ITS Project: MDL for Decision Trees

Gerben Meijer, s1542648

I. INTRODUCTION

In this project, the goal was to understand and implement Minimum Description Length for Decision Trees as proposed in the MDL trees paper[1]. The aim of MDL trees is to provide a method for building decision trees that generalize well and has a rigid theoretical foundation. In this report, I will describe the methodology of implementing MDL trees and comparing them to the CART algorithm.

II. METHODOLOGY

A. Implementation

In order to test both algorithms, it is necessary for them to be implemented. Everything is implemented in Python 3 using the Numpy[2] library. The implementation of the CART algorithm is provided in the scikit-learn[3] library. The MDL trees have been implemented by me. For MDL trees only a specific subset of the features have been implemented, this is explained further later in this report.

B. Tests

The implementation has been tested on 3 datasets. The small dataset from the paper, the Iris dataset and the Agaricus Lepiota dataset. The last 2 datasets have been acquired at the UCI machine learning repository [4].

1) Small dataset: The first dataset is the small dataset provided in the paper. It provides a small toy problem and provides a way to check the output of the implemented algorithm to a degree. It is only used to check the output of the implementation and is not used for further testing.

2) The Iris dataset: The second dataset is the famous Iris dataset. The Iris dataset contains 3 classes of flowers and 4 continuous attributes. Each class has 50 samples. Because the MDL tree implementation only supports binary classification, the easiest class (Iris Setosa) has been removed. The remaining two classes are somewhat harder to separate and thus provide a challenge for the MDL tree algorithm. For testing purposes, the set has been split into 2 equally balanced datasets: train and test.

3) The Agaricus Lepiota dataset: The Agaricus Lepiota dataset, which I will refer to as mushroom dataset hereafter, consists of 8124 samples, 22 (discrete) attributes and 2 classes: edible and poisonous. It is used to test how MDL handles a larger problem. The dataset is split into two equally large sets: train and test. Both sets are equally balanced and shuffled before splitting since the attribute values seemed to depend on the position in the dataset.

III. MDL TREE CONSTRUCTION

In this section, the MDL trees will be described in more detail. I will mostly describe the parts I have also implemented, which covers most of the paper.

A. The Problem

Decision trees are a machine learning algorithm used to classify data. A problem with these decision trees is that they are prone to overfitting, which means that they fit the training data almost perfectly but don't generalize well to unseen data. To overcome this problem, decision trees are usually pruned. The MDL tree algorithm tries to solve overfitting by using the Minimum Description Length Principle as a heuristic for both tree inference and pruning.

B. Basic Setup

In the paper, the problem is translated to an encoding problem. Suppose that both the encoding and decoding side know all the attribute values for the data that we are trying to learn. The encoding side wants to encode the labels for that data in the most efficient way. To do so, the data will be encoded by encoding a decision tree that explains a portion of the data and by adding all exceptions to that tree. The goal of the MDL tree algorithm is to generate a tree that yields the lowest encoded message length in this encoding scheme. This gives the algorithm a trade-off between coding the tree and coding the exceptions to the tree. It will only keep nodes if these nodes shorten the total message length.

C. Exception Encoding

As stated above, the MDL tree algorithm uses a certain encoding for exceptions. The paper proposes a specific encoding that can encode the positions of the exceptions in an efficient manner. This encoding is used to encode the exceptions to a predicted class in a leaf node for the given training dataset. Say we have a leaf node that predicts the class c_a, where $|c_a| = 10$, meaning there are 10 samples of class c_a at this leaf. There are also 2 samples of c_b in the training data at this leaf. Let us encode the position of these exceptions in the dataset of 12 samples in a naive way first. Every correctly predicted sample as 0, every exception to that rule as 1. This will yield, for instance, the bit string $\{0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1\}$ and costs 12 bits.

The paper proposes to encode this differently. We have n samples and know that the upper bound of the number of exceptions $b = \frac{n+1}{2}$ for binary classification problems. For this specific instance of the data, we have k exceptions. Both sides know b and n, so we only need to encode the value of k and the positions. We can transmit k in $\log(b + 1)$ bits, where $\log$ is the 2-log. Next up we only have to encode the positions. There can only be $\binom{n}{k}$ possible orders of k ones in a string of n bits. Therefore we can encode this order in $\log(\binom{n}{k})$ bits. This means that we can define the cost of encoding the exceptions as the function L, where L is defined as:

$$L(n, k, b) = \log(b + 1) + \log \binom{n}{k}$$

To avoid problems with large numbers in computation on a computer, the equation has been rewritten such that the $\log$ is done first.

