An Incremental MDL-based Decision Tree Learner

Harpreet Bhatia and Andrea Schuch

January 1, 2008

Contents

1 Introduction

This assignment is an extension of the work done by Quinlan and Rivest for encoding the decision tree using the MDL measure. But instead of batch pro-

cessing of the training data, our work aims to build an incremental learner. Our learner would be incrementally fed by incoming samples and decision nodes will be selected from the attributes using the crude two-part code MDL, which aims to minimize the composite length of the hypothesis and the data given the hypothesis i.e. $min(L(H) + L(D|H))$. There are multitude of challenges that we face in using the incremental MDL based decision learner, namely the right choice of coding scheme for the tree, the coding scheme for the exceptions, the time at which we need to restructure the tree and/or expand the tree. The choice of coding scheme is important because an inefficient coding measure can make our tree too compact or too spread out. The timing of the restructuring/expansion is critical as we would like to achieve the optimal tradeoff between a tree building process that is computationally too expensive and a tree building process that is too simplistic and error prone. In the end we would also like to see if the decision tree built by our incremental learner for n training samples is same as the decision tree built via the batch learning route for n training samples.

2 The MDL-measure

Supposing we are given a data set, for example Figure 7 on Page 14, which is drawn randomly from the population. We are then given the task to learn the underlying pattern in the given data and thereafter given the attributes we are required to predict the class of unseen data items.

One of the most trivial way to solve this problem is to memorize the attributes and class of all the given data items. If in future any data item comes with same sequence of attributes as the ones that has been memorized, the task of predicting the class would be to simply invoke the mapping of the stored sequence. But the above trivial way to solve this problem would expose the problem of overfitting, whereby our classifier is able to perform brilliantly on the training data set, but its performance deteriorates drastically on unseen data set. To perform equally well on unseen data set we need to find inherent patterns. For a given data set we can postulate infinitely many inherent patterns in the data i.e. hypotheses that predict the data. But which one is the best? There is no one best answer to this. Should we take the hypothesis which is simple and producing errors or should we choose the hypothesis that is complex but has high predictive power on training data. The minimum description length (MDL) principle provides an answer to this dilemma. Assuming we are able to somehow code into bits the hypothesis and the exceptions in data items that are not predicted by the hypothesis. Then as per crude two-part code MDL, the best hypothesis is the one that

has the minimum code length i.e.

Best Hypothesis = min (code length of hypothesis $L(H)$ + code length of the exceptions $L(D|H)$).

If the aim is to encode the data using a decision tree, then the coding would consist of two parts:

1. Coding of the decision tree i.e. coding the hypothesis
2. Coding the exceptions.

2.1 Coding of the Hypothesis

Coding the tree involves the following:

1. Coding the structure of tree.
2. Coding the attributes at the nodes.
3. Coding the cut values for non discrete values of the attributes
4. Coding the categories at the leaves.

Using the methodology adopted in [3] we describe a discrete attribute value decision tree using binary numbers.

0 would refer to a leaf while 1 would refer to an internal node. The code of an internal node i.e. 1 is followed by the code string for the attribute labeling the node which is further followed by the codes of the children of the current node for each possible attribute value in increasing order. The code for the decision tree is therefore the code of its root node.

Coding of the Structure Referring to the example in Figure 1 the code for the above tree is $1X1Y001Z01Y000$. Thus the number of bits needed to encode the structure of tree i.e. $1100101000 \equiv \log(2^{10}) \equiv 10$ bits.

As pointed out in [3] the simple scheme of labeling a leaf by 0 and an internal node by 1 is an inefficient way coding the tree for trees having higher arity. Thus as stated in [3] we will, in this assignment, be using the code length given by the formula $L(n,k,b) = L(n,k,\frac{n+1}{2}) = \log(\frac{n+1}{2}+1)+\log\binom{n}{k}$ [1], where n represents the total number of nodes (leaf and non leaf), k represents the number of non leaf nodes and b represents the maximum value of k.

[1] When writing log we always mean the basis 2 unless indicated otherwise.

