Learning Optimal Decision Trees Using Caching Branch-and-Bound Search

Gael Aglin, Siegfried Nijssen, Pierre Schaus
{firstname.lastname}@uclouvain.be
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium

Abstract

Several recent publications have studied the use of Mixed Integer Programming (MIP) for finding an optimal decision tree, that is, the best decision tree under formal requirements on accuracy, fairness or interpretability of the predictive model. These publications used MIP to deal with the hard computational challenge of finding such trees. In this paper, we introduce a new efficient algorithm, DL8.5, for finding optimal decision trees, based on the use of itemset mining techniques. We show that this new approach outperforms earlier approaches with several orders of magnitude, for both numerical and discrete data, and is generic as well. The key idea underlying this new approach is the use of a cache of itemsets in combination with branch-and-bound search; this new type of cache also stores results for parts of the search space that have been traversed partially.

Introduction

Decision trees are among the most widely used machine learning models. Their success is due to the fact that they are simple to interpret and that there are efficient algorithms for learning trees of acceptable quality.

The most well-known algorithms for learning decision trees, such as CART (Breiman et al. 1984; Quinlan 1993), are greedy in nature: they grow the decision tree top-down, iteratively splitting the data into subsets.

While in general these algorithms learn models of good accuracy, their greedy nature, in combination with the NPhardness of the learning problem (Laurent and Rivest 1976), implies that the trees that are found are not necessarily optimal. As a result, these algorithms do not ensure that:

- the trees found are the most accurate for a given limit on the depth of the tree; as a result, the paths towards decisions may be longer and harder to interpret than necessary;
- the trees found are the most accurate for a given lower bound on the number of training examples used to determine class labels in the leaves of the tree;
- the trees found are accurate while satisfying additional constraints such as on the fairness of the trees: in their predictions, the trees may favor one group of individuals over another.

With the increasing interest in explainable and fair models in machine learning, recent years have witnessed a renewed interest in alternative algorithms for learning decision trees that can provide such optimality guarantees.

Most attention has been given in recent years and in prominent venues to approaches based on mixed integer programming (Bertsimas and Dunn 2017; Verwer and Zhang 2019; Aghaei, Azizi, and Vayanos 2019). In these approaches, a limit is imposed on the depth of the trees that can be learned and a MIP solver is used to find the optimal tree under well-defined constraints.

However, earlier algorithms for finding optimal decision trees under constraints have been studied in the literature, of which we consider the DL8 algorithm of particular interest (Nijssen and Fromont 2007; 2010). The existence of this earlier work does not appear to have been known to the authors of the more recent MIP-based approaches, and hence, no comparison with this earlier work was carried out.

DL8 is based on a different set of ideas than the MIP-based approaches: it treats the paths of a decision tree as itemsets, and uses ideas from the itemset mining literature (Agrawal et al. 1996) to search through the space of possible paths efficiently, performing dynamic programming over the itemsets to construct an optimal decision tree. Compared to the MIP-based approaches, which most prominently rely on a constraint on depth, DL8 stresses the use of a minimum support constraint to limit the size of the search space. It was shown to support a number of different optimization criteria and constraints that do not necessarily have to be linear.

In this paper, we present a number of contributions. We will demonstrate that DL8 can also be applied in settings in which MIP-based approaches have been used; we will show that, despite its age, it outperforms the more modern MIP-based approaches significantly, and is hence an interesting starting point for future algorithms.

Subsequently, we will present DL8.5, an improved version of DL8 that outperforms DL8 by orders of magnitude. Compared to DL8, DL8.5 adds a number of novel ideas:

- it uses branch-and-bound search to cut large additional parts of the search space;
- it uses a novel caching approach, in which we store also

store information for itemsets for which the search space has been cut; this allows us to avoid redundant computation later on as well;

- we consider a range of different branching heuristics to find good trees more rapidly;
- the algorithm has been made any-time, i.e. it can be stopped at any time to report the best tree it has found so far.

In our experiments we focus our attention on traditional decision tree learning problems with little other constraints, as we consider these learning problems to be the hardest. How ever, we will show that DL8.5 remains sufficiently close to DL8 that the addition of other constraints or optimization criteria is straightforward.

In a recent MIP-based study, significant attention was given to the distinction between binary and numerical data (Verwer and Zhang 2019). We will show that DL8.5 outperforms this method on both types of data.

Our implementation is publicly available at `https://github.com/aglingael/dl8.5` and can easily be used from Python.

This paper is organized as follows. The next section presents the state of the art of optimal decision trees induction. Then we present the background on which our work relies, before presenting our approach and our results.

Related work

In our discussion of related work, we will focus our attention on alternative methods for finding optimal decision trees, that is, decision trees that achieve the best possible score under a given set of constraints.

Most attention has been given in recent years to MIP-based approaches. (Bertsimas and Dunn 2017) developed an approach for finding decision trees of a maximum depth K that optimize misclassification error. They use K to model the problem in a MIP model with a fixed number of variables; a MIP solver is then used to find the optimal tree.

(Verwer and Zhang 2019) proposed BinOCT, an optimization of this approach, focused on how to deal with numerical data. To deal with numerical data, decision trees need to identify thresholds that are used to separate examples from each other. A MIP model was proposed in which fewer variables are needed to find high-quality thresholds; consequently, it was shown to work better on numerical data.