$$\log \binom{n}{k} = \log(n!) \ - \log(k!) - \log((n - k)!)$$

where $\log(x!)$ is computed using

$$\log(x!) = \sum_{i=1}^{x} \log(i)$$

D. Tree Encoding

To encode the full message, we also need to find an encoding for the decision tree itself. The paper suggests the following encoding. Every element in the tree is preceded with a bit that indicates whether the element is a node or a leaf. Decision nodes are preceded with a 1, leafs with a 0. For leafs, this bit is followed by the default class which also costs 1 bit for a binary classification problem. For nodes, the index of the attribute has to be listed. This cost is not constant because nodes deeper in the network have fewer attributes to pick from. Given a problem with A attributes, at a depth d and with $d' \leq d$ features already used we'll need $\log(A - d')$ bits to encode the chosen attribute.

1) Continuous values: For continuous values, we'll need an extra value indicating the threshold value. In the implementation in python, the second proposed solution in the paper was used. The list of all possible values of the selected attribute was sorted and then $\sqrt{m}$ evenly spaced possible threshold values were considered, where m denotes the number of distinct values. Because we only consider $\sqrt{m}$ possible values, we'll need $\log(\sqrt{m})$ bits more to encode this threshold.

E. Tree Inference

Now that we have all equations for determining the encoding cost in bits, it is time to infer a tree from data. Building a tree happens in 2 phases: in the tree inference and tree pruning. Tree inference happens by repeating the following 3 steps until there are no more leafs left to be expanded:

1) Pick a leaf x with varying classes, not at max depth and where $A - d' \neq 0$
2) For each possible attribute in the set of $A - d'$, compute the communication cost of choosing this attribute as the split attribute for the replacement node of x. (This is a local computation that I will explain in further detail)
3) Replace x with the node that splits on the attribute that has the lowest communication cost (always expand, even if the score is higher than the original score)

The cost of the replacement is calculated using the cost for the new decision node plus all the costs for the individual leaf nodes. These costs are determined using the encodings proposed above. More specific, for discrete nodes:

$$C_{node} = 1 + \log(A - d')$$

For continuous nodes, this is extended with the cost for the threshold. Remember that m was the number of distinct values that the continuous attribute can take.

$$C_{node} = 1 + \log(A - d') + \log(\sqrt{m})$$

For leafs the cost is defined as the cost of that leaf and the cost of encoding the exceptions:

$$C_{leaf} = 2 + c \cdot L(n, n_{ex}, \frac{n + 1}{2})$$

TABLE I

BOTH ALGORITHMS APPLIED TO THE BINARY IRIS DATASET

Metric	CART	MDL
Train accuracy	1.0	0.94
Test accuracy	0.92-0.94	0.94
Tree size (nodes + leafs)	9	3

Where n is the total number of samples for that leaf, n_{ex} is the number of exceptions, and c denotes the ratio between the exception encoding cost c_D and the tree encoding cost c_T. Higher values of c make larger trees more profitable as stated in the paper.

The total replacement cost can then, in loose notation, be defined as:

$$C_{replacement} = C_{node} + \sum_{leaf \in Leafs} C_{leaf}$$

The tree will extend until there is no more leaf to expand.

1) Tree Pruning: After the induction phase, the tree is most likely way too costly. The induction step extended the tree even if the total cost rose because that might, in the end, lead to a lower cost. In the pruning phase, it is the goal to remove all the nodes for which this was not the case. Starting from the leafs going up, every node is considered for pruning. This process is relatively simple. If replacing the node by a leaf yields a lower total cost then it will be pruned and replaced with said leaf.

IV. RESULTS

A. Small Example

In the paper, running the MDL tree algorithm on their small dataset yielded only a leaf with default class **P**. The implementation does give us the same tree. This is printed by the program as the tree `<Leaf c:1 N:14>`, where `c:1` denotes the default class **P** which is encoded as 1 and `N:14` denotes that there are 14 samples in this leaf.

B. Iris

As stated before, the binary Iris dataset consists of 100 samples that have to be classified based on 4 continuous attributes. The dataset has been split into a train and a test set. In Table I the results of this run can be seen.

The results indicate that the MDL algorithm produced a very small tree that managed to explain 94% of both the training and test set. The MDL algorithm seems have traded the 100% test set accuracy for a slightly lower accuracy with good generalization. The CART algorithm implementation of scikit-learn does not prune the tree, which explains the size of their tree. The CART tree has 100% training accuracy but has a test accuracy that varies between 0.92 and 0.94. This happens due to some randomization in the CART implementation. The results indicate that the MDL algorithm is performing well, but cannot yet significantly outperform its competitor on this dataset.