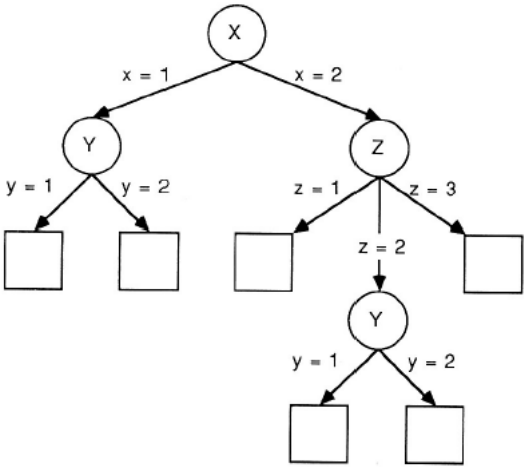

Figure 1: Example tree from [2].

Coding the Attributes As has been pointed out in [3] the length of the code needed to describe the attribute is $\log n$, where n is the number of available attributes at the node. An attribute is labeled as available if it has not been used along the path from the root to the current node.

Thus in the example tree given in Figure 1, the code length of the attribute X is $\log 3$, the code length of Z and Y in the left branch of the tree is $\log 2$ while the code length of the attribute Y below $Z = 2$ is $\log 1$. These coding lengths for the attributes are based on the Shannon-code $\sum_{i=1}^{N} P_i \log P_i$. Under the assumption that all attributes are equallly likely, the above Formula becomes equal to $\log n$.

Coding the Categories In case of a b-arity tree the coding of the categories would require $\log x$ bits where $x \leq b$, x is the number of categories left for each leaf. Thus for the example shown, the number of bits required to code the category at node Y in the leftmost sub tree is $\log 2 + \log 1 = 1$.

2.2 Coding of the Data Given the Hypothesis

As stated in [3] we shall be using the formula $L(n, k, b)$ for encoding the exceptions. $L(n, k, b) = \log(b+1) + \log \binom{n}{k}$. Where n = number of total data items, k = number of exceptions and b = maximum number of exceptions.

The total $L(D|H)$ score for the tree is given by the sum of local $L(D|H)$ scores for each leaf of the tree.

2.3 Discussion of the Measure

Since in our assignment we are using an incremental learner, we will not be comparing the MDL score for n_1 samples with MDL score of n_2 samples ($n_1 < n_2$) while trying to find the best tree. This is because even if one adds samples that are correctly classified by the tree, the MDL score increases with n as, although the code length of the tree remains constant, the length of coding the data increases with n. This represents an inherent weakness in the model as intuitively we would not expect the MDL score to increase if the tree is classifying the incoming samples correctly. This weakness is due to the fact that our model assumes the probability of making an error equal to the Laplace's Rule of Succession and the MDL score calculated is based on this prior probability. Hence, instead of comparing model with n_1 samples with model with n_2 sample we will be comparing models with same value of n. We will choose the tree that gives us the minimum value of the MDL score for a given value of n.

Secondly, a limitation of the two part MDL is that it does not define a practical way for finding the coding scheme even though the choice of the coding scheme for the tree plays a very important role in the construction of the tree. An inefficient coding scheme may either make the tree to compact or too spread out. This is because if the coding of $L(H)$ is too small wrt $L(D|H)$, then the tree will never expand and the tree will spread out too much in case it is the opposite.

3 The Algorithm

3.1 Representation of the Tree

Following [1] we choose a representation of the tree, where each *internal node* specifies

- the splitting attribute
- the different values of this attribute
- the positive and negative counts for each attribute value

and each *leaf node* specifies

- the classification
- and all classified instances.

(For effeciently comparing the MDL-measures, we might also store positive and negative counts for other available attributes in each node.)

3.2 Overview of the Algorithm

We pursue a modular approach in the definition of our algorithm. Figure 2 defines the overall procedure, making use of the following modules: The `update` procedure described in Figure 3 encodes all information about new sample, i.e. it stores the sample in the leaf node where it is classified and updates all the positive and negative counts along the path from the root node to that leaf. As described in Figure 4 `extend` expands a node turning a leaf node into a decision node taking the "best" of all available attributes for the split. Conversely, `prune` removes the subtree of a node turning it into a leaf node (which might require further pruning steps of parental nodes, see Figure 6 for details). The remaining procedure `restructure` (Figure 5) exchanges a suboptimal attribute at the current node for the best available[2] attribute and performs any changes in the structure of the tree that become necessary through this.

