Optimal Decision Trees Under Constraints

Frédéric Don-de-Dieu Gaël Roméo Aglin

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

October 2022

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Optimal Decision Trees Under Constraints
by Frédéric Don-de-Dieu Gaël Roméo Aglin

ICTEAM
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work was partially supported by bpost.

The possible solutions to a given problem emerge as the leaves of a tree, each node representing a point of deliberation and decision.

NIKLAUS WIRTH

Preamble

Machine Learning (ML) is an Artificial Intelligence (AI) subfield in which computer programs can learn to perform a task without being explicitly programmed. It relies on the analysis of past event scenarios to predict the future. Machine learning produces impressive results even for complex tasks. It has become an integral part of our lives and can even solve problems that are difficult for humans. However, the significant presence of machine learning models in our daily lives can have some drawbacks. Obviously, it is not possible to design infallible models. There are several reasons for this, including the following: (1) the high complexity of real-life problems leading to (2) the difficulty of collecting training data that are perfectly representative of the task at hand and (3) the sensitivity of the models to certain types of noise. These obvious problems can cause models to produce bad results in sensitive areas involving human life. Thus, in some critical areas, it may be dangerous to rely completely on machine learning predictions.

The ideal in machine learning would be to learn models that never make mistakes. Unfortunately, this is impossible for the above reasons. A good compromise would be to learn interpretable models. These are models that can explain the elements that underlie their results. In this case, humans would be able to understand them and judge for themselves whether or not to use them. Such models already exist. One of the most popular ones that we have studied during this thesis are Decision Trees (DTs). An example of a decision tree is shown in Figure 1.

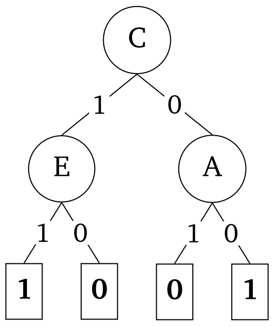


Figure 1: Decision tree with 3 internal nodes, 4 leaf nodes, depth = 2 and size = 7

The principle of a decision tree is to generate, through its paths, a set of rules based on the features of training data. Each rule describes a set of instances in the training data, and the tree associates a prediction with these instances. Therefore, decision trees are perfect examples of interpretable models. It is natural and intuitive to understand the predictions of a decision tree. Unfortunately, the crucial problem with interpretable models and thus with decision trees is their performance. Although decision trees offer reasonable performance, they sometimes perform poorly on complex problems. Traditional algorithms [Bre+84; Qui86; Qui93] for learning decision trees are heuristic and often produce a high error rate when they are limited by a maximum depth. They infer decision trees in a top-down fashion, iteratively dividing the data into subsets using heuristic criteria to choose the "best" splits. In order to reduce the error in such trees, deep trees must be generated, and in this case, they become complex and less interpretable. A possible solution to this problem, which we have investigated in this thesis, is Optimal Decision Trees (ODTs) [BB96]. More specifically, we have studied optimal binary decision trees. These are binary decision trees that allow to reduce as much as possible the error that could be produced for a given depth. The error reduction proposed by the optimal decision trees allows the building of better performing trees with lower depths, thus being also more interpretable. This was shown in 2017 by Bertsimas and Dunn [BD17]. Since then, several new approaches have been proposed to learn optimal binary trees. However, Laurent and Rivest proved in 1976 that finding an optimal binary tree under a size constraint is an NP-complete problem. This explains why most of the proposed algorithms struggle to learn optimal decision trees in a reasonable time. This thesis describes our contribution to facilitate the learning of optimal binary trees.

Part I of this thesis is divided into 3 chapters that present the important theories underlying the different algorithms that we propose in the following. Chapter 1 further defines what a decision tree is and presents two (2) well-known traditional algorithms (CART, C4.5) for learning decision trees. Chapter 2 presents a popular concept in data mining, Frequent Itemset Mining (FIM), that is used in the algorithms we propose in this thesis. Finally, Chapter 3 introduces an important data structure that is also used in our algorithms.

Part II of the thesis shows the main approaches proposed to efficiently learn optimal decision trees. Chapter 4 presents an efficient algorithm for learning optimal binary classification trees under a support constraint. This algorithm is a search algorithm combined with dynamic programming and the branch-and-bound concept. Its execution time outperforms its predecessors by several orders of magnitude. One of the important elements that allows the proposed algorithm to achieve record execution times is the use of a cache structure, due to its dynamic programming aspect. Unfortunately, this also represents its weakness, since the number of elements to store when learning optimal trees is exponential. In Chapter 5, we propose a memory cleaning algorithm that we embed into the algorithm proposed in Chapter 4. This algorithm acts as a garbage collector that takes care of automatically removing from the cache, after a certain limit, some stored elements that at the wipe time are not considered as part of the final solution and that are less likely to be useful further in the search.

Part III of this thesis is devoted to the different applications that our optimal tree learning algorithm could have. In Chapter 6, the basic algorithm proposed in Chapter 4 is modified to handle the training data with weighted instances. We show that this new algorithm is capable of solving problems that were impossible to solve beforehand. In the case of our thesis, we use it to solve the problem of optimal forests. With this new algorithm, we were able to solve existing mathematical models of optimal forests that could not be solved to optimality beforehand. This allows one to evaluate the relevance of these models that were intended to prove the intuition of the success of the boosting algorithms. In Chapter 7, we show that our algorithm is suitable for solving several other machine learning tasks and optimizing other objective functions besides those of Chapter 4. We then define the type of problems that can be solved with our algorithm and show how user-specific problems can be defined. Finally, Chapter 8 concludes this thesis.

All algorithms proposed in this thesis are made available through an open source library available at https://github.com/aia-uclouvain/pydl8.5. It is compatible with the scikit-learn[1] library and can be installed from PyPI[2] with the command `pip install pydl8.5`. The documentation is available at https://pydl85.readthedocs.io/en/latest/.

[1]https://scikit-learn.org/
[2]https://pypi.org

Bibliographic notes

This thesis led to the publication of, and is based on, the following works:

1. G. Aglin, S. Nijssen, and P. Schaus. “Learning optimal decision trees using caching branch-and-bound search”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 3146–3153.
2. G. Aglin, S. Nijssen, and P. Schaus. “Learning Optimal Decision Trees Under Memory Constraints”. *The paper is accepted at the joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD) 2022 but is not yet published at the moment this thesis is written.*
3. G. Aglin, S. Nijssen, and P. Schaus. “Assessing Optimal Forests of Decision Trees”. In: *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE Computer Society. 2021, pp. 32–39.
4. G. Aglin, S. Nijssen, and P. Schaus. “Évaluation des forêts optimales d’arbres de décision”. *The paper is accepted at Journées Francophones de Programmation par Contraintes (JFPC) 2022 but is not yet published at the moment this thesis is written.* ***(Summary paper)***
5. G. Aglin, S. Nijssen, and P. Schaus. “PyDL8.5: a library for learning optimal decision trees”. In: *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 2021, pp. 5222–5224. ***(Demo paper)***

Acknowledgments

With these words, I would like to thank all the people who have contributed to the achievement of this thesis. To all of you, mentioned or omitted in this text, I express my deep gratitude.

My first thanks go especially to my advisors Pierre Schaus and Siegfried Nijssen. Thank you for having trusted me without knowing me and for having accepted to guide me and collaborate with me until I obtained this degree. Learning from you has been a real pleasure. Thank you for all the scientific discussions that we had and that led to this thesis. To you Pierre, thank you for teaching me the work discipline necessary to any researcher. Thanks to you, I have learned the meaning of provable work and reproducible research. To you Siegfried, thank you for all the discussions that allowed me to improve the quality of my algorithms. Thank you especially for the proofreading and correction of papers sometimes late at night. This has greatly improved my writing skills. To both of you, thank you especially for your time. It will be a pleasure to continue collaborating with you.

Secondly, I would like to thank all the people I have met in my research department since the beginning of this adventure. I want to thank especially Vincent, Son, Hélène, Vianney and Nicolas with whom I shared my office during these years. Thanks for the friendly atmosphere, even if we were not very talkative. I also want to thank François, Mickael, Tanh, Raziel and Olivier for all the friendly moments spent together. Special mention for Vanessa and Sophie who made my life easier many times when I tried to complicate it.

Then, I want to thank all the people who believed more than me in this project and who encouraged me after my Master. I want to name Christiane, Princia, Sonia and Arlette. More than friends, you are sisters. Thank you also to my friend Fabrice with whom I had been nurturing this dream for a long time. A big thank goes to Ratheil whose meeting triggered the realization of this project. Thank you for believing in me.

It is now the place to thank all the other Beninese researchers with whom I spent a small or a large part of my time in my department or in Louvain-la-Neuve: John, Lionel, Parfait, Mêton-Mêton, Emery, Steaven, Modeste, Harold. Thank you guys for all the moments of conviviality and fraternity. The adventure was even more beautiful at your side.

To all my friends who, from near or far, supported me in this project: Crispin, Christophe, Samson, Anne, Thècle, Claudia, Erika, Sandra, I say thank you. I especially thank my friend Doriane. Thank you for all that you brought to me. I know that you are proud of this work. Find in this the sign of my friendship and my gratitude.

To my beloved Francine, thank you for your support and your daily presence in my life. You have lived my joys as well as my moments of doubt. This work is also the fruit of your efforts. Find in this our common accomplishment.

Finally, I would like to thank my family in particular. Of course, you are far away from me, but without your trust, your encouragement, your support and your prayers, I would have had difficulty in accomplishing this project. I have only had the opportunity to see you twice since the beginning of this thesis. Please find in this work, the fruit of my absence and your many calls and messages without replies. I especially thank my father Frédéric and my mother Philomène for their love and their advice. I also thank my sisters Hachette, Audrey and Ghislaine as well as my nephews and niece Oxyl, Yahn, Eddy, Georgine, Jordy and Kylian for their love and their healing presence in my life.

Gaël Aglin, October 2022.

Contents

Part I

Background

Decision Trees 1

1.1 Introduction

Decision trees are basically visual and intuitive flowcharts that help to make decisions. They describe a process to perform prediction tasks. Figure 1.1 shows an illustrative decision tree that can identify whether an animal is a bird. Note that a decision tree is built on the elements that describe the problem to which it applies. In Figure 1.1, these elements are the physical aspects of an animal. As can be seen, a decision tree is made up of nodes that test certain descriptive elements that are called *tests*. From these nodes, branches emerge that suggest different values or ranges of values for the tests. These nodes are *internal nodes*. An example of an internal node in Figure 1.1 is the root node, testing the *Height* of the animal. At the end of each path of the tree, there is a terminal node that contains one of the decisions to make. They are called *leaf nodes*. They are represented by rectangular nodes in the figure. The association of the internal nodes with their branches is called *splits*.

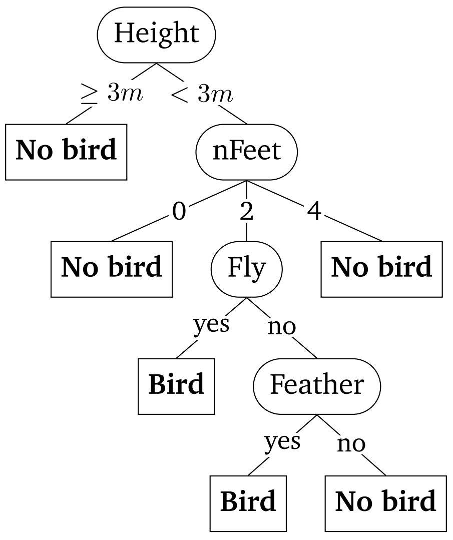


Figure 1.1: Decision tree to predict if an animal is a bird

When a decision tree is built, it is used to associate decisions with some objects whose decisions were unavailable during the training process. For each object, depending on the value of the elements tested, a path is followed through the branches until a leaf node is encountered. The leaf node provides the decision suggested by the tree for the object. For example, the tree in Figure 1.1 states that an animal with height equal to 5 meters is not a bird, since its height is greater than or equal to 3 meters.

In the context of machine learning, a decision tree is a great diagram that is used as a model to predict the targets of unseen data, based on training data. In concrete terms, an algorithm is run on previous observations and produces a decision tree based on these observations. Each internal node in the decision tree selects a feature on which a value check should be performed. The branches rising from the internal nodes are related to possible values or ranges of values for the tested features, while the predictions are associated with leaf nodes. To predict the target of an unseen instance, the instance is passed through the tree, from the root node to a leaf node, guided by the branches representing its values for each tested feature. The choice of the splits that form the tree as well as the predictions associated with leaf nodes by the algorithm are crucial in order to make predictions close to the reality. In order to learn accurate decision trees, the well-known algorithms rely on a top-down process using heuristics to determine the best splits and a specific function to choose the predictions at the leaf nodes.

In the remainder of this chapter, we will present in Section 1.2 the major notations that will be used in the thesis. Section 1.3 presents the top-down procedure used in the well-known classification tree learning algorithms, while highlighting the specificity of CART and C4.5. In Section 1.4, we present other kinds of problems that can be solved using decision trees.

1.2 Notation

In the remainder of this thesis, we will use numerous notations that need to be introduced. Here, we will define the ones that will be frequently encountered. The specific ones will be defined when they are needed.

A dataset is denoted $\mathcal{D}$ and is represented by $\mathcal{D} = \{(\boldsymbol{x}, y)\}^n$, with n being the number of instances. Each instance is denoted as a pair $(\boldsymbol{x}_i, y_i)$. $\boldsymbol{x}_i = (x_{i1}, \dots, x_{im})$ is described by a set $\mathcal{F} = \{F_1, \dots, F_m\}$ of m features. The features are also called *attributes*. Each value y_i is the target associated with $\boldsymbol{x_i}$. In classification problems, the targets are finite. In this case, the targets are also called *labels* or *classes*. We thus denote $\mathcal{C}$ the set of classes involved in the dataset $\mathcal{D}$.

Each model in this thesis can be represented by a hypothesis $h(\boldsymbol{x})$ and $\mathcal{H}$ is defined to be the set of all possible models that can be produced given a problem.

1.3 Top-down learning of decision trees

In the context of machine learning, the aim is to build models able to predict targets close to reality, based on previous observations. More specifically, in supervised learning, the previous observations are made up of feature vectors describing the instances, associated to targets to predict. The task is then to learn from these observations to build models able to predict the targets associated with unseen instances based only on their feature vectors.

Given a dataset $\mathcal{D} = \{(\boldsymbol{x}, y)\}^n$, the generic procedure used by the well-known top-down algorithms to learn classification trees is presented in the pseudocode of the Algorithm 1. Note that the CART and C4.5 algorithms notably follow this procedure.

