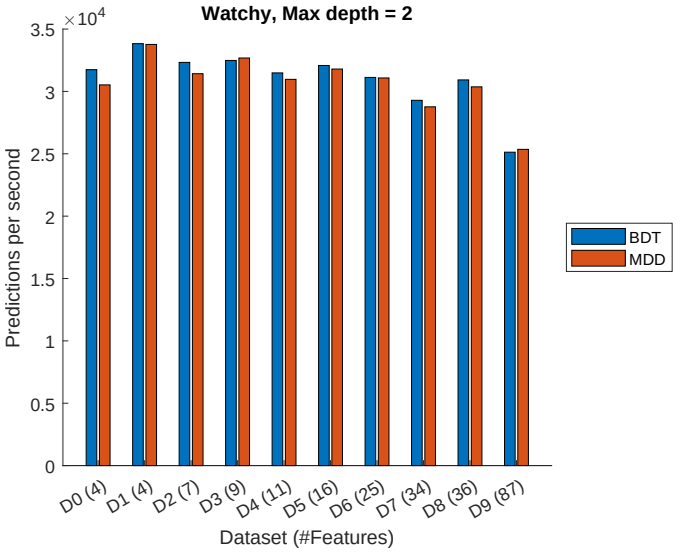

Fig. 6. Predictions per second on the Esp32 at max depth 2

2, by about 10.0 to 34.7%, but at their highest max depth they are between 56.8% and 124.1% higher. The other datasets are similar at a max of depth 2 with a increase of 12.3 to 21.7%, but have an increase of 7.0 to 34.3% at a max depth of 10. These datasets also stop increasing their average depth as the maximum depth continues to increase. This indicates that even with more room to grow their trees are already saturated. This is probably due to a lower number of instances or a lower feature count. This also happens with D7, although only at a higher depth.

A. Esp32

Running the test reveals that the BDT implementation is between -0.9% and 4.0% quicker than the MDD implementation at a max depth of 2 on the esp32 as seen in Figure 6. In two cases it is a little bit faster, but in most cases slower. D3 is the only case that is also faster with a max depth of 3.

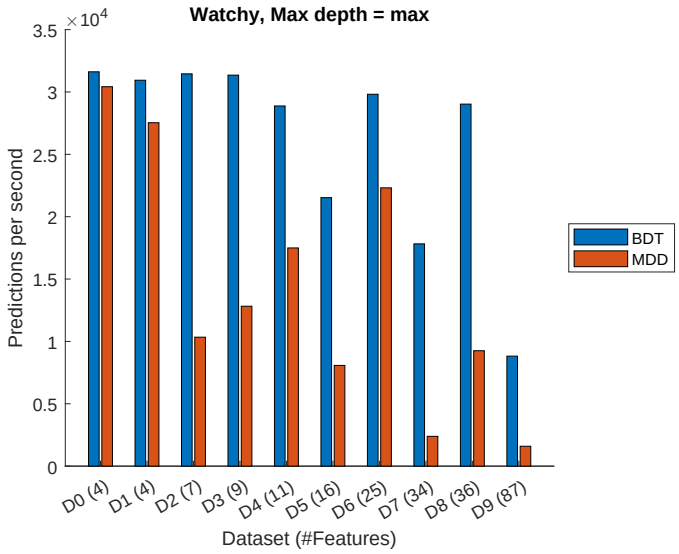

Fig. 7. Predictions per second on the Esp32 at the highest depth that fit in flash

Continuing to higher max depth values gives some problems. Some trees are too big to fit on the flash of the Esp32. The highest max depth settings are 8 for D2, 4 for D4, 4 for D5, 4 for D7, 3 for D8 and 5 for D9. This seems to occur once the average node count reaches a threshold between 120 and 140. Thus, in Figure 7 the BDT and MDD are compared for the highest depth values that could be measured on the Esp32.

For D0 the graph has remained nearly stable, at a max depth of 2 the BDT is 4.0% faster, and from depth 4 onwards it does not change anymore and has settled at 3.9% faster. For D1 this is similar but with the BDT 2.9% faster and it stabilizes at 12.4% at a max depth of 9. These are the datasets with the smallest difference. For the rest the BDT is from 33.6% faster at D6 to 645.6% faster at D7.

B. Linux server

On the Linux server, the results are a lot less close at a max depth of 2 as seen in Figure 8. Only with D9 the MDD is as fast as the BDT. For the rest of the datasets, the BDT is between 7.4% and 34.0% faster than the MDD.

Now that the system is not limited by storage anymore, more of the models can run the highest max depth of 10. Only D4, D5, D8 and D9 get to a max depth of 5, 5, 4 and 7, respectively. These MDD models got so large that compiling began to take a significant amount of time. As the Linux server was a shared resource, the process could not take too long.

As with Figure 7 for the Esp32, Figure 9 shows the predictions per second for the highest tested depth of the MDD. All the models that went to a max depth of 10 also reached a steady state at some point, as was seen with the Esp32. None of the MDD models stayed as close to the BDT models as with the Esp32. The closest was D1. The BDT was 20.0% faster at the max depth of 2 and 10 and only varied from 17.1% to 22.9% at the different max depth values. All