A benefit of MIP-based approaches is that it is relatively easy from a modeling perspective to add linear constraints or additional linear optimization criteria. (Aghaei, Azizi, and Vayanos 2019) exploit this to formalize a learning problem that also takes into account the fairness of a prediction.

(Verwer and Zhang 2019) recently proposed a Constraint Programming (CP) approach to solve the same problem. It supports a maximum depth constraint and a minimum support constraint, but only works for binary classification tasks. It also relies on branch-and-bound search and caching, but uses a less efficient caching strategy. The approach in the present paper is easily implemented and understood without relying on CP systems.

Another class of methods for learning optimal decision trees is that based on SAT Solvers (Narodytska et al. 2018; Bessiere, Hebrard, and O'Sullivan 2009). SAT-based studies, however, focus on a different type of decision tree learning problem than the MIP-based approaches: finding a decision tree of limited size that performs 100% accurate predictions on training data. These approaches solve this problem by creating a formula in conjunctive normal form, for which a satisfying assignment would represent a 100% accurate decision tree. We believe there is a need for algorithms that minimize the error, and hence we focus on this setting.

Most related to this work is the work of (Nijssen and Fromont 2007; 2010) on DL8, which relies on a link between learning decision trees and itemset mining. Similarly to MIP-based approaches, DL8 allows to find optimal decision trees minimizing misclassification error. DL8 does not require a depth constraint; it does however assume the presence of a minimum support constraint, that is, a constraint on the minimum number of examples falling in each leaf. In the next section we will discuss this approach in more detail. This discussion will show that DL8 can easily be used in settings identical to those in which MIP and CP solvers have been used. Subsequently, we will propose a number of significant improvements, allowing the itemset-based approach to outperform MIP-based and CP-based approaches.

Background

Itemset mining for decision trees

We limit our discussion to the key ideas behind DL8, and assume that the reader is already familiar with the general concepts behind learning decision trees.

DL8 operates on Boolean data. As running example of such data, we will use the dataset of Table 1a, which consists of three Boolean features and eleven examples. The optimal decision tree for this database can be found in Figure 1a. While it may seem a limitation that DL8 only operates on Boolean data, there are straightforward ways to transform any tabular database in a Boolean database: for categorical attributes, we can create a Boolean column for each possible value of that attribute; for numerical attributes, we can create Boolean columns for possible thresholds for that attribute.

DL8 takes an itemset mining perspective on learning decision trees. In this perspective, the binary matrix of Table 1a is transformed into the transactional database of Table 1b. Each transaction of the dataset contains an itemset describing the presence or absence of each feature in the dataset. More formally, the database $\mathcal{D}$ can be thought of as a collection $\mathcal{D} = \{(t, I, c) \mid t \in \mathcal{T}, I \subseteq \mathcal{I}, c \in \mathcal{C}\}$, where $\mathcal{T}$ represents the transaction or rows identifiers, $\mathcal{I}$ is the set of possible items, and $\mathcal{T}$ is the set of class labels; within $\mathcal{I}$ there are two items (one positive, the other negative) for each original Boolean feature, and each itemset I contains either a positive or a negative item for every feature.

Using this representation, every path in a decision tree can be mapped to an itemset $I \subseteq \mathcal{I}$, as shown in Figure **??**. For instance, the last path in this tree corresponds to the itemset $\{\neg a, \neg b, \neg c\}$. Please note that multiple paths can be mapped to the same itemset.

A	B	C	classe
0	1	1	0
1	0	1	1
0	0	1	1
0	1	0	0
1	0	0	1
0	0	0	0
0	0	1	0
1	1	0	1
0	0	0	1
0	0	1	0
0	0	0	1

(a) Binary matrix

rowId	items	class
1	a, b, c	0
2	a, b, c	1
3	a, b, c	1
4	a, b, c	0
5	a, b, c	1
6	a, b, c	0
7	a, b, c	0
8	a, b, c	1
9	a, b, c	1
10	a, b, c	0
11	a, b, c	1

(b) Transactional database

Table 1: Example database

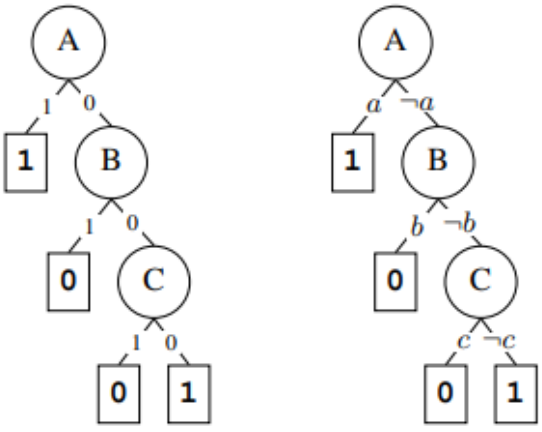

Figure 1: Optimal tree corresponding to database of Table 1. Max depth = 3 and minimum examples per leaf = 1

For every itemset I, we define its cover to be $cover(I) = \{(t', I', c') \in \mathcal{D} | I \subseteq I'\}$: the set of transactions in which the itemset is contained; the class-based support of an itemset is defined as $Sup(I, c) = |\{(t', I', c') \in cover(I) | c' = c\}|$, and can be used to identify the number of examples for a given class c in a leaf. Based on class-based supports, the error of an itemset is defined as:

$$leaf\_error(I) = |cover(I)| - \max_{c \in \mathcal{C}} \{Sup(I, c)\} \quad (1)$$

Unlike the CP-based approach, our error function is also valid for classification tasks involving more than 2 classes.