Algorithm 1: Generic top-down learning decision tree

```
input : $\boldsymbol{x}^n$, a set of feature vectors
input : $y^n$, a set of targets associated with $\boldsymbol{x}^n$
1  $feat\_splits \leftarrow$ splitsPerFeature($\boldsymbol{x}^n$)
2  return buildSubtree($\boldsymbol{x}^n, y^n, feat\_splits$)
3  Procedure buildSubtree($\boldsymbol{x}'^m, y'^m, splits$)
4      if stoppingConditions($\boldsymbol{x}'^m, splits$) then
5          return buildLeafNode(prediction($y'^m$))
6      $best\_split \leftarrow$ selectBestSplit($splits$)
7      $node \leftarrow$ buildInternalNode($best\_split$)
8      $remain\_splits \leftarrow splits \setminus best\_split$
9      foreach subset $\boldsymbol{x}^{*k}, y^{*k}$ fallen in each branch of $best\_split$ do
10         $node$.addChild(buildSubtree($\boldsymbol{x}^{*k}, y^{*k}, remain\_splits$))
11     return $node$
```

The algorithm takes as input the training dataset, represented by the feature matrix and the target vector. The idea of the algorithm is to iteratively choose the different splits that will constitute the final decision tree. For this reason, the different possible splits are enumerated per feature, based on the different feature vectors (line 1). The splits are considered differently given the learning algorithm. This will be explained in more details later. Once the splits are enumerated, the best is chosen to be used as the root node. Then, iteratively, the splits that are used as the children of the considered nodes are identified. This is highlighted in the algorithm as a recursive process represented by the function `buildSubtree()`. This function is responsible for selecting the good split at a certain point in the decision tree building process (lines 6-7).

There exist multiple criteria to select the best split. We will discuss them further below. Furthermore, the `buildSubtree()` function recursively launches the subroutine to identify child splits (line 10). Notice that the function `buildSubtree()` receives as arguments the feature matrix, the target vector, and the available splits. These parameters have not the same values at each call to the function `buildSubtree()`. Based on the different branches of the best split selected, the original dataset is refined to keep only the subset that satisfies the feature value required by the selected split (line 9). Moreover, the splits passed to each call of the `buildSubtree()` function are updated in order to remove the splits that have already been chosen. This prevents us from considering more than once a split in a decision path (line 8). When building the decision tree, if for any reason represented by the function `stoppingConditions()`, a split cannot be chosen, a leaf node is built based on the targets of the instances falling in the current path, and a class is predicted using the function `prediction()`. Generally, in classification tasks, the predicted class is the majority class. More formally, given a data subset $\mathcal{D}' = \{(\boldsymbol{x}', y')\}^n$ and a set $\mathcal{C}$ of values in y'^n, the predicted class is:

$$\operatorname*{argmax}_{c \in \mathcal{C}} \quad |\{(\boldsymbol{x}', y') \in \mathcal{D}' : y' = c\}| \tag{1.1}$$

1.3.1 Splits assessment

When learning decision trees, the idea is to infer the best tree so that the predictions in the leafs match the best with the real decisions. Based on a training dataset, the DT learning algorithms, like other machine learning algorithms, infer models that reflect at a certain point the labels in the training dataset. The labels are then used to predict on unseen data. The process of building a decision tree can be seen as a process to accurately identify the relevant node splits that are part of the tree. Each algorithm defines the splits in its own way.

A split is a feature associated with some values or range of values. In a decision tree, splits occur in nodes and define which branches arise from the node to the child nodes. The main differences observed in the splits considered in DT learning algorithms is about the number of branches in which the feature values are divided. This depends on the type of the feature, and mostly on the type of the DT inferred by the algorithm. Some algorithms infer binary decision trees, while others infer multiway trees. The splits in binary decision trees result in nodes splitting into two features values or ranges of values. The introductory decision tree depicted in Figure 1.1 shows three binary splits. The root node shows a split on the feature *Height*, divided into two ranges of values, and the deepest test node shows a split on the feature *Feather* into two values (yes/no). On the contrary, a multiway tree uses splits related to more than 2 branches. The node involving the feature *nFeet* shows a split into three values $\{0, 2, 4\}$.

Binary features These features contain only two values; generally 0/1, yes/no, or true/false. It can also be categorical features with only two possible values. It is intuitive to split these features into two branches. Therefore, only binary splits are considered for them.

Categorical features These are features which contain multiple (more than 2) values, and there is no order relation between these values. Let us assume a feature with k distinct values. In a multiway decision tree, the split considered by existing algorithms involves dividing the internal nodes into k branches, one per feature value. On the other hand, in the context of binary decision trees, the feature should be broken down into several binary splits. Each binary split involves a set of at most $k - 1$ possible values for a branch, while the second is related to values that are not involved in the first branch. The maximum number of possible splits is $2^{k-1} - 1$.

Numerical features These features contain multiple values with an order relation between them. In case of a numerical feature F with a finite set of k distinct values, multiway decision tree learning algorithms consider a split with k branches as for categorical features. This contrasts with binary decision tree learning algorithms that consider $k - 1$ binary splits. Each split acts as an interval cut between the k values. Each split results in two branches $\{feature \leq a, feature > a\}$, with $a \in F$.

Note that the CART [Bre+84] algorithm produces binary trees because these are more interpretable than multiway ones in which the number of branches can be huge, given the distribution of values in feature vectors. On the other hand, C4.5 [Qui93], which is an improvement of ID3 [Qui86] proposed by the same author (Quinlan), in its default mode, uses binary splits for numerical features and multiway splits for categorical features. The splitting is inherited from ID3. After enumerating the possible splits, the learning algorithms choose at each step of the algorithm, the best split to consider as node. Note that the choice of the best split depends mainly on the problem being solved. In the case of classification, the goal of learning algorithms is to minimize the misclassification error on the training dataset. This error is seen as the number of training instances incorrectly classified by the learned tree. As a decision tree splits the datasets into multiple subsets falling in leaf nodes, the misclassification error of a tree is the sum of the misclassification errors over the different leafs. Given a path p ending in a leaf ℓ in a decision tree $\mathcal{DT}$ which covers a data subset D_ℓ, the misclassification error is denoted $leaf\_error(\ell)$, $leaf\_error(D_\ell)$, or $leaf\_error(p)$ and is expressed as:

$$|D_\ell| - \max_{c \in \mathcal{C}} |\{(\boldsymbol{x}, y) \in D_\ell : y = c\}|, \tag{1.2}$$

where $\mathcal{C}$ is the set of classes in the target of the dataset. Therefore, the misclassification error in the whole tree is:

$$tree\_error(\mathcal{DT}) = \sum_{\ell \in leafs(\mathcal{DT})} leaf\_error(D_\ell), \tag{1.3}$$

where $leafs(\mathcal{DT})$ is the set of leafs in the tree $\mathcal{DT}$. This error measures the amount of heterogeneity of the different classes in the leafs of the decision tree. To learn trees that produce a low misclassification error, the traditional algorithms use a greedy approach. They all define an evaluation criterion that they use as a heuristic to decide at each step of the building process the best split to choose in order to build a tree that reduces the misclassification rate on the training data. Several criteria have been defined in the literature. In this thesis, we will present only the most popular ones. These are the Gini criterion and the Gain Ratio, respectively, proposed by CART and C4.5. They aim at finding the split that increases the homogeneity in class distributions according to different data subsets that arrive from the parent node.

Gini criterion or Gini index Assume a dataset $\mathcal{D}$ is splittable into a set S of splits. A Gini criterion is expressed as:

$$Gini(\mathcal{D}) = \sum_{c \in \mathcal{C}} p_c(1 - p_c) = 1 - \sum_{c \in \mathcal{C}} p_c^2, \tag{1.4}$$

where p_c is the proportion of class c in the target vector of $\mathcal{D}$. This value expresses the heterogeneity amount of classes in $\mathcal{D}$. As $\mathcal{D}$ can be divided into different splits, for each split S_i, the dataset is split into k subsets $S_{i1}, \ldots, S_{ik}$ respectively associated with data subsets $\mathcal{D}_{S_{i1}}, \ldots, \mathcal{D}_{S_{ik}}$. A Gini criterion is then computed for each split S_i in order to pick the one that decreases the most the heterogeneity in the class distributions of

$\mathcal{D}$. This is performed by computing the following gap value for each possible split S_i:

$$\begin{aligned}\Delta Gini(S_i|\mathcal{D}) &= Gini(\mathcal{D}) - \sum_{S_{ik}\in S_i} \frac{|\mathcal{D}_{S_{ik}}|}{|\mathcal{D}|} Gini(\mathcal{D}_{S_{ik}}) \\ &= Gini(\mathcal{D}) - Gini(\mathcal{D}_{S_i})\end{aligned} \tag{1.5}$$

The first term of this formula computes the Gini criterion for the dataset $\mathcal{D}$ while the second computes the Gini criterion of $\mathcal{D}$ split into S_i. This last term is computed by summing the Gini criterion of each data subset falling in each branch of the split S_i, weighted by its proportion in $\mathcal{D}$.

In case of binary splits, a simplified Gini criterion is presented in CART [Bre+84] and is called *Twoing criterion*. It is expressed as:

$$\Delta Twoing(S_i|\mathcal{D}) = \frac{p_0 \cdot p_1}{4} (\sum_{c\in\mathcal{C}} |p_{0c} - p_{1c}|)^2, \tag{1.6}$$

where $p_k = |\mathcal{D}_{S_{ik}}|/|\mathcal{D}|$ is the proportion of data in the branch k of the split S_i of the dataset $\mathcal{D}$. $p_{kc} = |\{\boldsymbol{x}, y \in \mathcal{D}_{S_{ik}} : y = c\}|/|\mathcal{D}_{S_{ik}}|$ is the proportion of class c in the data subset $\mathcal{D}_{S_{ik}}$.

Information Gain (IG) and Gain Ratio C4.5 uses the Gain Ratio criterion to compare and choose the best split. This criterion relies on the Information Gain criterion, which was introduced by ID3. The idea of the Information Gain criterion is based on Shannon entropy [Sha48], which is defined as the degree of uncertainty of a random variable. Its value is minimal in a distribution made up of a unique value. On the other hand, it reaches its highest value when the variable distribution contains an equal proportion of distinct values. This metric can then be used to quantify the homogeneity in a distribution. The entropy of a dataset $\mathcal{D}$ is computed as:

$$Entropy(\mathcal{D}) = -\sum_{c\in\mathcal{C}} p_c \log p_c \tag{1.7}$$

Similarly to the Gini index, the entropy is computed for each split S_i and the one that most significantly decreases the heterogeneity of the class distribution in $\mathcal{D}$ is chosen. The expression that computes the reduction of heterogeneity is called Information Gain and is expressed as:

$$\Delta InfoGain(S_i|\mathcal{D}) = Entropy(\mathcal{D}) - \sum_{S_{ik} \in S_i} \frac{|\mathcal{D}_{S_{ik}}|}{|\mathcal{D}|} \sum_{c \in \mathcal{C}} Entropy(\mathcal{D}_{S_{ik}}) \quad (1.8)$$

Although the use of Information Gain produces good results in practice, Quinlan noticed that the IG measure is biased towards multiway splits made up of a large number of branches. For this reason, he proposed a normalized version of IG called the Gain Ratio and used it in the C4.5 algorithm. The idea is to divide the IG of each split by a value called *split information* and expressed as:

$$splitInfo(S_i) = - \sum_{S_{ik} \in S_i} \frac{|\mathcal{D}_{S_{ik}}|}{|\mathcal{D}_{S_i}|} \log \frac{|\mathcal{D}_{S_{ik}}|}{|\mathcal{D}_{S_i}|} \quad (1.9)$$

This expression is an entropy computed on the proportions of the data subsets in each split. Then it is used to normalize the IG. Therefore, the gain ratio of a split S_i of a dataset $\mathcal{D}$ is expressed as:

$$\Delta GainRatio(S_i) = \frac{\Delta InfoGain(S_i|\mathcal{D})}{splitInfo(S_i)} \quad (1.10)$$

1.3.2 Stopping conditions

During the learning of a decision tree, the algorithms select at each step of the process the best split to use as node. Sometimes, it appears that there is no need to further split the data subset that arrives from the parent node. These cases are stopping conditions of the recursive call of `selectBestSplit()` in Algorithm 1. They prevent to call the recursive subroutine which is in charge of finding the best split. In this case, a leaf node is created, and a class is associated according to the equation 1.1. An obvious stopping condition is the case in which all instances arriving from the parent node belong to the same class. Indeed, the idea of a decision tree is to split the dataset into homogeneous groups in order to predict a class without ambiguity. Therefore, when all the instances of a group belong to the same class, there is no need to split them more. However, the use of this only stopping condition would lead to very large trees induction. After trying different stopping conditions which caused overpruning, the only one used by CART and C4.5, in addition to the class purity, is the fact that each split would lead to at least 2 branches with a minimum number of instances falling into each branch.

In addition to default CART and C4.5 stopping conditions, many implementations of these algorithms add a stopping condition related to a maximum depth specified as a hyper-parameter.

As these default stopping conditions of CART and C4.5 still lead to large trees, they suffer from overfitting. Their authors proposed post-pruning techniques to mitigate the overfitting issue.

1.3.3 Overfitting mitigation

After building a decision tree with soft stopping conditions, the tree might be very large and might overfit the data. The problem of such overfitted trees is double:

- the final tree is not interpretable,
- the tree perform poorly on unseen data.

The authors of well-known DT learning algorithms noticed that large trees do not generally provide a great increase of accuracy compared to shallow ones, while shallow trees prevent from overfitting. They thus proposed different approaches to post-prune the trees in order to make them shallow without sacrificing too much accuracy. To do so, most of them use a bottom-up approach in which they evaluate the impact of each subtree and replace them by a leaf when they are considered not so impactful. To assess the relevance of the subtrees, they generally use a dataset different from the training set and/or specific criteria different from the one used to build the tree. Below, we describe the techniques used by CART and C4.5 to perform post-pruning.

Error Complexity Pruning (CART) The way that the CART algorithm prunes the tree after the learning process is based on multiple steps. This is explained in Algorithm 2.

Notice that the algorithm takes a dataset as input. Indeed, when a training dataset is passed to CART algorithm, it is split into two (2) parts: one used for the learning and the second for the post-pruning. It is the second dataset which is considered in the current algorithm.