Except for `update`, each module only performs changes to the decision tree upon the improvement of some local MDL-score. We decided to have `restructure` called right after `update`, because exchanging an attribute high up in the tree might change its appearance considerably, and make the application of any of the other subalgorihms superfluous. Hence, they are only called in case `restructure` did not change the tree. Similarly, `extend` is only called if `prune` (which is called first, as it might lead to a reduction of the tree size) is "unsuccessful" (in the sense that it does not change the tree). See Appendix A for a detailed description of an example application of the algorithm.

[2]Attributes are *available* if they have not been used further up in the tree, i.e. along the path from the current node to the root.

1. `update` the tree with new instance
2. For all nodes along the path from the root to the leaf where the instance has been added: Try to `restructure` (i.e. exchange the splitting attribute and then re-extend as long as this improves the score).
3. If `restructure` did not change the tree:
   (a) Try to `prune` the node where the new sample has been added.
   (b) If `prune` did not change the tree: Try to `extend` the leaf node where the new sample has been added.

Figure 2: Outline of the overall algorithm.

1. Store the new sample at the leaf where it is classified in the existing tree (if the tree is empty, store at the root).
2. If the new sample cannot be classified by the existing tree, as one of the testing-attributes along the path are missing the new sample's value, grow a new branch and store the sample at its leaf.
3. Check if the leaf's classification is still the classification of the majority of all the leaves' samples. Otherwise: Switch the classification encoding positive samples as negative samples, and vice versa.
4. For all nodes along the path from this leaf node to the root node: Update the counts of negative and positive instances.

Figure 3: The updating algorithm. Item 2) is required for non-binary classification if the attribute values are not known in advance.

1. If the new sample is correctly classified by the leaf node, do nothing.
2. Otherwise, conduct the following test for all available attributes:
   (a) Split the node for this attribute.
   (b) Move all instances to the leaves, update the counts for positve and negative instances.
   (c) Calculate local MDL-scores of the resulting subtree (i.e. with the late leaf at root-position)
3. Calculate also the local MDL-score of the current node without additional splits.
4. Choose the attribute with the lowest MDL-score for the split or choose not to split if it would not improve the score.

Figure 4: The extension algorithm.

4 Experiments

4.1 Data Sets

4.1.1 SPECT Heart Data Set

The SPECT Heart Data Set classifies the data of each patient into the two categories "normal" and "abnormal" based on 22 categorial attributes. It does not have missing values, but can be expected to contain some noise.

4.1.2 Tic Tac Toe Endgame Data Set

The Tic Tac Toe endgame database encodes the complete set of possible board configurations at the end of tic-tac-toe games, where "x" is assumed to have played first. The target concept is "win for x" (i.e., true when "x" has one of 8 possible ways to create a "three-in-a-row"). It thus requires a binary classification based on 9 categorial ternary attributes (values: "x", "o" and "blank"). It does not contain missing values and can be expected to be noise-free. There was a note on the web-page saying that ID3 gives a "stripped-down" decision tree.

For each node along the path from the root down to the leaf where the new instance has been added:

1. Perform the following test:
   (a) Convert the current node in a leaf node moving all samples from below up to the current node, update.
   (b) Calculate extensions for all available attributes at the current node.
   (c) select the attribute whose extension has the lowest local MDL-score.
2. If the attribute with the lowest score is not equal to the current attribute:
   (a) Turn the current node into a leaf, abandoning all its subtrees.
   (b) Extend the current node splitting with the new attribute.
   (c) Recursively extend all newly created leaves until the score stops improving.

Figure 5: The restructuring algorithm. Instead of using a (much more efficient) pull-up procedure as in ID5-R [1], we simply extend the tree at the node where the attribute has been exchanged, applying the subalgorihm recursively on all newly created children until the tree is fully extended, again.