The canonical decision tree learning problem that we study in this work can now be defined as follows using itemset mining notation. Given a database $\mathcal{D}$, we wish to identify a collection $\mathcal{DT} \subseteq \mathcal{I}$ of itemsets such that

- the itemsets in DT represent a decision tree;
- $\sum_{I \in \mathcal{DT}} leaf\_error(I)$ is minimal;
- for all $I \in \mathcal{DT} : |I| \leq maxdepth$, where $maxdepth$ is the maximum depth of the tree;
- for all $I \in \mathcal{DT} : |cover(I) \geq minsup$, where $minsup$ is a minimum support threshold.

As stated earlier, in DL8, the maximum depth constraint is not required; MIP-based approaches have ignored the minimum support constraint.

DL8 Algorithm

Algorithm 1: $DL8(maxdepth, minsup)$

```
1  struct BestTree{ tree : Tree; error : float }
2  cache ← HashSet < Itemset, BestTree >
3  (τ, b) ← DL8 − Recurse(∅)
4  return τ
5  Procedure DL8.5-Recurse(I)
6      solution ← cache.get(I)
7      if solution was found then
8          return (solution.tree, solution.error)
9      if leaf_error(I) = 0 or |I| = maxdepth then
10         return (make_leaf(I), leaf_error(I))
11     (τ, b) ← (make_leaf(I), leaf_error(I))
12     for all attributes i do
13         if |cover(I ∪ {i})| ≥ minsup and
             |cover(I ∪ {¬i})| ≥ minsup then
14             (τ1, e1) ← DL8-Recurse(I ∪ {i})
15             if e1 ≤ b then
16                 (τ2, e2) ← DL8-Recurse(I ∪ {i})
17                 if e1 + e2 ≤ b then
18                     (τ, b) ←
                         (make_tree(i, τ1, τ2), e1 + e2)
19     cache.store(I, BestTree(τ, b))
20     return (τ, b)
```

A high-level perspective of the DL8 algorithm is given in Algorithm 2. Essentially, the algorithm recursively enumerates itemsets using the `DL8-Recurse(I)` function. The

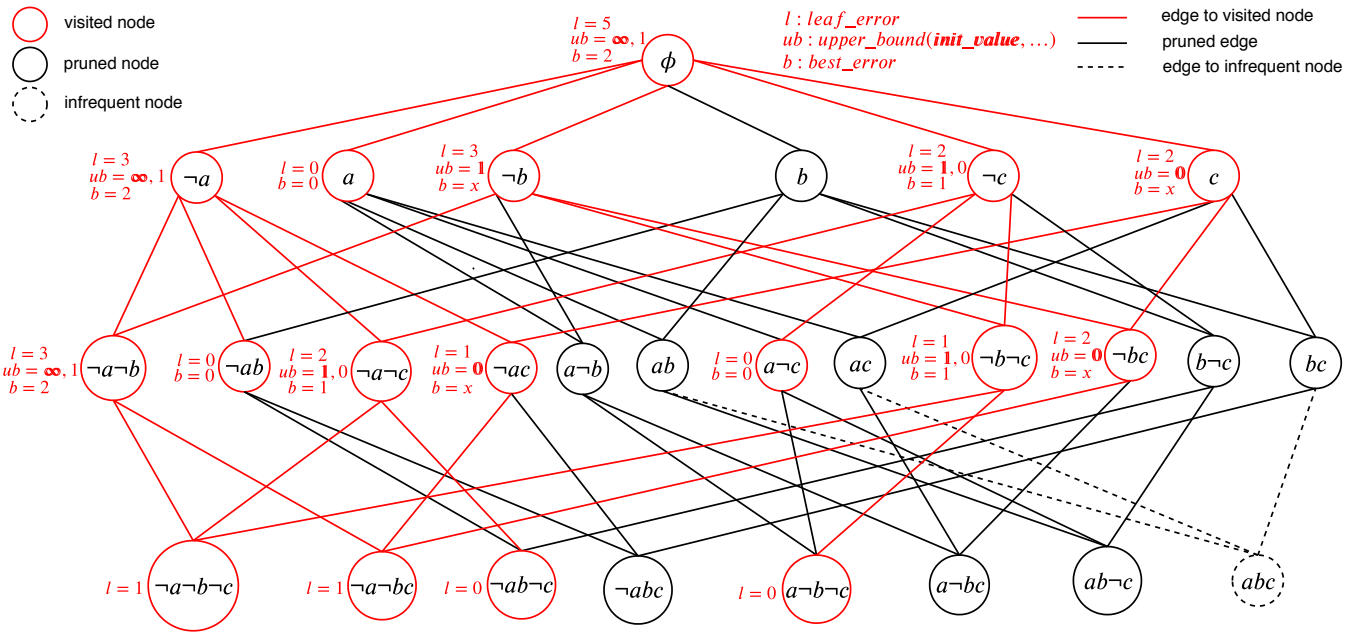

Figure 2: : Complete itemset lattice for introduction database and DL8.5 search execution

post-condition of this function is that it returns the optimal decision tree for the transactions covered by itemset I, together with the quality of that tree. This optimal tree is calculated recursively using the observation that the best decision tree for a set of transactions can be obtained by considering all possible ways of partitioning the set of transactions into two, and determining the best tree for each partition recursively.