The global idea of CART pruning algorithm is to successively transform some subtrees of the original tree into leaf nodes until there is no more subtree (only a root node remains) (lines 3-8). The intermediary trees obtained after each subtree replacement are saved. The variable $trees$ is used for this purpose in the algorithm (lines 1 and 7). In order to select the subtree that will be replaced by a leaf, a measure is computed for each internal node of the tree. Then the subtree rooted by the node having the lowest score is replaced. The calculated measure is called *error complexity*. Given a node m covering a data subset $\mathcal{D}_m$ and which

Algorithm 2: CART post-pruning of decision tree

```
input : $\mathcal{D} = \{\boldsymbol{x}, y\}^n$, a dataset used for pruning
input : $\mathcal{DT}$, the decision tree previously learnt
1  $trees \leftarrow$ list()
2  $new\_tree \leftarrow$ clone($\mathcal{DT}$)
3  while true do
4      if trees is not empty then $new\_tree \leftarrow$ clone($trees$.last())
5      $nodes \leftarrow$ getInternalNodes($new\_tree$)
       // data(m) returns the data covered by the node m
6      $\text{argmin}_{m \in nodes}$ error_complexity($m$) $\leftarrow$ buildLeafNode(data($m$))
7      $trees$.add($new\_tree$)
8      if $|nodes| = 1$ then break
9  $t_0 \leftarrow \text{argmin}_{t \in trees}$ tree_error$_{\mathcal{D}}(t)$
10 $error_0 \leftarrow$ tree_error$_{\mathcal{D}}(t_0)$
11 $se \leftarrow$ standard_error($error_0$)
12 $end\_trees \leftarrow \{t \in trees :$ tree_error$_{\mathcal{D}}(t) \in [error_0, error_0 + se]\}$
13 return $\text{argmin}_{t \in end\_trees}$ size($t$)
```

roots a subtree with a set $leafs$ of leaf nodes, the error complexity is expressed as:

$$error\_complexity = \frac{leaf\_error(\mathcal{D}_m) - \sum_{\ell \in leafs} leaf\_error(\ell)}{|leafs| - 1} \tag{1.11}$$

This error measures the complexity of the subtree rooted by m. More specifically, it measures the error reduction provided by the subtree on the dataset $\mathcal{D}_m$, over the size of the subtree needed for this reduction. The higher the complexity, the more impactful the subtree. So at each iteration of the algorithm, the less impactful subtree is removed and replaced by a leaf node. At the end of this process, one of the intermediary trees obtained from the subtree replacement processes should be used as the final tree. To accomplish this task, CART uses each of the trees found to predict on the unused data $\mathcal{D}$ provided as input of the algorithm and selects the one with the lowest classification error (line 9). Its error is then used to compute the standard error expressed as $standard\_error = \sqrt{\frac{E_0(1-E_0)}{|trees|}}$, where E_0 is the lowest misclassification error and $trees$ are the different intermediary trees. The final tree outputted by the algorithm is the one with the lowest size and having its error within the range $[E_0, E_0 + standard\_error]$ (lines 12-13).

Pessimistic Error Pruning (C4.5) In C4.5 algorithm, the technique to post-prune the trees is different. It does not require an additional dataset. It relies on a principle used by other algorithms [Qui86; Nib87]. The idea is to make a bottom-up traversal of internal nodes of the tree and compare them to their children on the basis of a measure. When the internal node score is better than the child scores, then the subtree rooted by the node is removed and the node is turned into a leaf node. In order to compare the internal nodes with their children, C4.5 used a measure called *pessimistic error*. It is based on a binomial distribution confidence interval. At each node, the number of instances and the proportion of these instances not belonging to the majority class are considered as a statistic sample of N trials with p events. Then an expected error rate can be derived in the *population* (unseen data) using a binomial distribution confidence interval. To increase the chance of pruning, Quinlan uses 25% as confidence level and uses the upper limit of the confidence interval as the expected proportion of error for the node when classifying unseen data. This measure for each node is compared to those of the children nodes. To compute the measure for the children nodes, a sum weighted by the proportion of instances in each child node is performed. When the parent reaches a lower expected error than the children, the subtree rooted by the parent is removed, and the parent node is turned into leaf. Otherwise, the parent node is kept and the bottom-up process continues.

1.4 Other type of decision trees

In the previous section, we presented the generic algorithm to learn a classification decision tree. Then, we emphasized how CART and C4.5 implement the subroutines of the generic learning algorithm. In this section, we present how the decision trees are used to perform other tasks than classification based on the reduction of misclassification error.

1.4.1 Regression

In supervised learning, the idea is to learn from existing observations in order to predict unseen ones. In the first part of this chapter, we considered the predictions to be categorical values. However, they can also be numerical values. A well-known example of this problem is the prediction of house sales price. This is called *regression* problems. In this kind of problem, all the possible values of the target variable are not necessarily present in the training data. The idea is thus to learn the relation between the feature vectors of the instances and the target vector. This allows to predict the real value of an unseen instance even if its value is not present in the training dataset. Numerous algorithms have been proposed to solve this task. In the context of this thesis, we explain how classification DT learning algorithms are adapted to infer decision trees capable of handling regression problems.

The most used error function when learning a regression model is the *Mean Squared Error* (MSE). Given a dataset $\mathcal{D}$ and a decision tree $\mathcal{DT}$ represented by a function $h(\boldsymbol{x})$, the MSE is computed using:

$$MSE(\mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{\mathbf{x},y \in \mathcal{D}} (h(\boldsymbol{x}) - y)^2 \tag{1.12}$$

As each node covers a subset of the training dataset, the MSE can be computed for each node and the total MSE of a tree is the sum over the MSEs of each leaf of the tree. The task of regression tree algorithms is thus to find trees that reduce the MSE. The closer the target vector values of a data subset, the lower is their MSE. The idea is thus to find splits that group the datasets into a closer distribution of target values. In order to choose the best splits to constitute the final decision tree, the well-known algorithms do not use a particular splitting criteria as in classification problems. Instead, they use the MSE score as a heuristic. Concretely, each possible split is assessed and the one that reduces the most the MSE is chosen as the best split. The MSE of a split is computed by summing the MSE of datasets falling in each branch of the split. Note, however, in Equation 1.12 that the value $h(\boldsymbol{x})$ predicted by the tree is needed to compute the MSE value. In each leaf of a decision tree, the predicted value $h(\boldsymbol{x})$ is chosen in order to reduce the MSE. The value $h(\boldsymbol{x})$ predicted by well-known algorithms is the mean of the target values of instances falling in a node.

1.4.2 Descriptive clustering

Most of the clustering algorithms in the literature do not provide insight into how clusters have been created. They are based on inter-cluster and/or intra-cluster distance metrics that they try to optimize using different greedy algorithms. Although the different distance measures optimized are well-known metrics, there is no way to assess the relevance of results produced by the different algorithms because, unlike supervised learning problems, there is no ground truth (target vector). It is also impossible to compare the different metrics to each other because they produce values that belong to different ranges. An approach to assess the relevance of clusters is to use reduction dimensionality algorithms [AW10; GR71; BG98] to make 2D or 3D visualizations. However, the relationship between the clusters and the features is broken and prevents the clusters from being assessed by domain experts.

In order to make clustering interpretable as some supervised learning algorithms, Blockeel et al. proposed in 1998 an algorithm to learn clustering trees. The idea is to learn a decision tree as in classification or regression. The only difference is that the leafs do not predict any categorical or numerical values. Instead, the leafs of the clustering trees represent clusters. The tree structure is used to describe the different clusters found by the tree. This allows the clusters to be interpretable and easily reviewable by domain experts. To achieve this result, they used logic programming to build trees represented by first-order logic rules. In addition, they used the maximized inter-cluster distance as a splitting criterion to choose the best split at each step of the algorithm.

1.4.3 Compact trees

The traditional way to learn a decision tree is to first grow it and then prune it to avoid overfitting. In contrast to the post-pruning process of CART which requires the generation of multiple intermediary trees, and the one of C4.5 which can sometimes preserve many nodes of the learned tree, another type of pruning emerged that is based on a specific criterion: Minimum Description Length (MDL) [Ris78; Ris87]. The MDL criterion focuses mainly on the length of information needed to describe a data. As a result, several researchers have used the MDL [RW88; QR89; WP93; VW94] criterion to reduce the tree size during the pruning phase. The idea was to use the MDL criterion to find a good balance between compact trees that make some errors and larger trees with lower errors. Rissanen and Wax [RW88] used the MDL criterion to derive splitting and pruning criteria, while Volf and Willems [VW94] also used it to determine in a recursive algorithm the good compact tree without necessarily performing a pruning process.

In this chapter, we only considered traditional top-down algorithms to learn decision trees. The second class of algorithms are the optimal algorithms. We discuss them in detail in Chapter 4.

Frequent Itemset Mining 2

2.1 Introduction

Advances in science and industry have resulted in the digitization of many services. The functioning of these services is mainly based on the use of data; either to store or to exchange information. The amount of data generated through digital tools is increasing exponentially. This is mainly due to the multiplication of dematerialized services and especially because of the services that require a maximum of information in order to offer a better user experience.

Traditionally, in order to offer a service or solve a problem, one relies on existing data about the task to solve, assuming that each situation is unique. Today, with the great storage capacity of computers, which has led to the storage of a huge amount of data, it is obvious that there could be elements of similarities shared by a few or even multitudes of events. Thus, the study of these similarities could allow to have a better understanding of certain events. In addition, the new learned knowledge could be used to enrich the knowledge base of other problems and facilitate their resolution. Also, it may allow one to predict certain future behaviors or phenomena.

In data mining, one of the main tasks is to find patterns in databases that can be shared by several transactions of the databases: this is called *pattern mining*. These patterns can be of several forms: texts, graphs, trees, etc. The multiplicity of items present in a database and the total number of transactions can strongly influence the time needed to explore the different patterns in order to retain the most interesting ones. Several techniques have been proposed in the literature to reduce the time required to find interesting patterns. In our thesis, we rely on patterns mining to propose efficient optimal decision tree learning algorithms. More precisely, we focus on the *Frequent Itemset Mining* (FIM) problem.

In this chapter, we introduce in Section 2.2 the different notations that will be used, and we define concretely the problem of frequent itemsets mining. In Section 2.3, we present Apriori, the first algorithm to find frequent itemsets in a database. Sections 2.4, 2.5 and 2.6 present some optimizations to speed up the discovery of frequent itemsets. Finally, in Section 2.7, we present frequent closed itemsets and why they are interesting.

2.2 Notation and problem definition

Consider the illustrative database represented in Table 2.1. Table 2.1a shows a database made up of several lines. Each line is a transaction

Table 2.1: Illustrative database

t_{id}	Items
t_1	{B}
t_2	{E}
t_3	{A, C}
t_4	{A, E}
t_5	{B, C}
t_6	{D, E}
t_7	{C, D, E}
t_8	{A, B, C}
t_9	{A, B, E}
t_{10}	{A, B, C, E}

(a) Transactional database

t_{id}	A	B	C	D	E
t_1	0	1	0	0	0
t_2	0	0	0	0	1
t_3	1	0	1	0	0
t_4	1	0	0	0	1
t_5	0	1	1	0	0
t_6	0	0	0	1	1
t_7	0	0	1	1	1
t_8	1	1	1	0	0
t_9	1	1	0	0	1
t_{10}	1	1	1	0	1

(b) Boolean matrix

of the database and is indexed with t_{id}, where id is the identifier of the transaction. Each transaction is represented by a set of items belonging to the transaction. This set of items is called *itemset*. For readability, we will interpret each line of the database as: *some items are present/selected in a transaction*. For example, in the transaction t_4, the items A and E have been selected, and we can note that $\{A, E\} \subseteq t_4$. The set of all transactions is denoted with $\mathcal{T}$. Here, $\mathcal{T} = \{t_1, \ldots, t_{10}\}$. On the other hand, the set of all the items present in the database is denoted $\mathcal{I}$. In the illustrative example, $\mathcal{I} = \{A, B, C, D, E\}$. More formally, the database is denoted $\mathcal{D}$ and can be seen as a collection $\{(t_{id}, I) | I \subseteq \mathcal{I}\}$ of transaction identifiers coupled to the itemset selected in the transactions.

The transactional database shown in Table 2.1a can be converted to a Boolean matrix format as presented in Table 2.1b. For each transaction, there is a column for each item. The value (0/1) of the item in the transaction indicates whether the item is selected in the transaction or not. This equality between the two representations will be used later in our optimal decision tree learning algorithms. In order to explain the problem of frequent itemset mining, there are some concepts that need to be introduced.

Cover Given an itemset $I \subseteq \mathcal{I}$, $cover(I)$ is the set of transactions that include the items present in the itemset I. It is expressed as $\{t'_{id} \in \mathcal{T} \mid (t'_{id}, I') \in \mathcal{D} \wedge I \subseteq I'\}$, where I' is the itemset selected in the transaction t'_{id}. In the example of Table 2.1, the cover of the itemset $\{B\}$ is $cover(\{B\}) = \{t_1, t_5, t_8, t_9, t_{10}\}$ while the itemset $\{A, C\}$ covers the transactions $\{t_3, t_8, t_{10}\}$.

Support The support of an itemset I is defined as the number of transactions in the cover of the itemset. We note it as $Support(I)$ and it is formally formulated as $Support(I) = |cover(I)|$. In the same way as the examples provided for the definition of the cover, $Support(\{B\}) = |\{t_1, t_5, t_8, t_9, t_{10}\}| = 5$ while $Support(\{A, C\}) = 3$.

Frequent Itemsets The frequent itemsets of a database are the itemsets whose support is greater than or equal to a threshold θ. The value of the threshold is user-defined.

The problem of frequent itemsets mining is therefore the task of finding the set $\{I \subseteq \mathcal{I} \mid Support(I) \geq \theta\}$ of all possible itemsets of $\mathcal{D}$ whose support is greater than or equal to θ. The problem has been introduced by Agrawal et al. in 1993 [AIS93]. For example, given a threshold $\theta = 2$, the itemset $\{A, E\}$ is frequent, as it appears in the transactions t_4, t_9, t_{10}. Its support is thus equal to 3.

2.3 Apriori

The *Apriori* algorithm [AS+94] is the first well-known algorithm to find frequent itemsets from a database. Notice that the problem of finding frequent itemsets is a non-trivial problem. In fact, the number of possible itemsets given a database $\mathcal{D}$ is exponential. As an itemset is simply a set of items present in the database, there is no precedence relation in an itemset. Therefore, an itemset $\{A, B\}$ is exactly equivalent to an itemset $\{B, A\}$. More specifically, any permutation of items in an itemset leads to the same itemset. This reduces the number of possible itemsets given the set $\mathcal{I}$ of items in $\mathcal{D}$. However, this number is still huge and is equal to $2^{|\mathcal{I}|}$. Due to the orderless character of items in an itemset, the search space of all possible itemsets given a database can be represented as a lattice. Figure 2.1 shows the search space of the itemsets given the database of Table 2.1.

In order to find the frequent itemsets given a database $\mathcal{D}$ and a threshold θ, a naïve approach would consist in exploring all the itemsets of the search space and for each of them, counting the number of transactions containing the itemset and checking if it is frequent. However, this approach would be very time-consuming. In order to reduce the part of the search space to explore, the Apriori algorithm relies on an important property of itemsets: *anti-monotonicity*.