1. Conduct the following test on the current node:
   (a) Remove the current's node subtree, turning the current node into a leaf.
   (b) Move samples covered by this subtree up to the current node.
   (c) Take the majority classfication of the samples as classification for the current node.
   (d) Update the counts of the current node.
   (e) If the parent node of the current node becomes superfluous, in the sense that both its subtrees lead to the same classification, prune it as well.
2. Compare the MDL-score of pruned tree to unpruned tree. If using a local MDL-measure make sure that its scope corresponds to the scope of the pruning. Adopt the pruned tree if it improves (i.e. minimizes) the score.

Figure 6: The pruning algorithm.

4.2 Experimental Setups and Results

For the evaluation of our incremental decision tree learner, we set up the following criteria: Naturally, the learner's performance should be evaluated in comparison to a "gold standard" non-incremental decision tree learner, the ID3 implementation of the Weka workbench. Furthermore, it should be evaluated on different data sets to maintain a certain degree of "objectiveness" and because comparing the results on different learning problems might give interesting insights. Finally, we wanted to test the algorithm itself on the one hand, and to compare the performance of different MDL-measures on the other hand. For the former, we tested the algorithm's performance on the *training* set using only the data component of the MDL-score (i.e. the coding length of the data given the hypothesis $L(D|H)$ as criterion). We reasoned that to the degree that the data was free of noise, a good algorithm should then produce a tree that classifies the training data perfectly. Should it not, it would mean that the algorithm was not flexible enough to produce a powerful hypothesis.

It turned out, that our algorithm fulfills this requirement almost – but not fully: On the SPECT training data, our algorithm yields a tree with 73 positive and 7 negative examples when we disable the $L(H)$ part of the MDL-score. This 73% classification accuracy does not compare too well to

the Weka's ID3-implementation, which achieves 96% perfection, especially since ID3 (in contrast to our current setup) does not aim at overfitting the training data, but should also include a certain amount of generalization (altough the resulting 12-level deep ID3-tree seems very prone to overfitting to us). On the noise-free Tic Tac Toe training data, which ID3 manages to classify correctly, our algorithm again achieves a good, but not perfect result, making one classification error on 79 samples. We checked the misclassified samples and found that their correct classification would not only require the tree to be changed (e.g. extended) once, but twice! Hence, the reason for our algorithm's imperfect classification of training data is that it will only perform a combination of more than one steps (e.g. two subsequent extension steps), if the first step does not already lead to a score improvement. To explain it further we also calculated the same samples for ID3, and found that while the first algorithmic step did not improve our MDL-score, the entropy decreases with each extension making the information gain positive.

Using the full MDL-score according to Rivest ($L(H) + L(D|H)$), our algorithm produced a two-level deep decision tree with 58 positive and 22 negative samles on the training set from the SPECT training data. On the SPECT test set this decision tree achieved a performance of 115 positive and 72 negative examples. This classification accuracy of 61.50% is below the performance of Weka's ID3 (76.47%) but still well above random classification.

Another interesting finding we made during these experiments was that the $L(D|H)$ by [3] sometimes gives preference to trees where all the children of a node have the same classification. Further calculation revealed that indeed, this $L(D|H)$ score may be smaller when errors are distributed differently. In order to fix this, we calculated an alternative score for $L(D|H)$, by calculating L for the root node instead of summing up the L's of all leaves as in [3]. The effect was, that the "faulty" children disappeared, but at the same time the induced tree was much smaller in size. We conclude that – for some reason – our alternative way of calculating the coding length is less accurate (while the "faulty" children produced by the original score do not influence the classification accuracy, of course).

5 Discussion

5.1 Performance

We have built an MDL-based incremental decision tree learner. Our algorihm is limited to binary classification, but it can handle non-binary attribute val-

ues. Our experiments with different variations of Rivest's MDL-score showed that the algorithm's performance is well above random classification. However, we do not reach the performance level of Weka's ID3 algorithm.