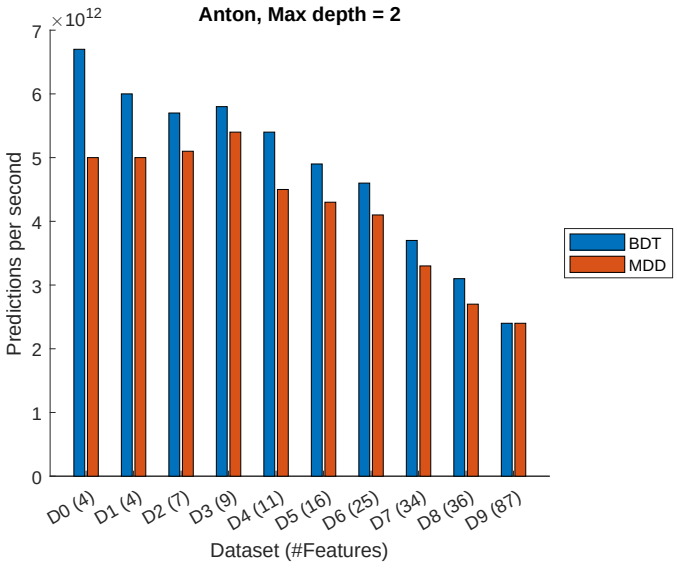

Fig. 8. Predictions per second on the Linux server at max depth 2

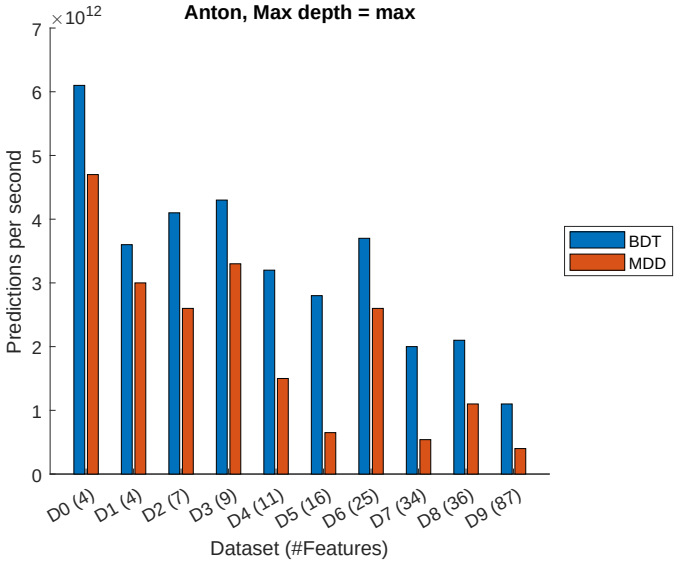

Fig. 9. Predictions per second on the Linux server at the highest depth that was tested

other datasets are faster for the BDT by between 29.8% at D0 and 330.8% at D5.

V. Discussion

There are only a couple of cases where the MDD is faster than the BDT, but this is only 0.9% which is probably only coincidental. The MDD stays close to the BDT at lower max depth values as only a couple of features per tree are present and these most likely have only 1 node per value. So the MDD can not differ a lot from the BDT. At a depth of 2, the BDT can have at most 3 nodes, so a maximum of 3 features. So in the worst case the average depth of the MDD is 3, only 50% higher. But as the max depth of the BDT increases, there can be more features in a tree and the MDD can grow bigger. So the dataset with only a few features keeps the BDT and MDD closer to each other, as there is an inherent feature limit per tree.

A. Binary nodes

Using only binary nodes, the real strength of the MDD could not be capitalized on. By having one node with more edges, there would only be the need for one node per feature per path. Resolving a feature within one node could be a lot faster than passing through multiple nodes, provided that the implementation of an MDD node can be done in a way that enables it to have a higher throughput than the BDT equivalent. The conversion is adding and reordering nodes from the BDT. Only combining nodes when there is a repeated threshold for a feature can locally benefit part of a tree.

This paper only looks at converting a BDT to an MDD, but this has the inherent flaw that in most cases the MDD will never be smaller or quicker than the BDT while using only binary nodes to make the MDD. Because it is created from the BDT and each threshold from the BDT is reused in the MDD in a less efficient order. By reordering the nodes it rarely happens that the number of nodes is decreased in the branch and most of the time it only adds another repeat of nodes that already exist and makes the tree bigger.

B. Future research

Although emulating MDD nodes with BDT nodes seems to be the biggest holdback in this paper, there are still a couple of things that can be done to improve this implementation.

1) Binary nodes order: While using the binary nodes to emulate MDD nodes, all thresholds for the feature of the current node need to be distributed to the left or right of this node. How this distribution is done can be improved. For example, determining what node is used most based on the training data. Now it is distributed based on threshold count.

2) Feature order: Selecting the right order to pass all features in the MDD could also play a crucial role in improving the MDD. This paper only used a basic way of selecting this order. Some experiments were done by just testing all feature orders and seeing which forest had the fewest nodes or the lowest depth. However, this of course becomes very computationally intensive very fast as the number of features increases. But this small experiment did show there is still some gain in this area.

3) FPGA: In [1] it is theorized that in certain cases the MDD model can be faster than the BDT model based on node count and tree depth, although this is presumably done with ideal MDD nodes. They also established that an FPGA would be the best way to implement the MDD. Maybe there could also be some possibilities to implement a better MDD node on the FPGA at the same time and could see the MDD jump ahead in some cases.

VI. Conclusion

In the best case for the MDD, it can be a bit quicker than the BDT. This only happens rarely and is closely matched by the BDT while always using more nodes. This also only happens

at the lowest maximum depths. While the MDD model can sort of keep up with the BDT at lower feature counts, however, they are still slower. At higher feature counts the model completely falls apart, creating enormous trees that are way slower than the BDT which has more moderately sized trees.

So in the current implementation using Treelite while only using binary nodes, converting binary decision trees to multi-valued decision diagrams has no added value to the prediction speed of a model.

REFERENCES

[1] H. Nakahara, A. Jinguji, S. Sato, and T. Sasao, "A random forest using a multi-valued decision diagram on an fpga," in *2017 IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL)*, 2017, pp. 266–271.

[2] S. Buschjäger and K. Morik, "Decision tree and random forest implementations for fast filtering of sensor data," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 1, pp. 209–222, 2018.