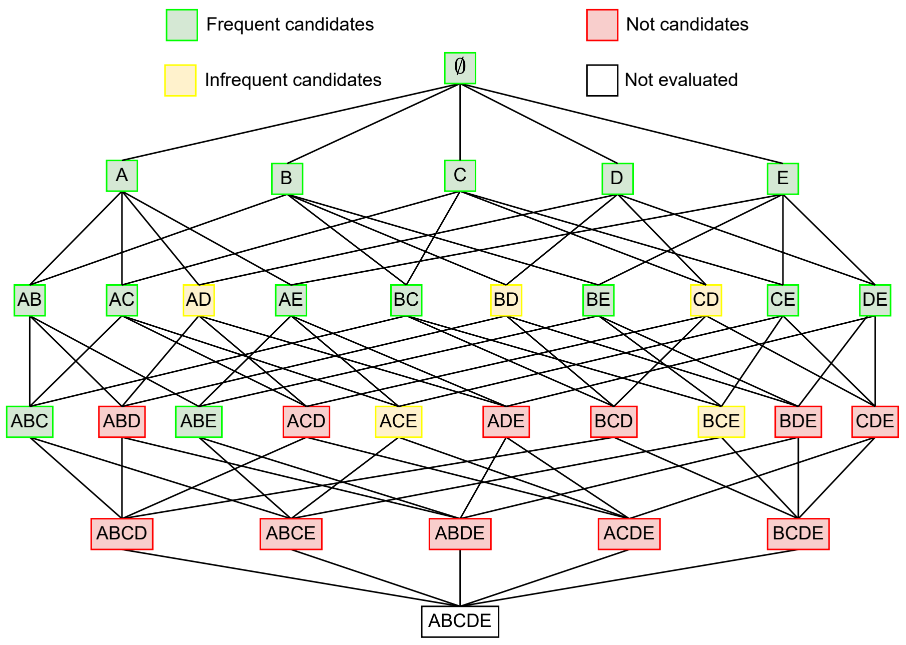


Figure 2.1: Itemsets search space and Apriori execution

Anti-monotonicity The anti-monotonicity property states that for two itemsets I and I' such that $I \subseteq I'$, the set T' of transactions covered by I' is also covered by I. Formally, it means that $T' \subseteq T \iff I \subseteq I'$, with I and I' covering T and T' respectively. The intuition of this property is that when an itemset I' is present in some transactions, any itemset I which is a subset of I', is at least present in all transactions containing I'. For example, the itemset $\{A, B\}$ is present in transactions $\{t_8, t_9, t_{10}\}$ of the database of Table 2.1. Notice that the itemset $\{A\}$ which is a subset of $\{A, B\}$, is also present in $\{t_8, t_9, t_{10}\}$. In addition, it is also present in transactions $\{t_3, t_4\}$.

The principle of anti-monotonicity gives a clue about the frequency of some itemsets. Precisely, the anti-monotonicity property entails that $I \subseteq I' \iff Support(I) \geq Support(I')$. Therefore, knowing that the itemset I' is frequent implies that I is also frequent. The opposite is also true. This implies that: *if an itemset I is known to be infrequent, all its supersets are also infrequents.* For example, the itemset $\{C, D, E\}$ is present only in the transaction t_7. Therefore, its support is 1. Notice that all its supersets $\{A, C, D, E\}, \{B, C, D, E\}, \{A, B, C, D, E\}$ are also infrequent and all have their support equal to 0.

This property avoids exploring all children nodes of an itemset in the search space when it appears that the itemset is infrequent. Concretely, the Apriori algorithm relies on the anti-monotonicity property to avoid evaluating some useless itemsets. Its pseudocode is presented in Algorithm 3.

Algorithm 3: Apriori

```
input : $\mathcal{D}$, a database of transactions
input : $\theta$, the minimum support used for the frequency
$\mathcal{F}_1 \leftarrow$ compute_1-itemsets() // frequent itemsets of size = 1
$\mathcal{F}_2 \leftarrow$ compute_2-itemsets() // frequent itemsets of size = 2
$k \leftarrow 3$
while $\mathcal{F}_{k-1}$ is not empty do
    Generate candidates $\mathcal{C}_k$ of $k$-itemsets using joins on $\mathcal{F}_{k-1}$
    Prune $\mathcal{C}_k$
    $\mathcal{F}_k \leftarrow \{I \in \mathcal{C}_k \mid Support(I) \geq \theta\}$
    $k \leftarrow k + 1$
return $\cup_{i=1}^{k-2} \mathcal{F}_i$
```

The term k-itemset is used to indicate an itemset whose size is k, i.e. it contains k items while $\mathcal{F}_k$ represents the set of frequent k-itemsets. Notice in the algorithm that Apriori is a level-wise algorithm. It computes the frequent itemsets by iteratively computing the k-itemsets for each level k. In order to compute efficiently the frequent itemsets, it uses the anti-monotonicity property to avoid checking some itemsets. The tricks used for this reduction of the search space cannot be applied to 1-itemsets and 2-itemsets. In Algorithm 3, some procedures (lines 1-2) are used to state that the frequent 1-itemsets and 2-itemsets are computed differently. For this computation, a naive approach would consist of enumerating every 1-itemset and 2-itemset and passing through the database $\mathcal{D}$ to compute their support and compare them with θ. However, an efficient approach has been explored in the literature [AAP01] to compute these specific itemsets.

Starting from the level $k = 3$, the Apriori algorithm uses a special trick to reduce the number of candidates to consider when looking for the frequent itemsets. The trick is based on the anti-monotonicity property. As the property states that all subsets of a frequent itemset are also frequents, the Apriori algorithm does not allow in the candidates $\mathcal{C}_k$ of potential frequent k-itemsets, some itemsets whose any subset was not known to be frequent at level $k-1$. To accomplish this task, it computes the candidates $\mathcal{C}_k$ by performing a joining operation. The idea is to join pairs of frequent itemsets found at level $k-1$ to compute the candidates $\mathcal{C}_k$. This removes from the candidates $\mathcal{C}_k$ the itemsets whithout any frequent subsets by ensuring that each itemset of $\mathcal{C}_k$ has at least 2 frequent subsets. For example, if we have two itemsets $\{i_1, i_2, i_3\}$ and $\{i_1, i_2, i_4\}$ in $\mathcal{F}_3$, the itemset $\{i_1, i_2, i_3, i_4\}$ that comes from the join of the two itemsets becomes a valid candidate and is placed in $\mathcal{C}_4$. To avoid duplicates of itemsets in $\mathcal{C}_k$ (different itemsets joined can lead to the same candidate), the joining of pairs of itemsets in $\mathcal{F}_{k-1}$ is performed only for pairs sharing the same first $k-2$ items, assuming that the itemsets are sorted using a specific order. This is why this process starts from $k = 3$.

After computing the candidate itemsets $\mathcal{C}_k$, Apriori does not count directly the support of the candidates. As the counting operation of each itemset is an expensive operation, the Apriori algorithm uses a technique to prune the candidate itemsets. As anti-monotonicity states that all subsets of a frequent itemset should also be frequent, the Apriori algorithm prunes some itemsets from the set $\mathcal{C}_k$ when all their subsets are not present in $\mathcal{F}_{k-1}$. This reduces the number of itemsets to consider in $\mathcal{C}_k$.

After pruning $\mathcal{C}_k$, the support of the remaining candidates is counted, and those whose support is greater than or equal to θ are considered as members of $\mathcal{F}_k$. A naive approach to count the supports is to perform a loop over the items of each transaction of the database. [AS+94] presents a more interesting technique to compute the supports based on the use of a *hash tree* data structure. The Apriori algorithm stops when there is a k for which $\mathcal{F}_k$ is empty. The algorithm then returns the union of sets $\mathcal{F}_1$ up to $\mathcal{F}_{k-1}$. Figure 2.1 shows an execution trace of the Apriori algorithm using the introductory database and $\theta = 2$. Note that, although the great candidate generation process and the pruning of these candidates, the Apriori algorithm still spends a considerable time counting the support of the remaining candidate itemsets. Many researchers have proposed some techniques to reduce the impact of support counting on the frequent itemset mining algorithms. In the next sections, we will present some of them.

2.4 Projection-based algorithms

In order to reduce the time required to count supports in the frequent itemset mining algorithms, Zaki et al. [Zak+97] propose an idea which is also used by Agarwal et al. in *DepthProject* [AAP00] and *TreeProjection* [AAP01] algorithms to simplify this operation. Even if the idea of these algorithms is a bit complex, in this thesis, we will simplify it to give just an overview of the main approach used to facilitate the support counting. In these works, they consider the search space of possible itemsets as a lexicographic tree (also called trie for *re-trie-val* tree) by defining order between the items. Figure 2.2 shows an example of the trie used as the search space when considering the introductory example.

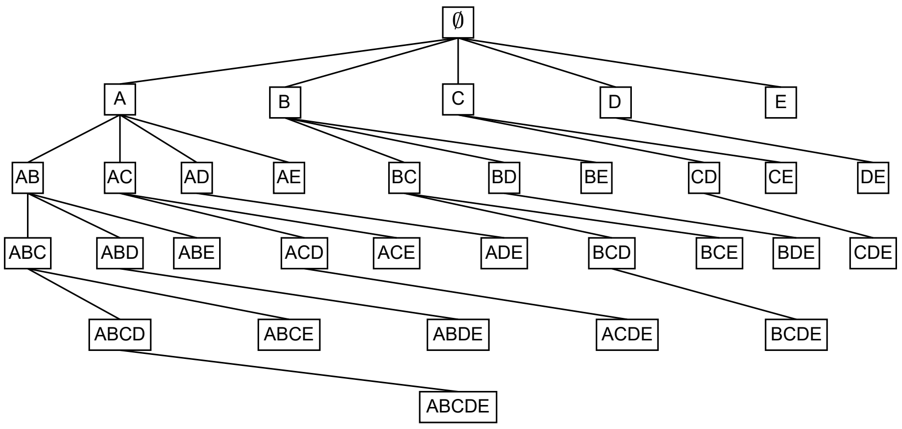


Figure 2.2: Lexicographic tree as itemset search space

To explore the search space, they use tree-based search algorithms like breadth-first search (BFS) or depth-first search (DFS) to explore it. By using these algorithms to explore the search space, they ensure that each node in the trie is explored before its children. In other words, they ensure that each itemset is checked before its supersets. Remember that according to the anti-monotonicity property, the transactions containing a superset of an itemset are a subset of the transactions containing the itemset itself. The idea of the technique proposed is thus to reduce the database needed to be used to count the support of itemsets as the search progresses. In this case, there is no need to pass multiple times through the whole database. The technique is called *database projection*.

As the search space is explored using a tree-based search algorithm, each time a node is explored, it corresponds to an itemset, and the database projection occurs to filter the part of the database needed to be used by the children of the itemset. The Table 2.2 shows an example of the database projection technique. Let us assume that we initially have the database shown in Table 2.2a then the search algorithm branches on the node representing the itemset $\{A\}$. At this point, notice that the itemset $\{A\}$ is not present in the transactions $\{t_1, t_2, t_5, t_6, t_7\}$. Therefore, these transactions become useless to be considered by the supersets of the itemset $\{A\}$. The database projection will then create a conditional database without these transactions. The new database will be stored in the node of itemset $\{A\}$ and will be used for the counting processes of the child itemsets. The resulting database is shown in

Table 2.2: Illustrative database

t_{id}	Items
t_1	{B}
t_2	{E}
t_3	{A, C}
t_4	{A, E}
t_5	{B, C}
t_6	{D, E}
t_7	{C, D, E}
t_8	{A, B, C}
t_9	{A, B, E}
t_{10}	{A, B, C, E}

(a) Original database

t_{id}	Items
t_3	{A, C}
t_4	{A, E}
t_8	{A, B, C}
t_9	{A, B, E}
t_{10}	{A, B, C, E}

(b) Relevant transactions

t_{id}	Items
t_3	{C}
t_4	{E}
t_8	{B, C}
t_9	{B, E}
t_{10}	{B, C, E}

(c) Relevant items

Table 2.2b.

Moreover, the database projection operation goes beyond the simple building of conditional databases. As the support counting is performed by checking each item in each transaction, the projection of the database also aims to reduce the number of items per transaction to decrease the number of checks required to perform the counts. For this, after filtering the relevant transactions, all the items that cannot be added to the current itemset to create a frequent itemset are removed for the conditional database. The final result for the example we are considering is shown in Table 2.2c. The item $\{A\}$ is removed. The others are preserved because the itemsets $\{A, B\}$, $\{A, C\}$ and $\{A, E\}$ have at least a support greater than or equal to the value of θ which is 2. The final reduced database is therefore the one passed to the child nodes. This allows to reduce the complexity of the support counting.

The algorithm used to guide the search can be a BFS or a DFS. This is agnostic to the size of the search space explored and does not affect the run time of the algorithm. However, there is a considerable problem with the BFS strategy according to the memory complexity. The number of nodes needed to be kept in memory is considerable in BFS and can be problematic for tasks involving a large database.

2.5 Vertical Representations

In this section, we introduce another technique used in literature to speed up the support counting. The technique is based on a vertical representation of the database and intersection operations. The technique has been proposed by Holsheimer et al. [Hol+95]. It was based on the *Monet*[1] database management system (DBMS). To perform the support counting, the transactional database is loaded in the DBMS and the instructions provided by the DBMS are used to perform the different operations needed to find the frequent itemsets. Concretely, the idea is based on the conversion of the original transactional database into a vertical representation. Table 2.3 shows an example of the vertical representation of a database. That vertical representation is supported by the *Monet* DBMS.

Table 2.3: Horizontal vs Vertical representation

t_{id}	**Items**
t_1	{B}
t_2	{E}
t_3	{A, C}
t_4	{A, E}
t_5	{B, C}
t_6	{D, E}
t_7	{C, D, E}
t_8	{A, B, C}
t_9	{A, B, E}
t_{10}	{A, B, C, E}

(a) Horizontal database

Items	*tid-list*
A	$t_3, t_4, t_8, t_9, t_{10}$
B	$t_1, t_5, t_8, t_9, t_{10}$
C	$t_3, t_5, t_7, t_8, t_{10}$
D	t_6, t_7
E	$t_2, t_4, t_6, t_7, t_9, t_{10}$

(b) Vertical database

In the classic representation of a transactional database, also called horizontal representation, each row represents a transaction listing the set of items contained in it. On the other hand, in the vertical transaction, each row represents an item associated to the list of the identifiers of the transactions in which the item is present. The list of transaction identifiers is called *tid-set* or *tid-list*.