From the side of the algorithm, there is only one limitation that might (partially) account for this. This limitation occurs in case a new sample requires a combination of more than one step (in particular two extension steps) to improve the score, and the first of these steps does not lead to an improvement of the score. In this case, our algorithm would not change the tree, even though this might lead to an improvement of the classfication performance. We did not fix this, due to two reasons: Firstly, there is no clear limit on the number of combined steps necessary until a score improvement is achieved. While it would be easy to have the algorithm check for two steps in advance, checking all possible comibinations of steps until limited only be the number of available attributes, would clearly affect efficiency. Secondly, changing the algorithm would not even be necesary if we could find an MDL-score which has the same bias as the information gain measure used by ID3 which prevents this problem (c.f. Section 4.2).

Apart from this, the algorithm has proven flexible enough to (over-)fit a noise-free data-set, just as it can build a more generalizing decision tree, depending on the score. Moreover, due to the great flexibility of our algorithm and the tree representation, we discovered during the experiments that under certain circumstances some MDL-scores prefer to split a node, even if the resulting children all have the same classification. We found this irritating, as obviously this cannot lead to an improvement of the classification accuracy. Although, again, it would have been possible to modify our algorihm such as to prevent this, we decided against it for two reasons: Firstly, it does not influence the classification accuracy as the resulting tree does not contain any added information. Secondly, we actually think this clearly is a problem of the MDL-score and not of the algorithm.

In sum, the algorithm's overall classification performance is crucially dependent on the MDL-measure in use, as the algorithm generally seems to have the ability adapt the tree to any new input samples. Concerning the MDL-score, the performance is dependent on the exactness by which it approaches the actual encoding lengths, but also relative size of the two endoding lenths to one another.

5.2 Efficiency

Our algorihm proved efficent enough to perform training and classification of our data sets within a reasonable amount of time (much less than ten minutes for SPECT). Nevertheless, we thought of the following ways to improve its

efficency:

For example, preventing the split of a node if all resulting leaves have the same classification (see previous Section) should improve the algorithm's efficiency. Likewise, calculating the MDL-score for the whole tree might be expensive (depending on the complexity of the score). It might be cheaper to store (part of) the score (e.g. in the tree) with dynamic programming or to use local scores (i.e. scores for subtrees calculated on the basis of those samples which are stored in the subtree) instead of the full tree's (global) score for the evaluation of algorithmic steps affecting only part of the tree. However, with most scores, a problem of compositionality arises in the latter case, as the sum of all local scores is not necessarily equal to the global overall score of the tree.

Clearly, `restructure` is the most expensive of all of our algorithmic procedures, especially because it (unlike ID5-R in [1]) does not use a pull-up algorithm, but builds anew the whole subtree underneath the exchanged attribute. Besides improving the efficency of `restructure` itself by adding a pull-up procedure, it could also be avoided to be called (unneccessarily frequently) in the first place. It would be worth to explore if the decision to skip the restructuring procedure could be based on measures, such as:

- Number of considered attributes decreasing with time
- Based on when restructuring was last performed on the particular node
- Based on the overall number of instances
- Based on on the percentage of data covered by the particular node
- How many wrongly classified instances

A Illustration of the Learning Process

Our example shows how the algorithm is dealing with the test set given in Figure 7. We are then given the task to learn the underlying pattern in the given data and thereafter given the attributes we are required to predict the class of unseen data items.

Our algorithm will work as follows:

Sample 1: When the first sample comes, the `update` module will be called. Since the tree is empty, it will create a leaf and label the class as N. Then for this given leaf, the `update` module will store the number of positive instances as 1 and number of negative instances as 0.

No	Attributes				Class
	Outlook	Temperature	Humidity	Windy	
1	sunny	hot	high	FALSE	N
2	sunny	hot	high	TRUE	N
3	overcast	hot	high	FALSE	P
4	rain	mild	high	FALSE	P
5	rain	cool	normal	FALSE	P
6	rain	cool	normal	TRUE	N
7	overcast	cool	normal	TRUE	P
8	sunny	mild	high	FALSE	N
9	sunny	cool	normal	FALSE	P
10	rain	mild	normal	FALSE	P
11	sunny	mild	normal	TRUE	P
12	overcast	mild	high	TRUE	P
13	overcast	hot	normal	FALSE	P
14	rain	mild	high	TRUE	N

Figure 7: Sample set adopted from [1]

Sample 2: When the second sample comes, the `update` module will be called. This new sample will share the same leaf as the first sample. The count of the leaf would be updated to 2 positives and 0 negatives.