Figure 2 illustrates the search space of itemsets for the dataset of Table 1, where all the possible itemsets are represented. Intuitively, DL8 starts at the top node of this search space, and calculates the optimal decision tree for the root based on its children.

A distinguishing feature of DL8 is its use of a cache. The idea behind this cache is to store the result of a call to `DL8-Recurse(I)`. Doing so is effective as the same itemset can be reached by multiple paths in the search space: itemset ab can be constructed by adding b to itemset a, or by adding a to itemset b. By storing the result, we can reuse the same result for both paths.

Note that the optimal decision tree for the root can only be calculated after all its children have been considered; hence, the algorithm will only produce a solution once the entire search space of itemsets has been considered.

In our pseudocode, we use the following other functions. Function $make\_leaf(I)$ returns a decision tree with one node, representing a leaf that predicts the majority class for the examples covered by I. The function $make\_tree(i, \tau_1, \tau_2)$ returns a tree with a test on attribute i, and subtrees τ_1 and τ_1.

The code illustrates a number of optimizations implemented in DL8:

Maximum depth pruning In line 9 the search is stopped as soon as the itemsets considered are too long;

Minimum support pruning In line 13 an attribute is not considered if one of its branches has insufficient support ; in our running example, the itemset $\{a, b, c\}$ is not considered due to this optimization;

Purity pruning In line 9 the search is stopped if the error for the current itemset is already 0;

Quality bounds In the loop of lines 12–18, the best solution found among the children is maintained, and used to prune the second branch for an attribute if the first branch is already worse than the best solution found so far.

We omit a number of optimizations in this pseudo-code that can be found in the original publication, in particular, optimizations that concern the incremental maintenance of data structures. While we will use most of these optimizations in our implementation as well, we do not discuss thesein detail here for reasons of simplicity.

The most important optimization in DL8 that we do not use in this study is the *closed itemset mining* optimization. The reason for this choice is that this optimization is hard to combine with a constraint on the depth of a decision tree. Similarly, while DL8 can be applied to other scoring functions than error, as long as the scoring function is *additive*, we prioritize accuracy and the depth constraint here as we focus on solving the same problem as in recent MIP-based studies.

Our approach: DL8.5

As identified in the introduction, DL8 has a number of weaknesses, which we will address in this section.

The most prominent of these weaknesses is that the size of the search tree considered by DL8 is unnecessarily large.

Reconsider the example of Figure 2, in which DL8's pruning approach does not prune any node except from one infrequent itemset (abc). We will see in this section that a new type of caching brand-and-bound search can reduce the number of itemsets considered significantly.

The pseudo-code of our new algorithm, DL8.5, is presented in Algorithm 2. DL8.5 inherits a number of ideas from DL8, including the use of a cache, the recursive traversal of the space of itemsets, and the use of depth and support constraints to prune the search space.

The main distinguishing feature of DL8.5 concerns its use of bounds during the search.

In DL8.5, the recursive procedure `DL8.5-Recurse` has an additional parameter, *init_ub*, which represents an upper-bound on the quality of the decision trees that the recursive procedure is expected to find. If no sufficiently good tree can be found, the procedure returns a tree of type `NO TREE`. Initially, the upper-bound that is used is $+\infty$ (line 3). However, as soon as the recursive algorithm has found one decision tree, or has found a better tree than earlier known, the quality of this decision tree, calculated in line 21, is used as upper-bound for future decision trees and is communicated to the children in the search tree (line 25, line 14, 18).

The upper-bound is used to prune the search space using a test in line 17; intuitively, as soon as we have traversed one branch for an attribute, and the quality of that branch is already worse than accepted by the bound, we do not consider the second branch for that attribute.

In line 18 we use the quality of the first branch to bound the required quality of the second branch further.

An important modification involves the interaction of the bounds with the cache. In DL8.5, we store an itemset also if no solution could be found for the given bound (line 29 is still executed even if the earlier loop did not find a tree).

In this case, the special value `NO_TREE` is associated with the itemset in the cache, and the upper-bound used during the last search is stored. The benefit of doing so is that at a later moment, we may reuse the fact that for a given itemset and bound, no sufficiently good decision tree can be found. In particular, in line 9, when the current bound (*init_ub*) is worse than the stored upper-bound for a `NO TREE` itemset, we return the `NO TREE` indicator immediately.

Other modifications in comparison with DL8 improve the anytime behavior of the algorithm. In line 6 the search can be interrupted when a time-out is reached, and line 12 offers the possibility to consider the attributes in a specific heuristic order to discover good trees more rapidly.