Once the database representation has been turned into vertical and loaded into the DBMS, it becomes straightforward to find all the transactions covered by each item. Therefore, it is enough to count the size of the *tid-list* per item to find all the frequent 1-itemsets. The count operation was ensured by the DBMS. After this operation, the rows of infrequent items are removed from the database. To find the transactions covered by the 2-itemsets, an intersection operation is performed between the *tid-lists* of the frequent 1-itemsets using the DBMS. For example, to find the *tid-list* of $\{A, B\}$, the intersection $\{t_3, t_4, t_8, t_9, t_{10}\} \cap \{t_1, t_5, t_8, t_9, t_{10}\}$ of the *tid-lists* of the itemsets $\{A\}$ and $\{B\}$ is performed. This results to $\{t_8, t_9, t_{10}\}$. A count operation is thus performed on the resulting 2-itemsets and the infrequent itemsets are removed

[1]https://www.monetdb.org/

from the DBMS while the frequent ones are kept. The intersection operation provides the set of transactions covered by an itemset and aims at reducing the number of transactions after each call. To find frequent 3-itemsets two possibilities were considered in [Hol+95]: the intersection between 2-itemsets and 1-itemsets or the intersection between two 2-itemsets. For example, the itemset $\{A, B, C\}$ can be built by intersecting $\{A, B\}$ and $\{C\}$ or by intersecting $\{A, B\}$ and $\{A, C\}$. The advantage of the last technique is that both intersected itemsets are generated from another intersection, and their *tid-list* is thus reduced. This speeds up the intersection process. Moreover, from a memory perspective, after the frequent 2-itemsets found, there is no need for the 1-itemsets and they can be removed from the DBMS. The process is level-wise like the Apriori algorithm. Therefore, all rows of the DBMS representing the k-itemsets can be dropped after computing the frequent $(k+1)$-itemsets.

After the introduction of the vertical representation and the intersection operations, Zaki et al. proposed the *Eclat*[Zak+97] algorithm. In this algorithm, they incorporate the vertical representation and the intersection operations into a tree-based algorithm. The use of a DBMS as an operation performer has been dropped by Zaki et al. [Zak+97]. Eclat algorithm has been designed to be a standalone algorithm exploring a lexicographic tree to find the frequent itemsets. The dataset is read and saved in memory as a vertical database, and the intersection operations during the exploration of the search space allows the reduction of the database subset used to count the supports. Eclat uses a BFS to explore the search space. Because the algorithm explores a tree, the database obtained after each intersection is stored at each node for the child nodes and this has an impact on the memory usage. In order to reduce this impact, Zaki and Gouda propose *dEclat* [ZG03] algorithm. It is an improvement of *Eclat* which is based on a DFS strategy. This reduces the number of nodes to be kept in memory. Moreover *dEclat* uses a concept called *diffsets* to reduce the size of the *tid-lists* stored in the nodes of the search space. The idea is to store for each node of the search space, the difference between its *tid-list* and that of its parent. This is proven to considerably save the memory usage of the algorithm.

2.6 Bitwise operations

In addition to the vertical representation of the databases, Burdick et al. propose another variation of the database representation [BCG01]. The idea still relies on the vertical representation. However, instead of maintaining for each item, a *tid-list* recording the set of the identifiers of transactions covered by the item Burdick et al. maintain a bit vector.

The idea is to keep for each item of a database $\mathcal{D}$, a bit vector of size $|\mathcal{D}|$. Each bit at index i in the bit vector has the value 1 or 0 and this expresses whether the transaction t_i is covered or not by the item. The bit vector of each itemset is a binary representation of its cover. The first idea was to reduce the memory consumption of the algorithm. Even when the number of elements kept by the bit vector is larger than the *tid-list*, the memory consumption of the bit vector is low. In fact, the number of identifiers maintained by the *tid-list* reduces as long as we go deep in the search, but in case of bit vectors, the number of elements maintained is still $|\mathcal{D}|$. However, the identifiers maintained by the *tid-list* are integers and need generally 32 bits to store each of them. On the other hand, the bit vector uses only 1 bit to indicate the coverage or not of a transaction.

In the context of bit vectors, the support counting is performed by counting the number of bits set to 1. In the same way as for vertical representation algorithms, the cover of $(k+1)$-itemsets is obtained by intersecting the bit vector representing the cover of k-itemsets or 1-itemsets. When the itemsets to evaluate during the search are long, many transactions of the database are no longer covered by the long itemsets. In the case of the *tid-lists*, the list reduces as long as the itemsets becomes longer, but the size of the bit vector remains unchanged, and it becomes time-consuming to count the support by taking into account transactions which are not covered. To reduce this impact, [BCG01] perform a database projection in the nodes for which their parent reaches a certain support. This operation removes the bits of the transactions not covered, but it requires a specialized data structure to keep the indices of transactions not removed to perform reliable intersections. The projection operation reduces the cost of intersections and counting but adds a non-negligible overhead.

In 2017, Schaus et al. shows that recent advances in computer technology lead to a great performance of bitwise operations. Concretely, they showed that bitwise operations are efficient when performed by computers of 64 bits architecture on bit vectors of 64 bits. This speeds up the counting and intersection processes. They rely on this advance to propose an efficient algorithm to find frequent itemsets using bit vectors and constraint programming [SAG17]. The algorithm proposed does not perform a concrete projection operation as in [BCG01]. However, they rely on a specialized data structure, the *Reversible Sparse Bitset* [Dem+16], to avoid performing some useless operations. Moreover, the used data structure reduces the memory consumption because a unique instance of the data structure is used to maintain the bit vectors along all the nodes of the search space. We will detail the Reversible

Sparse Bitset data structure in the next chapter, as we rely on it to build the algorithms we propose in this thesis.

2.7 Frequent closed itemsets

In the pattern mining literature, there is a lot of research on the compression of frequent itemsets. Concretely, the objective is to find a limited set of patterns that can be used to retrieve the whole set of itemsets without loss of information. Pasquier et al. propose the concept of *frequent closed itemsets*. These are itemsets that do not have any superset that has the same support as them. More formally, an itemset $I \subseteq \mathcal{I}$ is a closed itemset $\mathit{iff}\ \nexists i \in \mathcal{I} \mid Support(I \cup \{i\}) = Support(I)$. This itemset is frequent if its support is greater than or equal to the threshold θ. An interesting characteristic of closed itemsets is that, given a database, all frequent itemsets and their corresponding supports can be derived from the complete set of the frequent closed itemsets. Another characteristic of closed itemsets is that there is only one closed itemset that covers a specific set of transactions. Therefore, a closed itemset can be used to uniquely represent a set of transactions. There exist many algorithms [Pas+99; PHM+00; ZH02; Wan+05] to find frequent closed itemsets, but in this section, we intentionally do not provide any details on these algorithms. The idea here is just to describe what a closed itemset is, as we will talk about it later in the thesis.

Reversible sparse Bitsets 3

3.1 Introduction

In discrete optimization problems, the aim is to assign discrete values of specific domains to some variables in order to optimize a specific function which relies on the variables. The well-known algorithms that show impressive results in solving discrete optimization problems are mostly based on specialized techniques. These techniques are mostly based on search algorithms. Among these techniques, we can cite Dynamic Programming (DP), Branch-and-Bound (BnB), Constraint Programming (CP), etc. Apart from the interesting properties used by these techniques to solve combinatorial problems, the implementation of the algorithms using these techniques is crucial to show great performance in terms of space/time complexities. For this reason, many of the implementations of the algorithms that rely on these techniques use specific algorithmic techniques to increase their performance. Generally, they use tailored data structures that can cover the specific problem at hand. Some of these data structures are generic enough to cover other kinds of problems than the one for which they were designed. This is the case of the Reversible Sparse Bitsets (RSBS) [Dem+16] data structure designed by Demeulenaere et al. for CP solvers to efficiently solve the well-known *table constraint* or *extensional constraint* [BR97; Gen+07]. We use the RSBS data structure in this thesis to speed up the learning of ODTs.

In the next sections of this chapter, we describe the RSBS data structure and highlight the cases in which it can be used.

3.2 Bitsets as state representation

Most of the algorithms designed to solve combinatorial problems till optimality rely on search algorithms. These algorithms generally explore a search space represented as a tree, in which each node in a path represents an assignment of a value to a variable of the problem. The state of a node in the search space generally depends on its ancestor states. Therefore, to ensure the consistency of the results found from the search, a certain number of parameters needs to be maintained at each node during the search so that the child nodes can use them. An example of a parameter generally maintained in the search algorithms is the set of remaining values to explore for each variable of the problem. There are many other kinds of parameters that can be maintained during a search. This depends mainly on the problem to solve. Let us consider a parameter p that contains a list of integer values and that should be maintained. Assume also that different operations can be performed on p to remove some of its elements depending on the point where we are in the search. It is important to use the right data structure to represent p. Since the set of possible values of p is discrete, an intuitive way to represent it is an array. The RSBS provides a way to represent this kind of parameter. However, it does not use a classic array for its implementation. Instead, an RSBS represents the list of integers as a bitset.

An array containing elements from a finite set can easily be turned into a bitset. Figure 3.1 shows an example of equivalency between an array and a bitset. In this example, we assume that the finite domain Ω to which any value of p should belong is $\Omega = \{1, \dots, 8\}$, that is, p can contain only integers ranging from 1 to 8. We also assume that p currently contains all values of Ω. In Figure 3.1a, an array is used to keep the list of the eight integer values of p, while a bitset holds a list of eight bits related to each value in Ω (Figure 3.1b). For each bit at the index i in the bitset, the value 1 denotes that the integer at the index i in Ω is contained in p. The value 0 means the opposite. Note that the indices are considered starting from 0.

Array:	1	2	3	4	5	6	7	8

(a) Array structure

Bitset:	1	1	1	1	1	1	1	1

(b) Bitset structure

Figure 3.1: Array vs Bitset

When after some operations, some values are no longer possible to be considered in p, the array structure is reduced to a shorter one that contains only valid values. This is shown in Figure 3.2a where the values 5 and 7 are removed from p. On the other hand, the bitset shown in Figure 3.2b still has the same size as Ω but the bits at the indices of the values 5 and 7 are turned into 0, denoting that the elements at these indices in Ω are no longer present in p.

Array:	1	2	3	4	6	8

(a) Array structure

Bitset:	1	1	1	1	0	1	0	1

(b) Bitset structure

Figure 3.2: Filtered array and bitset

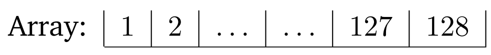
The first advantage of using bitsets instead of arrays is the memory usage. Even when some values are removed from p as long as the search evolves, the memory required to store the array of integers is generally higher than that required for a bitset when the sets are large. In fact, each integer in an array requires 32 bits in the common programming language implementations to be stored. In the worst case, it can require 64 bits. On the other hand, a bitset only requires 1 bit to denote the presence of an integer. The parameter p would contain at least 32 times fewer values than Ω for the array to be a good option. But in practice, it is generally less memory-consuming to use a bitset than an array.

Due to the representation of bitsets, they are more suitable for the cases in which the number of elements maintained is more relevant than the data itself. This is why they are used to represent transactions in many frequent itemsets mining algorithms [BCG01], where the aim is to find itemsets that cover a certain number of transactions. The bitsets provide very efficient instructions to perform bitwise operations such as the count. A count of the number of elements in an array is performed in $\mathcal{O}(n)$, where n is the size of the array. In case of bitsets, the structure encodes a one-hot encoding of the array. Therefore, the count of the number of elements is equivalent to a count of the number of bits set to 1. In most of the programming languages, this operation is performed efficiently on 64-bitset (bitsets of size 64), by using the CPU instruction `popcnt` when available. Otherwise, it is performed using a pre-populated lookup table loaded in the CPU cache. These instructions provided by the programming languages or the CPUs allow the count operation to be performed in constant or approximately constant time on 64 bits at once. This is due to the fact that the basic data unit manipulated by 64 bits CPUs is data of 64 bits. Similarly, many other bitwise operations, such as the intersection ($\cap$) or the union ($\cup$) are also optimized to be performed on 64 bits at once. This makes the bitsets relevant to store parameters that require these kinds of operations.

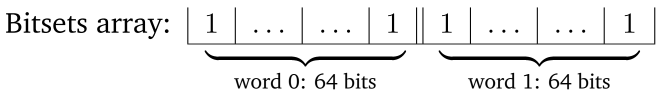
To benefit from these optimizations, most of the data structures that rely on bitsets implement them as an array of bitsets of size 64. Each 64-bitset in the array is called a *word*. The number of words needed is thus the ceiling of the size of the domain divided by 64. Figure 3.3 shows an illustrative example. Assuming a list of 128 values from 1 to 128 in Ω and p, an array of size 128 is needed, while a bitset-based implementation uses an array of 2 64-bitsets.

3.3 Sparse Bitsets

An important element to take into account when counting the number of elements in a set is the size of the set. When using arrays, the size



(a) Array structure



(b) Bitset structure

Figure 3.3: Array vs array of bitsets

of the structure corresponds exactly to the number of elements that are included. In contrast, the bitsets structure always has the same size as Ω, even when some values are no longer considered. For this reason, even when there exist efficient bitwise functions to perform operations, the fact of considering many more elements than a classic array can be problematic if Ω is large. This situation occurs in the Mafia [BCG01] algorithm that converts transactions into bitsets to find frequent itemsets. As the number of elements present in the set is getting low, it becomes less efficient to consider a bitset containing a large number of bits instead of an array of few values. To mitigate this drawback of bitsets, [BCG01] considers the option of removing the bits set to 0 when they become numerous. However, this requires additional operations to perform the filtering and to maintain the list of indices of remaining bits.

In the case of RSBS, the fact that the operations are performed per word of 64 bits provides better performance over Mafia. Moreover, the bitsets are implemented as sparse bitsets. The idea is to integrate in the data structure, a variable that keeps information on regions where there are a bunch of 0s in order to avoid operating on them when necessary. To facilitate the identification of these regions, the RSBS uses the slicing of bitsets into an array of words. The words that have all their bits set to 0 are those identified as empty regions. Figure 3.4 shows an example of what a sparse bitset looks like.

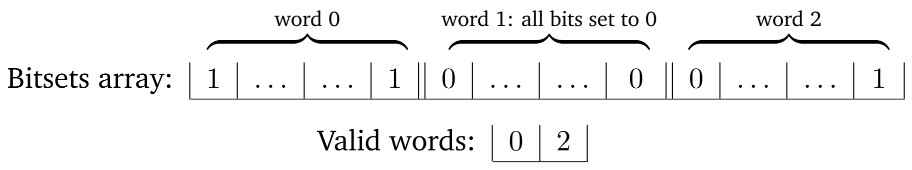


Figure 3.4: Sparse bitset

Notice the presence of an additional array denoting the indices of non-zero words to consider. Keeping information about empty regions is interesting, as it prevents from operating on these regions in the case of some operations like the count. Indeed, it is useless to check the number of bits set to 1 in a region where all bits are set to 0. Moreover, the sparsity introduced is also relevant for the intersection operation, where it is known that the intersection of a 0 bit and any other one always results in a 0.