Sample 3: When the third sample comes, the `update` module will take the third sample to the only existing leaf and update the count at the leaf to 2 positives and 1 negative.

Since, there is no node in the decision tree, the pruning step will not be required.

Next the restructure module will be called.

This will calculate the MDL score the existing tree as follows:

Coding of tree: Code length of structure of tree = 1 bit

Code length for the attribute = 0

Code length for the default class = 1 bit

Coding of data: The samples are N, N and P. Thus the L score is going to be:

$L(3, 1, 3) = \log 4 + \log \binom{3}{1} = 2 + \log 3 = 2 + 1.585 = 3.585$

Total code length $= 3.585 + 1 + 1 = 5.85$

After that for attribute which have different values in the give sample it will split the tree.

In the three samples we have so far, it will split based on *humidity*.

Thus the tree will look as follows:

The structure of tree would be $1H00$.

The Code length for the structure of tree $= L(3,1,2) = \log(3) + \log\binom{3}{2} = 3.17$

The code length for attributes $= \log 4 = 2$

The code length for default classes $= \log 2 + \log 1 = 1$

The code length of data $= L(2,0,2) + L(1,0,1) = \log 3 + \log 2 + \log 2 + \log 1 = 3.585$. Therefore the total code length $= 3.585 + 1 + 2 + 3.17 = 9.665$.

Similarly, the score for the tree splitting along the attribute windy would be calculated.

Since the MDL score of the 1 node tree is greater than 5.85, the decision tree will not be expanded.

Sample 4: When the fourth sample comes, the `update` module will take the fourth sample to the only existing leaf and update the count at the leaf to 2 positives and 2 negative.

Since, there is no node in the decision tree the pruning step will not be required.

Next the restructure module will be called.

This will calculate the MDL score the existing tree as follows:

Coding of tree: Code length of structure of tree = 1 bit

Code length for the attribute = 0 bit

Code length for the default class = 1 bit

Coding of data: The samples are N, N and P. Thus the L-score is going to be:

$L(4,2,4) = \log 5 + \log\binom{4}{2} = 2.32 + \log 6 = 2 + 2.58 = 4.58$

Total code length $= 4.58 + 1 + 1 = 6.85$

Again the MDL score after splitting will be greater than the MDL score before splitting. Hence the tree will continue to remain in the unexpanded form.

Sample 5: When the fifth sample comes, the `update` module will take the fifth sample to the only existing leaf and update the count at the leaf to 2 positives and 3 negatives.

Since, there is no node in the decision tree the pruning step will not be required.

Next the restructure module will be called.

This will calculate the MDL score the existing tree as follows:

Coding of tree: Code length of structure of tree = 1 bit

Code length for the attribute = 0

Code length for the default class = 1 bit

Coding of data: The samples are N, N and P. Thus the L score is going to be:

$L(5, 3, 5) = \log 6 + \log \binom{5}{3} = 2.58 + \log 10 = 2.58 + 3.32 = 5.9$

Total code length = $5.9 + 1 + 1 = 7.9$

Again the MDL score after splitting will be greater than the MDL score before splitting. Hence the tree will continue to remain in the unexpanded form. As described in Section 2.3, this is probably due to infefficient coding, in particular a bad relation between the codinglenghts $L(H)$ and $L(D|H)$.

References

[1] Utgoff, P. E. (1989), Incremental Induction of Decision Trees, Machine Learning, 4, p.161-186

[2] C.S. Wallace1 and J.D. Patrick (1993): Coding Decision Trees Journal Machine Learning Springer Netherlands, Volume 11, Number 1, p 7 – 22

[3] J. R. Quinlan and R. L. Rivest: Inferring (1989): Decision trees using the minimum description length principle. In Information and Computation 80 (3), p. 227 – 248