A number of different heuristics could be considered. In our experiments, we consider three: the original order of the attributes in the data, in increasing or in decreasing order of information gain (such as used in C4.5 and CART).

Our modifications of DL8 improve drastically the pruning of the search space. Figure 2 indicates which additional nodes are pruned during the execution of DL8.5 (for an alphabetic order of the attributes). At the end, 17 nodes are visited instead of 27.

Figure 3 shows a part of the execution of DL8.5 in more detail. The initial value of the upper-bound at node ϕ is $+\infty$ (line 11). The attribute A provides an error of 2; the upper-

Algorithm 2: $DL8.5(maxdepth, minsup)$

```
1  struct BestTree{init_ub : float; tree : Tree; error : float}
2  cache ← HashSet < Itemset, BestTree >
3  bestSolution ← DL8-Recurse(∅, ∞)
4  return bestSolution.tree
5  Procedure DL8-Recurse(I, init_ub)
6  Procedure DL8.5-Recurse(I, init_ub)
7      if leaf_error(I) = 0 or |I| = maxdepth or time-out is reached then
8          return BestTree(init_ub, make_leaf(I), leaf_error(I))
9      solution ← cache.get(I)
10     if solution was found and ((solution.tree ≠ NO_TREE) or (init_ub ≤ solution.init_ub)) then
11         return solution
12     (τ, b, ub) ← (NO_TREE, +∞, init_ub)
13     for all attributes i in a well-chosen order do
14         if |cover(I ∪ {i})| ≥ minsup and |cover(I ∪ {¬i})| ≥ minsup then
15             sol_1 ← DL8.5-Recurse(I ∪ {i}, ub)
16             if sol_1.tree = NO_TREE then
17                 continue
18             if sol_1.error ≤ ub then
19                 sol_2 ← DL8.5 − Recurse(I ∪ {i}, ub − sol_1.error)
20                 if sol_2.tree = NO_TREE then
21                     continue
22                 feature_error ← sol_1.error + sol_2.error
23                 if feature_error ≤ ub then
24                     τ ← make_tree(i, sol_1.tree, sol_2.tree)
25                     b ← feature_error
26                     ub ← b − 1
27                 if feature_error = 0 then
28                     break
29     solution ← BestTree(init_ub, τ, b)
30     cache.store(I, solution)
31     return solution
```

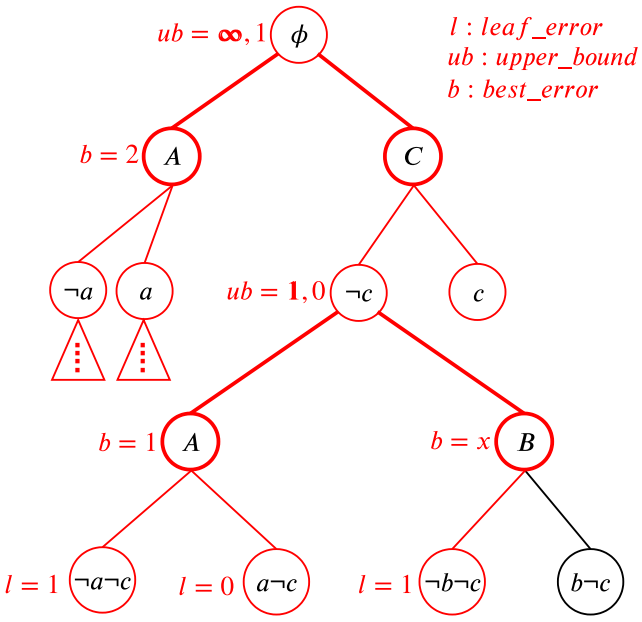

Figure 3: Example of pruning

bound value is subsequently updated from $+\infty$ to 1 in line 25. In the first branch for attribute C, the new value of the upper-bound is passed down recursively (line 14). Notice that the initial value of the upper-bound at node $\neg c$ is 1. At this node, the attribute A is first visited and provides an error of 1 by summing errors of $\neg a \neg c$ and $a \neg c$ (line 21). The upper-bound for subsequent attributes is then updated to 0 and passed down recursively to the first branch of attribute B. After visiting the first item $\neg b \neg c$ the obtained error is 1 and greater than the upper-bound of 0. The second item is pruned as the condition of line 18 is not satisfied. So, there is no solution by selecting the attribute B, which leads to storing the value `NO TREE` for this itemset. This error value is represented in Figures 2 and 3 by the character x.

The reuse of the cache is illustrated for itemset $\neg ac$ (Figure 2). The first time we encounter this itemset, we do so coming from the itemset $\neg a$ for an upper-bound of zero; after the first branch, we observe that no solution can be found for this bound, and we store `NO TREE` for this itemset. The second time we encounter $\neg ac$, we do so coming from the parent c, again with an upper-bound of 0. From the cache we retrieve the fact that no solution could be found for this bound, and we skip attribute A from further consideration.

Results

In our experiments we answer the following questions:

Q1 How does the performance of DL8.5 compare to DL8, MIP-based and CP-based approaches on binary data?

Q2 What is the impac*TO*f different branching heuristics on the performance of DL8.5?