3.4 Reversibility

Even though the use of a sparse bitset often reduces the memory consumption compared to a classic array and also shows interesting performance in terms of time, it still shares an important drawback with arrays. Specifically, in search algorithms, many parameters stored at a node are passed to the child nodes to preserve the consistency of the search. Generally, a copy of the relevant parameters is made from the parent node and is passed to the children so that the children can operate on them. This copy generates a non-negligible overhead as it is performed each time a node is expanded. Using a sparse bitset instead of an array does not avoid this copy. To prevent this behavior, the RSBS uses a technique generally used in CP solvers: *the reversibility*.

In constraint programming, many data structures are implemented in a reversible way. This allows the structures to recover their previous states after different changes. This is useful when the structures are maintained at the different nodes of the search. In this case, only one instance of the data structure is required during the whole search. At each step of the search, different changes can be applied to the unique instance. However, when another path of the search has to be explored, the structure can roll back its state so that another path can be explored without inconsistency. This mechanism is often used in CP solvers and is called the *trailing* [MSV17]. The principle is simplified here for our purposes.

Specifically, the reversible structures are implemented as a stack. Doing this, the different changes appear as successive layers added on top of the initial state. When a backtracking is performed, the top layers are popped so that the old states are retrieved. This is straightforward when the structure is just an integer. However, it becomes more complex when the reversible structure is a set and its successive states can remove some elements from the previous ones, as for the empty words in a sparse bitset.

Algorithm 4 shows the `RsparseBitset` class provided to implement a reversible sparse bitset data structure. Only the methods useful for the ODTs learning are shown here. The others can be implemented by

Algorithm 4: `RsparseBitset`

```
words : array of stack of 64-bitsets // layers of bitset arrays
nonZeroIndices : array of int // the array used for the sparsity
limit : stack of int // last index of non-zero words in nonZeroIndices
Method intersect(mask : array of 64-bitsets)
    cLimit ← limit.top()
    for i from 0 to cLimit do
        realIndex ← nonZeroIndices[i]
        newWord ← words[realIndex].top() & mask[realIndex]
        words[realIndex].push(newWord)
        // update the list and number of empty words
        if newWord = $0^{64}$ then
            nonZeroIndices[cLimit] ← nonZeroIndices[i]
            nonZeroIndices[i] ← realIndex
            cLimit ← cLimit − 1
    limit.push(cLimit)
Method backtrack()
    limit.pop()
    for i from 0 to limit.top() do
        realIndex ← nonZeroIndices[i]
        words[realIndex].pop()
Method countSetBits()
    val ← 0
    for i from 0 to limit.top() do
        realIndex ← nonZeroIndices[i]
        val ← val + words[realIndex].popcnt()
    return val
```

performing a loop over each word of the bitset array. The class requires 3 variables:

(1) *words* This variable represents the current state of the reversible sparse bitset. It is similar to the sparse bitset array presented in the previous section. The main difference is that each word in the bitset array is implemented as a stack. This allows the words to be reversible, as a stack can pop the top elements to recover the previous ones. Each stack of the words is initialized by the successive words of the first bitset array used as the initial state.

(2) *nonZeroIndices* This variable is the same as the array of valid words shown in Figure 3.4. It maintains the list of the indices of words that are not equal to 0. It is required to implement the sparsity. The fact of keeping this list allows one to avoid operations on useless regions of the bitset array. It is initialized by the indices of the non-empty words of the initial bitset array and is updated as long as the state of the bitset array changes.

(3) *limit* This value is a stack of integers, where the integer k on the top states that it is only the indices from 0 to k in *nonZeroIndices* which contain the current non-empty words of the bitset array. It is also initialized by the information of the initial state and updated each time the state changes.

Figure 3.5 shows an example of the initial state of a reversible sparse bitset. The variable *words* is initialized with a sparse bitset. Notice that in the initial state, the word at the index 1 is empty. For this reason, the index 1 is not in the list of the *nonZeroIndices* variable. The variable *limit* keeps the information about the last index in *nonZeroIndices* which contains empty words for the current state. In fact, as long as the state of the RSBS changes, the number of empty words increases and the *nonZeroIndices* variable is updated. But in reality, as the RSBS needs to roll back to previous states, the indices in *nonZeroIndices* are not removed when empty words appear. Instead, the indices in *nonZeroIndices* are reordered so that the indices of empty words for the current state are placed at the beginning of the list, and the variable *limit* keeps track of the last index where ends up the empty words of the current state. As the initial state keeps track only of non-empty words, all the indices in *nonZeroIndices* are valid. Thus, the *limit* variable is set to the maximum index, here 2.

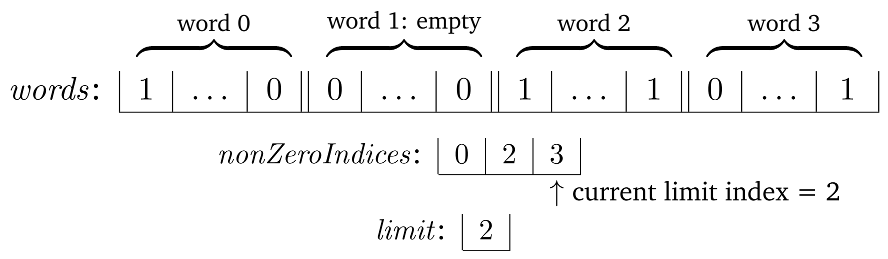

word 0
word 1: empty
word 2
word 3
words: 1 ... 0 || 0 ... 0 || 1 ... 1 || 0 ... 1
nonZeroIndices: 0 2 3
↑ current limit index = 2
limit: 2

Figure 3.5: Example of initial state of a reversible sparse bitset

When an intersection occurs between a RSBS and another bitset, some operations need to be ensured to preserve the consistency of the RSBS. This is described by the method `intersect()` (lines 4-13) of Algorithm 4. The intersection is performed only on the non-empty words, as the intersection of 0 bits with another one will not change its state. When the intersection of a word leads to a new word with all its bits set to 0 (0^{64}), the $nonZeroIndices$ is updated. For this, a swap is made between the current index and the last one. Then the current limit is updated by decreasing its value by one, denoting that the current index is no longer valid. The new limit obtained after looping over all non-empty words is added on top of the $limit$ stack. An example is shown in Figure 3.6. Consider that its previous state is the one shown in Figure 3.5.

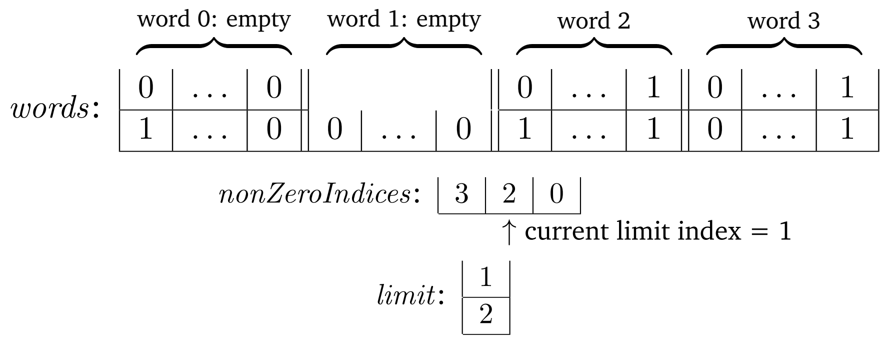


Figure 3.6: A reversible sparse bitset state after an intersection

During the intersection, there is no operation performed on the word 1 which was empty. Moreover, the word at the index 0 leads to an empty word after the intersection. Therefore, its index 0 is swapped with the last index 3 and the current limit of non-empty words indices is decreased and set to 1. The 1 is thus pushed on top of the $limit$ stack denoting that only the indices 3 and 2 contain non-empty words at the current state.

To roll back to the previous state, the method `backtrack()` is used. This function is quite simple. The top value of $limit$ is popped so that the last limit is revealed. Then, for each non-empty word of the previous state (found in $nonZeroIndices$), its previous state is recovered by popping the bitset on top of the stack. Notice in Figure 3.6 that the order of $nonZeroIndices$ will not be preserved. However, all the non-empty word indices are restored.

The count function detailed previously is also described in the algorithm by the method `countSetBits()`. It performs a loop on the non-empty words. For each of them, the number of bits set to 1 is efficiently counted using the CPU instruction `popcnt` or other instructions provided by the programming language. Then a sum is performed over the number of bits per word and the total is returned.

Part II

ODTs under constraints

Learning ODTs under depth and support constraints

4

This chapter is based on the paper G. Aglin, S. Nijssen, and P. Schaus. "Learning optimal decision trees using caching branch-and-bound search". In: Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 34. 04. 2020, pp. 3146–3153.

4.1 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 exist efficient algorithms for learning trees of acceptable quality.

The most well-known algorithms for learning decision trees, such as CART [Bre+84] and C4.5 [Qui93], as discussed in Chapter 1, 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 NP-hardness of the learning problem [LR76], 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 binary 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 [BD17; VZ19; AAV19]. 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 binary 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 [NF07; NF10]. 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 [Agr+96] 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 chapter, 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. However, 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 [VZ19]. We will show that DL8.5 outperforms this method on both types of data.

This chapter 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.

4.2 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 this end, 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.

Verhaeghe et al. (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 this chapter 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 [Nar+18; BHO09]. SAT-based studies, however, focus on a different type of decision tree learning problem than the MIP-based approaches which is 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.

4.3 Background

4.3.1 Dataset binarization

As the running example to illustrate the optimal decision tree learning problem, we will use the dataset of Table 4.1. It consists of 3 Boolean features (or attributes) and 11 examples (also called instances or data points). Indeed, our algorithm, like most of optimal decision trees learning algorithms, operates on Boolean data.

Table 4.1: Example database

A	B	C	class
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

While it may seem a limitation to propose a learning algorithm that only operates on Boolean data, there exist ways to transform any tabular database into a Boolean database. The binarization process consists in converting each non-Boolean feature column of a database into columns of Boolean (0/1) values. To accomplish this task, the process depends on the feature type.

Categorical features For categorical features, a one-hot encoding can be performed. A one-hot encoding is a process by which an original feature column of a dataset is turned into a set of b Boolean columns, with b equal to the number of distinct values in the original feature column. The values in the new columns are set to 1 when the original feature value is equal to the new column name. Values of other features are set to 0. An illustrative example is provided by Table 4.2. The original color feature of Table 4.2a is converted into Boolean features (Table 4.2b), where each feature value denotes whether the data point value for the original feature is equal to the name of the feature or not. There are 4 distinct values in the original feature. Therefore, the feature has been converted into 4 Boolean features representing each color. Note that one-hot encoding prevents information loss while converting a categorical dataset into a binary one.

Table 4.2: One-hot encoding

Color
Red
Red
Yellow
Green
Blue
Yellow

(a) Categorical feature

Red	Yellow	Green	Blue
1	0	0	0
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
0	1	0	0

(b) Generated Boolean features

Numerical features In the case of numerical features, the one-hot encoding leads to information loss. As the features can take any real value, considering only values present in the feature column may cause an ignoring of other possible values. As shown in Chapter 1, the suitable binary splits for numerical features are of type $feature < threshold$. This means that a split considers all values less than a threshold to be 1 while others are considered as 0. Performing this technique by using each distinct value of the original feature as a threshold will ensure the creation of Boolean features without information loss. This technique is illustrated in Table 4.3. In the original feature column (Table 4.3a), some height values in meter are shown. For each, a threshold of type

$height < threshold$ is created. The data point values are thus turned into 1 for the threshold which requirement is fulfilled while others are set to 0 (Table 4.3b).

Table 4.3: Binarization of a numerical feature

Height
1.12
1.55
1.23
1.85
1.9
1.6

(a) Numerical feature

< 1.12	**< 1.23**	**< 1.55**	**< 1.6**	**< 1.85**	**< 1.9**
0	1	1	1	1	1
0	0	0	1	1	1
0	0	1	1	1	1
0	0	0	0	0	1
0	0	0	0	0	0
0	0	0	0	1	1

(b) Generated Boolean features

4.3.2 Itemset mining for decision trees

DL8 is an algorithm that takes an itemset mining perspective on learning decision trees. In this perspective, the binary matrix shown in Table 4.1 and reminded in Table 4.4a is transformed into the transactional database of Table 4.4b. The optimal decision tree for this database can be found in Figure 4.1a.

Table 4.4: Database conversion

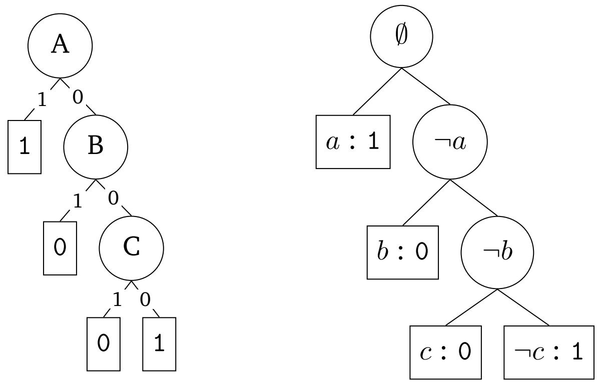
A	B	C	class
0	1	1	0
1	0	1	1
0	0	1	1
0	1	0	0
1	0	0	1
0	0	0	0
0	0	1	0
1	1	0	1
0	0	0	1
0	0	1	0
0	0	0	1

(a) Binary matrix

Id	items	class
1	$\neg a, b, c$	0
2	$a, \neg b, c$	1
3	$\neg a, \neg b, c$	1
4	$\neg a, b, \neg c$	0
5	$a, \neg b, \neg c$	1
6	$\neg a, \neg b, \neg c$	0
7	$\neg a, \neg b, c$	0
8	$a, b, \neg c$	1
9	$\neg a, \neg b, \neg c$	1
10	$\neg a, \neg b, c$	0
11	$\neg a, \neg b, \neg c$	1

(b) Transactional database

Each transaction of the new dataset (Table 4.4b) contains an itemset describing the presence or absence of each feature in the original dataset (Table 4.4a). More formally, the transactional database $\mathcal{D}$ can be thought of as a collection $\mathcal{D} = \{(t, I, c) \,|\, t \in \mathcal{T}, I \subseteq \mathcal{I}, c \in \mathcal{C}\}$, where



(a) Common view **(b) Itemset view**

Figure 4.1: Optimal tree corresponding to database of Table 4.4. Max depth = 3 and minimum examples per leaf = 1

$\mathcal{T}$ represents the transaction or rows identifiers, $\mathcal{I}$ is the set of possible items, and $\mathcal{C}$ 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 classic decision tree can be mapped to an itemset $I \subseteq \mathcal{I}$. Therefore, a classic decision tree can be converted to an equivalent itemset-based decision tree. This is shown in Figure 4.1b where the classic decision tree in Figure 4.1a is shown with an itemset view. The last path from left to right in the tree in Figure 4.1a corresponds to the itemset $\{\neg a, \neg b, \neg c\}$. Note that multiple paths can be mapped to the same itemset because the paths are ordered while itemsets are not.