TABLE II

BOTH ALGORITHMS APPLIED TO THE MUSHROOM DATASET

Metric	CART	MDL
Train accuracy	1.0	1.0
Test accuracy	1.0	1.0
Tree size (nodes + leafs)	41	25

TABLE III

BOTH ALGORITHMS APPLIED TO THE 100 SAMPLE MUSHROOM DATASET

Metric	CART	MDL
Train accuracy	1.0	0.99
Test accuracy	0.78	0.95
Tree size (nodes + leafs)	17	9

C. Mushrooms

Next up is the larger mushrooms dataset. It has almost a hundred times more samples and might, therefore, provide more of a challenge to both algorithms. The results can be found in Table II

As visible in the results, both algorithms have a 100% accuracy on this dataset. This not very useful to compare the algorithms. As an extra test, both trees were only given (the same) 100 samples for training. This was done to test their generalization ability. The results are denoted in III

This shows a convincing difference between the algorithms. The ability of MDL to trade in training accuracy in favour of a smaller tree allows it to generalize better to unseen data. Meanwhile, the CART algorithm has completely overfitted on the training data.

This result introduced the need for a more specific test on the behaviour of the two algorithms. In Figure 1 the test accuracy of both algorithms is plotted over the number of training samples per class. This is an average over 10 shuffles of the dataset that were equally shuffled for both algorithms for every shuffle. This result shows the generalizing capabilities of MDL trees. Figure 2 shows the respective average tree size for the same 10 shuffles. It is clearly visible that the MDL

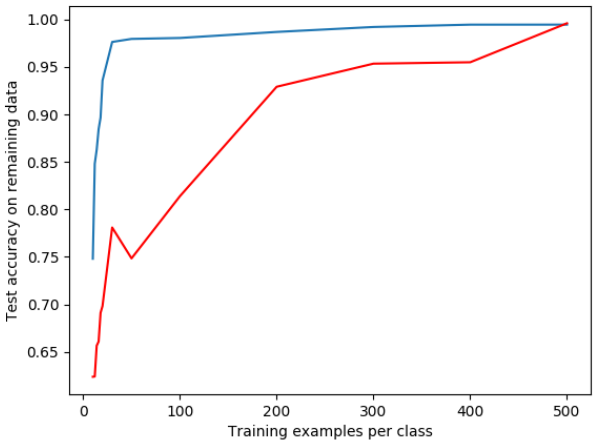

Fig. 1. Average test accuracy of MDL trees (blue) and CART trees (red)

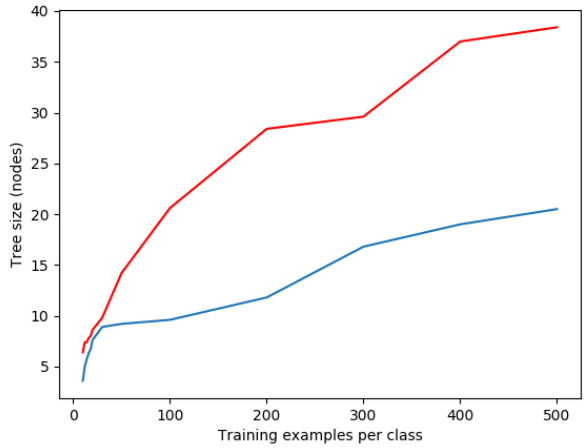

Fig. 2. Average tree size of MDL trees (blue) and CART trees (red)

tree algorithm handles small datasets extremely well. It only needs roughly 50 samples in total to learn this 22 attributes binary classification problem with $> 95\%$ accuracy.

V. CONCLUSION

The MDL tree algorithm is an algorithm for inducing and pruning decision trees from data based on the MDL principle. It manages to perform equal or better than the CART algorithm without pruning on cases where the training data is large enough and yields significant improvements on small datasets.

VI. DISCUSSION

As already stated before, the CART algorithm implementation from scikit-learn does not prune trees. While the results in this paper sure show the effectiveness of the MDL algorithm, the comparison is not entirely fair as CART overfits massively without pruning. Due to time constraints, it was impossible to implement C4.5 or CART with pruning for this report.

APPENDIX

The code produced for this project can be found at `https://github.com/Gerryflap/MDL_tree` and in the zip file.

REFERENCES

[1] J. Ross Quinlan and Ronald L. Rivest. Inferring decision trees using the minimum description length principle. *Information and Computation 80.3*, 1989. pages 227-248.

[2] Numpy. `http://www.numpy.org/`.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[4] Dua Dheeru and Efi Karra Taniskidou. UCI machine learning repository, 2017.