Q3 What is the impac*TO*f caching incomplete results in DL8.5 (NO TREE itemsets)?

Q4 How does the performance of DL8.5 compare to DL8, MIP-based and CP-based approaches on continuous data?

As a representative MIP-based approach, we use BinOCT, as it was shown to be the best performing MIP-based approach in a recent study (Verwer and Zhang 2019). The implementations of BinOCT[1], DL8 and the CP-based approach[2] used in our comparison were obtained from their original authors, and use the CPLEX 12.9[3] and OscaR[4] solvers. Experiments were performed on a server with an Intel Xeon E5-2640 CPU, 128GB of memory, running Red Hat 4.8.5-16.

To respect the constrain*TO*f the CP-based algorithm all the datasets used in our experiments have binary classes. We compare our algorithms on 24 binary datasets from CP4IM[5], described in the first columns of Table 2.

Similar to Verwer and Zhang (2019), we run the different algorithms for 10 minutes on each dataset and for a maximum depth of 2, 3 and 4. All the tests are run with a minimum suppor*TO*f 1 since this is the setting used in BinOCT.

We do not spli*TO*ur datasets in training and test sets since the focus of this work is on comparing the computational performance of algorithms that should generate decision trees of the same quality. The benefits of optimal decision trees were discussed in (Bertsimas and Dunn 2017).

We compare a number of variants of DL8.5. The following table summarizes the abbreviations used.

Abbreviation	Meaning
d.o.	the original order of the attributes in the data is used as branching heuristic
asc	attributes are sorted in increasing value of information gain
desc	attributes are sorted in decreasing value of information gain
n.p.s.	no partial solutions are stored in the cache

Table 2 shows the results for a maximum depth equal to 4, as we consider deeper decision trees of more interest. If optimality could not be proven within 10 minutes, this is indicated using *TO*; in this case, the objective value of the best tree found so far is shown. Note that we here exploit the ability of DL8.5 to produce a result after a time-out. The best solutions and best times are marked in bold while a star (*) is added to mark solutions proven to be optimal.

BinOCT solved and proved optimality for only 1 instance within the timeout; the older DL8 algorithm solved 7 instances and the CP-based algorithm solved 11 instances. DL8.5 solved 19 (which answers **Q1**). The difference in performance is further illustrated in Figure 4, which gives *cactus plots* for each algorithm, for different depth constraints. In these plots each point (x, y) indicates the number of instances (x) solved within a time limit (y). While for lower depth thresholds, BinOCT does find solutions, the performance of all variants of DL8.5 clearly remains superior to

[1] https://github.com/SiccoVerwer/binoct

[2] https://bitbucket.org/helene_verhaeghe/classificationtree/src/default/classificationtree/

[3] https://www.ibm.com/analytics/cplex-optimizer

[4] https://oscarlib.bitbucket.io

[5] https://dtai.cs.kuleuven.be/CP4IM/datasets/

tha*TO*f DL8, BinOCT and the CP-based algorithm, obtaining orders of magnitude better performance.

Comparing the different branching heuristics in DL8.5, the differences are relatively small; however, for deeper trees, a descending order of information gain gives slightly better results. This confirms the intuition that chosen a split with high information gain is a good heuristic (**Q2**).

If we disable DL8.5's ability to cache incomplete results, we see a significant degradation in performance. In this varian*TO*nly 12 instances are solved optimally, instead of 19, for a depth of 4. Hence, this optimization is significant (**Q3**).

To answer **Q4**, we repeat these tests on continuous data. For this, we use the same datasets as Verwer and Zhang (2019). These datasets were obtained from the UCI repository[6] and are summarized in the first columns of Table 3. Before running DL8, the CP-based algorithm and DL8.5, we binarize these datasets by creating binary features using the same approach as the one used by (Verwer and Zhang 2019). Note that the number of generated features is very high in this case. As a result, for most datasets all algorithms reach a time-out for maximum depths of 3 and 4, as was also shown by Verwer and Zhang (2019). Hence, we focus on results for a depth of 2 in Table 3. Even though BinOCT uses a specialized technique for solving continuous data, the table shows that DL8.5 outperforms DL8, the CP-based algorithm and BinOCT. Note that the differences between the different variations of DL8.5 are small here, which may not be surprising given the shallowness of the search tree considered.

Conclusions

In this paper we presented the DL8.5 algorithm for learning optimal decision trees. DL8.5 is based on a number of ideas: the use of itemsets to represent paths, the use of a cache to store intermediate results (including results for parts ofthe search tree that have only been traversed partially), theuse of bounds to prune the search space, the ability to useheuristics during the search, and the ability to return a resulteven when a time-out is reached.

Our experiments demonstrated that DL8.5 outperforms existing approaches by orders of magnitude, including approaches presented recently at prominent venues.

In this paper, we focused our experiments on one particular setting: learning maximally accurate trees of limited depth without support constraints. This was motivated by our desire to compare our new approach with other approaches. However, we believe DL8.5 can be modified for use in other constraint-based decision tree learning problems, using ideas from DL8 (Nijssen and Fromont 2010).

Acknowledgements. This work was supported by Bpost.

References

Aghaei, S.; Azizi, M. J.; and Vayanos, P. 2019. Learning optimal and fair decision trees for non-discriminative decision-making. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 1418–1426.