For every itemset I, we define its cover to be $cover(I) = \{(t', I', c') \in \mathcal{D} \mid I \subseteq I'\}$: the set of transactions in which the itemset is contained. This is equivalent to the cover concept used in pattern mining. Moreover, pattern mining defines the support to be the number of elements in a cover. In decision tree problems, the class-based support of an itemset is defined and is expressed as $Support(I, c) = |\{(t', I', c') \in cover(I) \mid c' = c\}|$. It 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:

$$leaf\_error(I) = \mid cover(I) \mid - \max_{c \in \mathcal{C}} \left(Support(I, c) \right) \tag{4.1}$$

This error of an itemset I is by equivalence the error of a leaf node ending a path mapped by the itemset I. It is called the misclassification rate of the classification error. Unlike the CP-based approach, our error function is also valid for classification tasks involving more than 2 classes. For each itemset representing a path to a leaf node, a class is associated using the expression:

$$maxclass(I) = \operatorname*{argmax}_{c \in \mathcal{C}} \big(Support(I, c)\big) \tag{4.2}$$

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 $\mathcal{DT}$ 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 ignore the minimum support constraint.

4.3.3 DL8 Algorithm

The optimal tree is calculated recursively by DL8 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 subsets based on the values of different features, and determining the best subtree for each partition recursively. Specifically, the tree with the lowest error is computed by recursively looking for the best subtree at the left split and the best subtree at the right split. To find the best subtree at each side of a split, all the possible splits are recursively assessed and the one producing the lowest error is picked. Given a dataset with a set $\mathcal{F}$ of features, each splittable in items i and $\neg i$, the lowest error can then be computed by using the following recursive formula:

$$min\_error(I) = \begin{cases} \min_{F \in \mathcal{F}} min\_error(I \cup \{i\}) + min\_error(I \cup \{\neg i\}) & \text{if } |I| < maxdepth; \\ leaf\_error(I) & \text{if } |I| = maxdepth, \end{cases} \tag{4.3}$$

Figure 4.2 illustrates the search space of itemsets for the dataset of Table 4.4, where all the possible itemsets are represented.

Intuitively, DL8 starts at the root node of this search space, and calculates the optimal decision tree for the root based on its children.

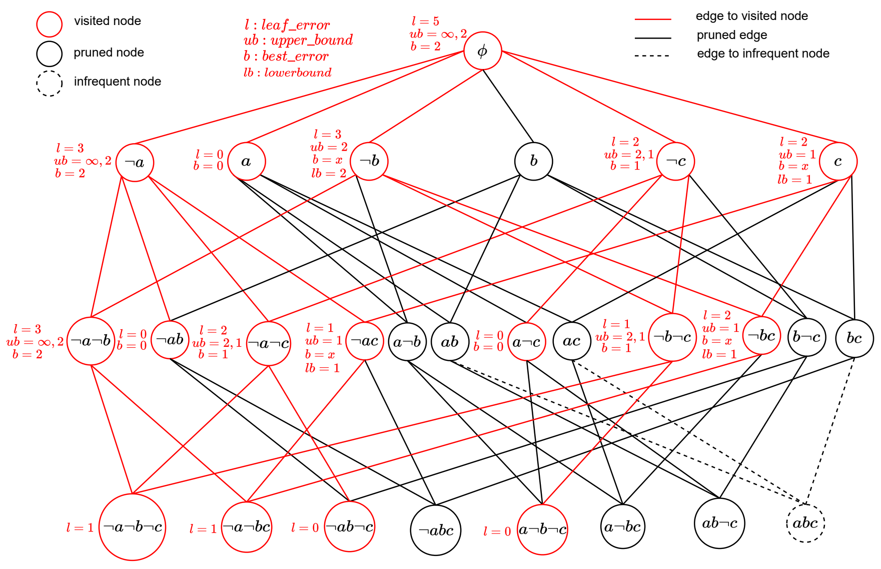


Figure 4.2: Complete itemset lattice for introduction database and DL8.5 search execution

4.3.4 Caching in DL8

A distinguishing feature of DL8 is its use of a cache. The idea behind this cache is to store the optimal subtree found for each itemset. 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. DL8 uses a lexicographic tree also known as trie or prefix tree, to implement its cache. In fact, in addition to the number of elements stored in the cache, the memory consumption of the algorithm depends also on the size of each itemset stored. To reduce this impact, the trie data structure allows to reduce the number of items to store to represent all itemsets. Specifically, items shared by multiple itemsets are represented once in trie. Algorithm 5 shows how the trie is implemented in DL8. The function `insertOrGet()` is used to insert an itemset in the cache and prepare the structure that will hold the solution for the itemset. It is represented in the pseudocode by the ***BestTree*** data structure. After the insertion of a new itemset, its leaf error is filled in the solution structure (lines 13-14). However, when the itemset already exists in the cache, the function outputs the existing solution for the itemset. DL8 uses a sorted representation of itemsets to uniquely represent all the itemsets composed of the same items.

Another interesting aspect of this cache is the solution structure that

Algorithm 5: Class Trie_Cache

```
1  struct Entry{solution: BestTree; childEntries: HashSet<int, Entry>}
2  struct BestTree{lb : float; feat : int; error : float }
3  rootEntry ← Entry(BestTree(0, NO_FEAT, +∞), {})
4  Method insertOrGet(I : array of int) // I: itemset
5      entry ← rootEntry
6      for i ∈ I do // foreach item in the itemset
7          if i ∈ entry.childEntries then
8              entry ← entry.childEntries.get(i)
9          else
10             childEntry ← Entry(BestTree(0, NO_FEAT, +∞), {})
11             entry.childEntries.push({i, childEntry})
12             entry ←childEntry
13     if entry.solution.error = +∞ then
14         entry.solution.error ← leaf_error(I)
15     return entry
16 Method exploreTree(I : array of int) // I: itemset
17     if I = {} then
18         node ← rootEntry
19     else
20         node ← insertOrGet(I)
21     F ← node.solution.feat
22     while F ≠ NO_FEAT do
23         for items i and ¬i known as splits of feature F do
24             exploreTree(I ∪ i)
25             exploreTree(I ∪ ¬i)
```

would hold the optimal subtree found as the best for each itemset. Notice in the ***BestTree*** structure that there is no parameter to hold the subtree found as solution. Instead, a single integer variable is used to hold the feature representing the root of the best subtree found for the itemset. This reduces the impact of the solution storage on the memory consumption of the algorithm. It is initialized by default for a new itemset to the value `NO_FEAT`. This states that it is a new itemset and that no solution has yet been found for it. The same variable is used to store the class associated to the itemsets of leaf nodes. As the subtree found as solution for each itemset is not saved in the solution data structure, DL8 proposes the `exploreTree()` function to explore in the cache, the nodes of the optimal tree found at the end of the search. Starting from the root itemset, the corresponding node is found in the trie (lines 17-20).

Once the node is found, the feature representing the root of its optimal subtree is selected (line 21). As long as the feature exists, a DFS is performed to recursively reach the optimal nodes representing the left and right child nodes (lines 22-25). Note that this function can be used for instance to print a string representation of the final tree.

4.3.5 Pseudocode of DL8

Pseudocode of the DL8 algorithm is given in Algorithm 6. Essentially, the algorithm recursively enumerates itemsets using the DL8−Recurse(I) function. The postcondition 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. However, using the caching structure of DL8, only the root of this decision tree is returned instead of the complete tree.

Algorithm 6: DL8(maxdepth, minsup)

```
1  cache ← Trie_Cache()
2  DL8−Recurse({})
3  return cache.exploreTree({})
4  Procedure DL8−Recurse(I : array of int)
       // add to the cache or get from it if it already exists
5      node ← cache.insertOrGet(sort(I))
6      solution ← node.solution
7      if solution.feat ≠ NO_FEAT then
8          return solution
9      if solution.error = 0 or |I| = maxdepth then
10         solution.feat ← maxclass(I)
11         return solution
12     for each feature F splittable in items i and ¬i do
13         if |cover(I ∪ {i})| < minsup or |cover(I ∪ {¬i})| < minsup then
14             continue
15         sol_1 ← DL8−Recurse(I ∪ {¬i})
16         if sol_1.error < solution.error then
17             sol_2 ← DL8−Recurse(I ∪ {i})
18             if sol_1.error + sol_2.error < solution.error then
19                 solution.feat ← F
20                 solution.error ← sol_1.error + sol_2.error
21     return solution
```

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.

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. This means that the path does not fulfill the requirement of the minimum number of instances per leaf. In our running example, the itemset $\{a, b, c\}$ is not considered due to this optimization because its support is 0;

Purity pruning In line 9 the search is stopped if the error for the current itemset is already 0. If the classification error committed by a path is null, there is no need to split it more;

Quality bounds In the loop of lines 12–20, 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 these in detail here for reasons of simplicity.

The most important optimization in DL8 that we do not use in this chapter is the *closed itemset mining* optimization. This optimization is related to the representation used to represent the itemsets stored in the cache of the algorithm. In addition to the sorted form of items, DL8 proposes to represent an itemset in the cache by the closed itemset of its cover. Doing this, all the paths that cover the same subset of instances are uniquely represented. This optimization of DL8 leads to a high number of solution reuses in the cache. However, we do not use it in our approach. The reason for this choice is that this optimization is not trivial to be combined with a constraint on the maximum depth of a decision tree. Similarly, while DL8 can be applied to other scoring functions than misclassification error, as long as the scoring function is *additive*, we prioritize in this chapter the accuracy and the depth and support constraints here as we focus on solving the same problem as in recent studies. The use of our algorithm to solve other problems is treated in another chapter.

4.4 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 4.2, 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 branch-and-bound search can reduce the number of itemsets considered significantly.

Algorithm 7: DL8.5(maxdepth, minsup)

```
cache ← Trie_Cache()
DL8.5−Recurse({}, +∞)
return cache.exploreTree({})
Procedure DL8.5−Recurse(I : array of int, ub : int)
    // add to cache or get if exists
    node ← cache.insertOrGet(sort(I))
    solution ← node.solution
    if solution.feat ≠ NO_FEAT or ub ≤ solution.lb or ((|I| = maxdepth or time-out is reached) and solution.error > ub) then
        return solution
    if solution.error = solution.lb or ((|I| = maxdepth or time-out is reached) and solution.error ≤ ub) then
        solution.feat ← maxclass(I)
        return solution
    for each feature F in a well-chosen order and splittable in items i and ¬i do
        if $|cover(I ∪ \{i\})| < minsup$ or $|cover(I ∪ \{¬i\})| < minsup$ then
            continue
        sol_1 ← DL8.5−Recurse(I ∪ {¬i}, ub)
        if sol_1.feat = NO_FEAT then
            continue
        sol_2 ← DL8.5−Recurse(I ∪ {i}, ub − sol_1.error)
        if sol_2.feat = NO_FEAT then
            continue
        solution.feat ← F
        solution.error, ub ← sol_1.error + sol_2.error
        if solution.error = solution.lb then
            break
    if solution.feat = NO_FEAT and solution.lb < ub then
        solution.lb ← ub
    return solution
```

The pseudo-code of our new algorithm, DL8.5, is presented in Algorithm 7. 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.

Hierarchical upper bound In DL8.5, the recursive `DL8.5−Recurse` procedure has an additional parameter, `ub`, which represents an upper-bound on the quality of the decision trees that the recursive procedure is expected to find. If no tree sufficiently better than the upper bound can be found, the procedure returns the value `NO_FEAT` as subtree root to express that there is no solution. Initially, the upper-bound that is used is $+\infty$ (line 2). 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 22, is used as upper-bound for future decision trees and is communicated to the children in the search tree (line 15, 18). This is why this upper bound is called hierarchical upper bound. The upper-bound is used to prune the search space using a test in line 16; 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. Concretely, the satisfaction of the error produced against the upper bound is really assessed at leaf nodes, i.e. $|I| =$ `maxdepth`. When the error produced at leaf node does not satisfy the upper bound constraint (lines 7-8), the default value (`NO_FEAT`) set as solution is returned and sent back to ancestor nodes. Otherwise, the class value is set (lines 9-11). In line 18 we use the quality of the first branch to bound the required quality of the second branch further.

Infeasibility lower bound An important modification involves the interaction of the bounds with the cache. In DL8, a solution is always found and saved for an itemset. Therefore, when the same itemset is encountered through another path in the search space, its solution is returned. The special value `NO_FEAT` thus indicates in DL8 that it is the first time an itemset is encountered. However, in DL8.5, because of the upper bound, it can happen that there is no solution for an itemset fulfilling the upper bound constraint. Consequently, in addition to new itemsets, the value `NO_FEAT` also indicates in DL8.5 the already assessed itemsets for which the solution is not found. A naive approach to handle these itemsets would consist of an automatic re-computation of the solution of all them because they do not have a solution. However, in

DL8.5, we treat differently these itemsets because they provide important information on the search. The first time these itemsets have been assessed without finding the solution, that was because the upper bound provided to the call of `DL8.5−Recurse` is more restrictive than the actual solution. Based on this information, it can be concluded that the actual best error is at least equal to the restrictive upper bound provided. This upper bound is thus used as a lower bound (lines 25-26). The benefit of this method is that at a later moment, we can reuse the fact that for a given itemset already evaluated without having found its solution, no sufficiently good decision tree can be found, without needing to perform a new calculation. In particular, in line 7, when the upper bound is lower than or equal to the inferred lower bound. In the same way, when during the assessment of an itemset, its error is equal to the lower bound, then the best solution is found for this itemset. In this case, there is no need to continue the evaluation of other attributes (lines 9-11 and lines 23-24).

Other differences from DL8 Other modifications in comparison with DL8 improve the behavior of the algorithm. In lines 7 and 9 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. Another difference is the number of different heuristics considered when branching on features. In our experiments, we consider three: the original alphabetic order of the attributes in the data, in increasing and finally in decreasing order of information gain (such as used in C4.5 and CART).

Our modifications of DL8 drastically improve the pruning of the search space. Figure 4.2 indicates which nodes are pruned during the execution of DL8.5 (for an alphabetic order of the attributes). At the end, 17 nodes over 27 are visited instead of 26 over 27 for DL8.