Agrawal, R.; Mehta, M.; Shafer, J. C.; Srikant, R.; Arning, A.; and Bollinger, T. 1996. The quest data mining system. In *KDD*, volume 96, 244–249.

Bertsimas, D., and Dunn, J. 2017. Optimal classification trees. *Machine Learning* 106(7):1039–1082.

Bessiere, C.; Hebrard, E.; and O'Sullivan, B. 2009. Minimising decision tree size as combinatorial optimisation. In *International Conference on Principles and Practice of Constraint Programming*, 173–187. Springer.

Breiman, L.; Friedman, J.; Olshen, R.; and Stone, C. 1984. Cart. *Classification and Regression Trees*.

Laurent, H., and Rivest, R. L. 1976. Constructing optimal binary decision trees is np-complete. *Information processing letters* 5(1):15–17.

Narodytska, N.; Kasiviswanathan, S.; Ryzhyk, L.; Sagiv, M.; and Walsh, T. 2018. Verifying properties of binarized deep neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Nijssen, S., and Fromont, E. 2007. Mining optimal decision trees from itemset lattices. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 530–539.

Nijssen, S., and Fromont, E. 2010. Optimal constraint-based decision tree induction from itemset lattices. *Data Mining and Knowledge Discovery* 21(1):9–51.

Quinlan, J. R. 1993. C4. 5: Programming for machine learning. *Morgan Kauffmann* 38(48):49.

Verwer, S., and Zhang, Y. 2019. Learning optimal classification trees using a binary linear program formulation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, 1625–1632.

[6] https://archive.ics.uci.edu/ml/index.php

Dataset	nItems	nTrans	BinOCT		DL8		CP-Based		DL8.5							
									d.o.		asc		desc		n.p.s.	
			obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)
anneal	186	812	115	*TO*	∞	*TO*	**91***	**450.69**	**91***	129.24	**91***	127.45	**91***	121.87	91*	250.64
audiology	296	216	2	*TO*	∞	*TO*	1	*TO*	1*	180.84	1*	204.19	1*	195.73	1	*TO*
australian-credit	250	653	82	*TO*	∞	*TO*	66	*TO*	**56***	566.71	**56***	586.39	**56***	593.38	57	*TO*
breast-wisconsin	240	683	12	*TO*	∞	*TO*	8	*TO*	**7***	305.3	**7***	325.7	**7***	330.71	7	*TO*
diabetes	224	768	170	*TO*	∞	*TO*	140	*TO*	**137***	553.49	**137***	562.83	**137***	565.5	137	*TO*
german-credit	224	1000	223	*TO*	∞	*TO*	204	*TO*	**204***	558.73	206	*TO*	**204***	599.87	204	*TO*
heart-cleveland	190	296	39	*TO*	∞	*TO*	25	*TO*	**25***	124.1	**25***	130.3	**25***	132.23	**25***	214.76
hepatitis	136	137	7	*TO*	**3***	66.62	**3***	109.36	**3***	13.46	**3***	14.06	**3***	14.88	**3***	27.28
hypothyroid	176	3247	55	*TO*	∞	*TO*	53	*TO*	**53***	392.22	**53***	368.95	**53***	427.34	53	*TO*
ionosphere	890	351	27	*TO*	∞	*TO*	20	*TO*	17	*TO*	11	*TO*	13	*TO*	17	*TO*
kr-vs-kp	146	3196	193	*TO*	∞	*TO*	**144***	483.15	**144***	216.11	**144***	206.18	**144***	223.85	**144***	528.72
letter	448	20000	813	*TO*	∞	*TO*	574	*TO*	550	*TO*	586	*TO*	802	*TO*	550	*TO*
lymph	136	148	6	*TO*	**3***	56.29	**3***	112.48	**3***	8.7	**3***	11.03	**3***	8.47	**3***	25.04
mushroom	238	8124	278	*TO*	∞	*TO*	**0***	352.18	**0***	331.39	4	*TO*	**0***	0.11	4	*TO*
pendigits	432	7494	780	*TO*	∞	*TO*	38	*TO*	32	*TO*	26	*TO*	14	*TO*	32	*TO*
primary-tumor	62	336	37	*TO*	**34***	2.79	**34***	8.96	**34***	1.48	**34***	1.51	34*	1.38	**34***	2.43
segment	470	2310	13	*TO*	∞	*TO*	**0***	128.25	**0***	3.54	**0***	6.99	**0***	7.05	**0***	3.54
soybean	100	630	15	*TO*	**14***	41.59	**14***	40.13	**14***	5.7	**14***	6.34	**14***	5.75	**14***	18.41
splice-1	574	319	574	*TO*	∞	*TO*	∞	*TO*	224	*TO*	224	*TO*	141	*TO*	224	*TO*
tic-tac-toe	54	958	180	*TO*	**137***	3.76	**137***	9.17	**137***	1.43	**137***	1.54	**137***	1.55	**137***	2.12
vehicle	504	846	61	*TO*	∞	*TO*	22	*TO*	16	*TO*	18	*TO*	13	*TO*	16	*TO*
vote	96	435	6	*TO*	**5***	29.84	**5***	44.47	**5***	5.48	**5***	5.33	**5***	5.58	**5***	12.82
yeast	178	1484	395	*TO*	∞	*TO*	366	*TO*	**366***	318.87	**366***	326.2	**366***	334.15	**366***	470.88
zoo-1	72	101	**0***	0.52	**0***	1.11	**0***	0.2	**0***	0.01	**0***	0.01	**0***	0.01	**0***	0.01

Table 2: : Comparison table for binary datasets with max depth = 4

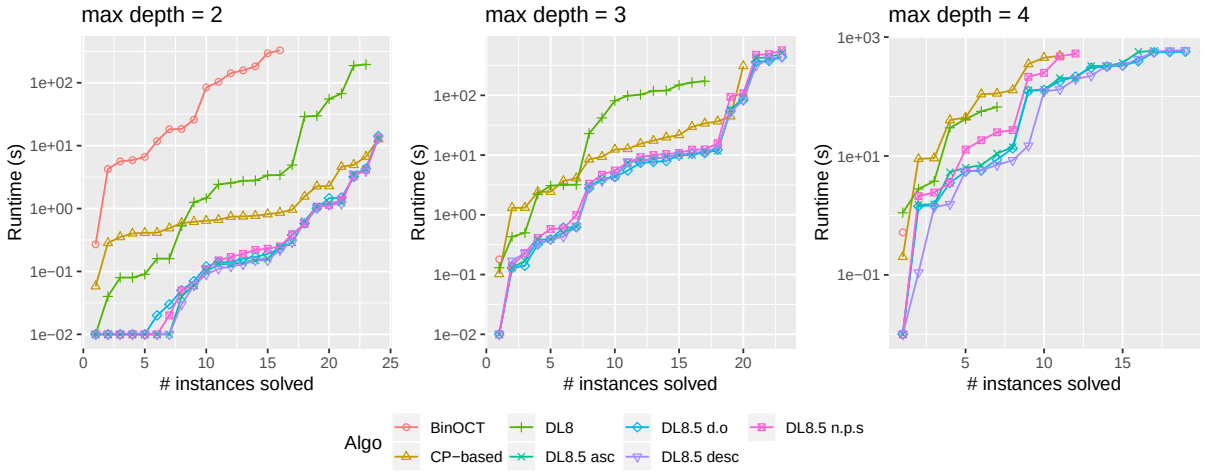

Figure 4: Cumulative number of instances solved over time

Dataset	nTrans	nFeat	nItems	BinOCT		DL8		CP-Based		DL8.5					
										d.o.		asc		desc	
				obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)
balance-scale	625	4	32	**149***	1.2	**149***	0.5	**149***	0.01	**149***	0.01	**149***	0.01	**149***	0.01
banknote	1372	4	3710	101	*TO*	**100***	363.01	∞	*TO*	**100***	52.81	**100***	63.41	**100***	58.07
bank	4521	51	3380	449	*TO*	448	*TO*	∞	*TO*	**446***	253.87	**446***	223.42	**446***	222.66
biodeg	1055	41	8356	212	*TO*	212	*TO*	∞	*TO*	**202***	341.57	**202***	365.83	**202***	370.26
car	1728	6	28	**250***	4.09	**250***	0.32	**250***	0.02	**250***	0.01	**250***	0.01	**250***	0.01
IndiansDiabetes	768	8	1714	171	*TO*	**171***	36.75	∞	*TO*	**171***	7.43	**171***	8.46	**171***	8.72
Ionosphere	351	34	4624	29	*TO*	**29***	423.52	∞	*TO*	**29***	25.68	**29***	33.76	**29***	33.04
iris	150	4	52	**0***	0.02	**0***	0.05	**0***	0.01	**0***	0.01	**0***	0.01	**0***	0.01
letter	20000	16	352	625	*TO*	**591***	5.97	**591***	392.09	**591***	8.61	**591***	8.26	**591***	8.83
messidor	1151	19	9460	383	*TO*	383	*TO*	∞	*TO*	383*	533.27	383*	563.83	383*	534.32
monk1	124	6	22	22*	0.33	22*	0.28	22*	0.01	22*	0.01	22*	0.01	22*	0.01
monk2	169	6	22	57*	0.79	57*	0.28	57*	0.01	57*	0.01	57*	0.01	57*	0.01
monk3	122	6	22	8*	0.31	8*	0.28	8*	0.01	8*	0.01	8*	0.01	8*	0.01
seismic	2584	18	2240	166	*TO*	164*	117.34	∞	*TO*	164*	44.3	164*	47.36	164*	47.17
spambase	4601	57	16012	660	*TO*	900	*TO*	∞	*TO*	741	*TO*	845	*TO*	586	*TO*
Statlog	4435	36	3274	460	*TO*	443	*TO*	∞	*TO*	443*	205.14	443*	193.87	443*	188.9
tic-tac-toe	958	18	36	282*	7.52	282*	0.33	282*	0.01	282*	0.01	282*	0.01	282*	0.01
wine	178	13	1198	6*	73.1	6*	7.0	6*	74.72	6*	1.17	6*	1.45	6*	1.09

Table 3: Comparison table for continuous datasets with max depth = 2