4.4.1 Illustration of the hierarchical upper bound

Figure 4.3 shows a part of the execution of DL8.5 in more detail to illustrate the hierarchical upper bound. The initial value of the upper-bound at node ϕ is $+\infty$ (line 2). The attribute A provides an error of 2; the upper-bound value is subsequently updated from $+\infty$ to 2 in line 22. In the first branch for attribute C, the new value of the upper-bound is passed down recursively (line 15). Notice that the initial value of the upper-bound at node $\neg c$ is 2. At this node, the attribute A is first visited and provides an error of 1 by summing errors of $\neg a \neg c$ and $a \neg c$

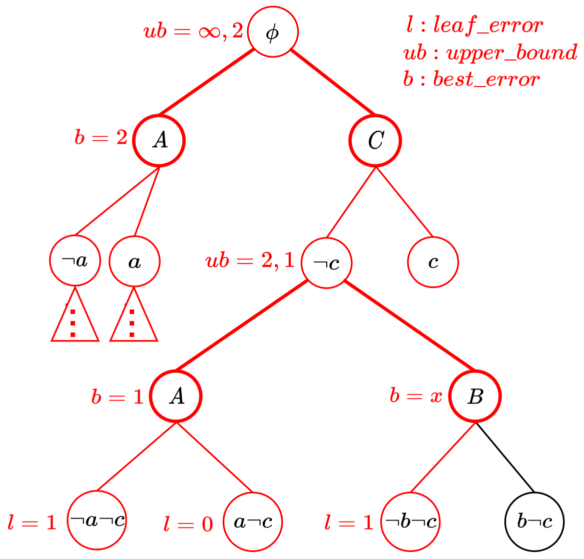


Figure 4.3: Example of pruning

(line 22). The upper-bound for subsequent attributes is then updated to 1 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 not lower than the upper-bound of 1. The second item is pruned as the condition of line 16 is satisfied. So, there is no solution by selecting the attribute B, which leads to keep unchanged the value NO_FEAT for this itemset. This error value is represented in Figures 4.2 and 4.3 by the character x.

The reuse of the cache is illustrated for itemset $\neg ac$ (Figure 4.2). The first time we encounter this itemset, we do so coming from the itemset $\neg a$ for an upper-bound of 1; after the first branch, we observe that no solution can be found for this bound, and we store NO_FEAT for this itemset and a lower vound of 1. 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 because it is not greater than the lower bound, and we skip attribute A from further consideration (lines 7-8).

4.4.2 Efficient counting of supports

As explained above, the DL8.5 algorithm performs an exploration of relevant paths that could be part of the final ODT. It relies on the branch-and-bound to avoid exploring a considerable number of paths. However, the assessment of the remaining relies on the observation of the error they produce. The computation of these errors is one of the major time-consuming part of the algorithm. In the context of misclassification rate, this error computation is based on the supports per class. Therefore, an efficient technique to count the supports would result in a reduction of the run time of the algorithm.

In the implementation of DL8 algorithm, the supports per class of each path is based on a simple loop over the database to identify the instances covered by the path. Obviously, the part of the database to explore is reduced as long as we go deep in the search space. However, this way of counting of supports is perfectible.

At first, in our algorithm, we rely on the same technique as in DL8 but in addition to this technique, we experiment another one. It is the support counting based on the Reversible Sparse Bitset (RSBS) introduced in Chapter 3. Indeed, this technique has been used in [SAG17] to propose a frequent itemset algorithm and has shown interesting results. As explained in Chapter 3, the RSBS data structure is very efficient to handle the problems of number of valid elements counting.

4.5 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 impact of different branching heuristics on the performance of DL8.5?

Q3 What is the impact of the infeasibility lower bound in DL8.5?

Q4 How does the RSBS data structure impact the run time of DL8.5?

Q5 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 [VZ19], as it was shown to be the best performing MIP-based approach at the moment we were performing this study. The implementations of BinOCT[1], DL8 and the CP-based approach[2] used in our comparison were obtained from their original authors, and we use the CPLEX 12.9[3] and OscaR[4] solvers.

[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

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 constraint of 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 4.5.

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 support of 1 since this is the setting used in BinOCT.

We do not split our 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 [BD17].

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
no LB	d.o. is used and infeasibility lower bound is disabled
RSBS	d.o. is used and reversible sparse bitset is used to count supports

Table 4.5 shows the results for a maximum depth equal to 4, as we consider deeper decision trees of computationally more interest. The first three columns describe respectively the names of the datasets as well as the number of features and the number of transactions in the datasets. The remaining columns represent the different algorithms considered. Each of them is described by two columns representing respectively the objective value and the execution time. 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 when the time limit imposed is reached. The best solutions and best times are marked in bold while a star (*) is added to mark solutions proven to be optimal.

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

Table 4.5: Comparison table for binary datasets with max depth = 4

Dataset	nFeat	nTrans	BinOCT		DL8		CP-Based		DL8.5									
									d.o.		Asc		Desc		no LB		RSBS	
			obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)
anneal	93	812	115	*TO*	∞	*TO*	**91***	450.69	**91***	129.24	**91***	127.45	**91***	121.87	**91***	250.64	**91***	**33.11**
audiology	148	216	2	*TO*	∞	*TO*	**1**	*TO*	**1***	180.84	**1***	204.19	**1***	195.73	**1**	*TO*	**1***	**176.20**
australian-credit	125	653	82	*TO*	∞	*TO*	66	*TO*	**56***	566.71	**56***	586.39	**56***	593.38	57	*TO*	**56***	**187.09**
breast-wisconsin	120	683	12	*TO*	∞	*TO*	8	*TO*	**7***	305.3	**7***	325.7	**7***	330.71	**7**	*TO*	**7***	**128.80**
diabetes	112	768	170	*TO*	∞	*TO*	140	*TO*	**137***	553.49	**137***	562.83	**137***	565.5	**137**	*TO*	**137***	**179.57**
german-credit	112	1000	223	*TO*	∞	*TO*	**204**	*TO*	**204***	558.73	206	*TO*	**204***	599.87	**204**	*TO*	**204***	**153.20**
heart-cleveland	95	296	39	*TO*	∞	*TO*	**25**	*TO*	**25***	124.1	**25***	130.3	**25***	132.23	**25***	214.76	**25***	**59.78**
hepatitis	68	137	7	*TO*	**3***	66.62	**3***	109.36	**3***	13.46	**3***	14.06	**3***	14.88	**3***	27.28	**3***	**8.76**
hypothyroid	88	3247	55	*TO*	∞	*TO*	**53**	*TO*	**53***	392.22	**53***	368.95	**53***	427.34	**53**	*TO*	**53***	**43.66**
ionosphere	445	351	27	*TO*	∞	*TO*	20	*TO*	17	*TO*	11	*TO*	13	*TO*	17	*TO*	**10**	*TO*
kr-vs-kp	73	3196	193	*TO*	∞	*TO*	**144***	483.15	**144***	216.11	**144***	206.18	**144***	223.85	**144***	528.72	**144***	**25.07**
letter	224	20000	813	*TO*	∞	*TO*	574	*TO*	550	*TO*	586	*TO*	802	*TO*	550	*TO*	**335**	*TO*
lymph	68	148	6	*TO*	**3***	56.29	**3***	112.48	**3***	8.7	**3***	11.03	**3***	8.47	**3***	25.04	**3***	**5.95**
mushroom	119	8124	278	*TO*	∞	*TO*	**0***	352.18	**0***	331.39	4	*TO*	**0***	**0.11**	4	*TO*	**0***	9.94
pendigits	216	7494	780	*TO*	∞	*TO*	38	*TO*	32	*TO*	26	*TO*	**14**	*TO*	32	*TO*	24	*TO*
primary-tumor	31	336	37	*TO*	**34***	2.79	**34***	8.96	**34***	1.48	**34***	1.51	**34***	1.38	**34***	2.43	**91***	**0.5**
segment	235	2310	13	*TO*	∞	*TO*	**0***	128.25	**0***	3.54	**0***	6.99	**0***	7.05	**0***	3.54	**0***	**0.33**
soybean	50	630	15	*TO*	**14***	41.59	**14***	40.13	**14***	5.7	**14***	6.34	**14***	5.75	**14***	18.41	**14***	**1.75**
splice-1	287	319	574	*TO*	∞	*TO*	∞	*TO*	224	*TO*	224	*TO*	**141**	*TO*	224	*TO*	**212**	*TO*
tic-tac-toe	27	958	180	*TO*	**137***	3.76	**137***	9.17	**137***	1.43	**137***	1.54	**137***	1.55	**137***	2.12	**137***	**0.32**
vehicle	252	846	61	*TO*	∞	*TO*	22	*TO*	16	*TO*	18	*TO*	**13**	*TO*	16	*TO*	**13**	*TO*
vote	48	435	6	*TO*	**5***	29.84	**5***	44.47	**5***	5.48	**5***	5.33	**5***	5.58	**5***	12.82	**5***	**2.1**
yeast	89	1484	395	*TO*	∞	*TO*	**366**	*TO*	**366***	318.87	**366***	326.2	**366***	334.15	**366***	470.88	**366***	**81.92**
zoo-1	36	101	**0***	0.52	**0***	1.11	**0***	0.2	**0***	0.01	**0***	0.01	**0***	0.01	**0***	0.01	**0***	**0.001**

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 4.4, 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 that of DL8, BinOCT and the CP-based algorithm, obtaining orders of magnitude better performance.

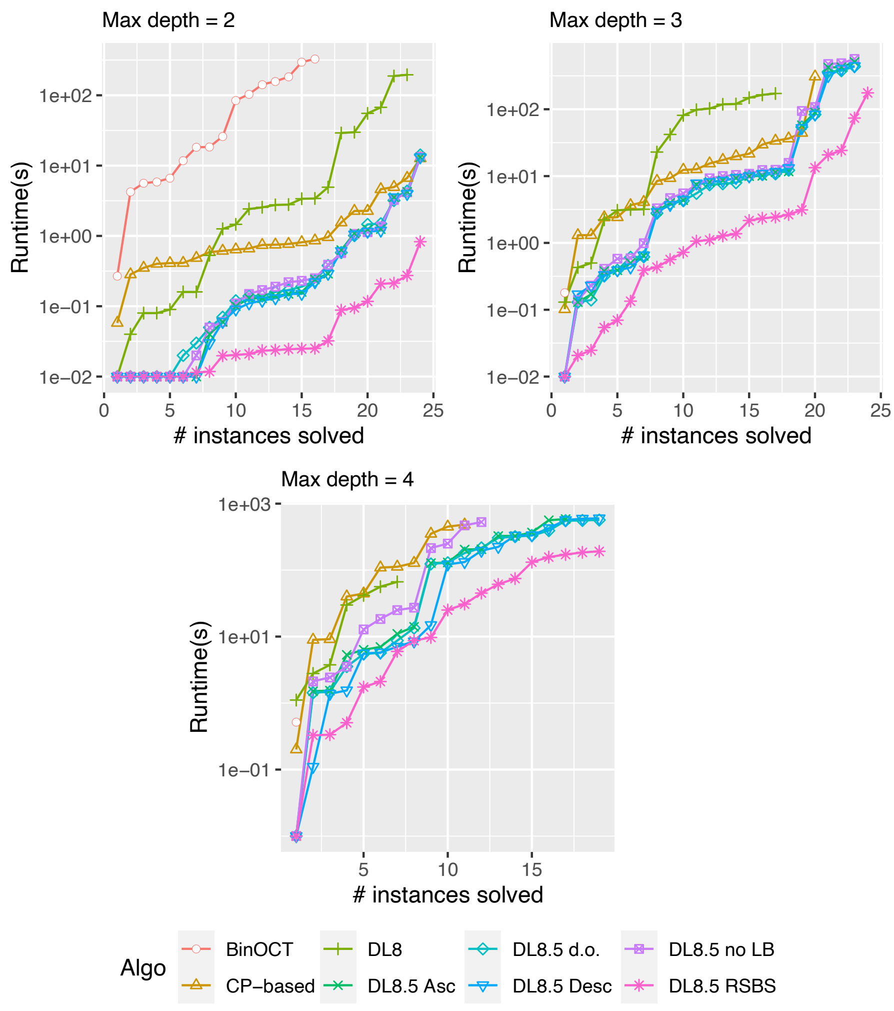


Figure 4.4: Cumulative number of instances solved over time

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 use the infeasibility lower bound, we see a significant degradation in performance. In this variant (no LB), only 12 instances are solved optimally, instead of 19, for a depth of 4. Hence, this optimization is significant (**Q3**).

Regarding the use of a RSBS data structure to perform the support counting, it can be noticed in the figure that this variant of DL8.5 is the most efficient. This can be confirmed in Table 4.5. The implementation of DL8.5 RSBS that is presented here does not use any branching heuristic to sort the attributes. For this, its results as well as the results of DL8.5 d.o. which does not use a branching heuristics are highlighted in the table to facilitate their comparison. The best approach is underlined. Notice that DL8.5 RSBS has always the best performance, even in comparison with the other variants on most datasets (**Q4**). Based on this conclusion, the use of RSBS in DL8.5 is provided as the standard support counting technique and is enabled each time we refer to DL8.5 in the following.

To answer **Q5**, 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 4.6. Before running DL8, the CP-based algorithm and DL8.5, we binarize these datasets by creating binary features using the thresholds splits of numerical features explained in Subsection 1.3.1. Note that the number of generated features is very high in this case. As a result, for most datasets most of the algorithms reach a time-out for maximum depths of 3 and 4, as shown by Verwer and Zhang (2019). Hence, we focus on results for a depth of 2 in Table 4.6. 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.

4.6 Conclusions

In this chapter, we presented the DL8.5 algorithm for learning optimal decision trees. DL8.5 is based on a number of ideas: the use of itemsets

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

Table 4.6: Comparison table for continuous datasets with max depth = 2

Dataset	nTrans	nFeat	nBinFeat	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	21	**49***	1.64	**49***	0.01	**49***	0.29	**49***	0.004	**49***	**0.001**	**49***	0.003
banknote	1372	4	5021	**606**	*TO*	∞	*TO*	**606***	60.99	**606***	**14.73**	**606***	16.83	**606***	29.74
bank	4521	51	3763	484	*TO*	∞	*TO*	**477***	50.31	**477***	**16.74**	**477***	18.5	**477***	21.2
biodeg	1055	41	8304	322	*TO*	∞	*TO*	**248***	164	**248***	**38.5**	**248***	58.38	**248***	101.3
car	1728	6	21	**250***	2.49	**250***	**0.03**	**250***	0.32	**250***	**0.006**	**250***	0.007	**250***	0.008
IndiansDiabetes	768	8	1255	251	*TO*	∞	*TO*	**250***	4.54	**250***	**0.73**	**250***	0.83	**250***	0.89
Ionosphere	351	34	8149	50	*TO*	∞	*TO*	**48***	139.22	**48***	**28**	**48***	32.07	**48***	136.11
iris	150	4	124	**14***	4.62	**14***	**0.57**	**14***	0.41	**14***	**0.004**	**14***	0.01	**14***	0.01
letter	20000	16	257	603	*TO*	∞	*TO*	**586***	3.03	**586***	**0.56**	**586***	0.61	**586***	**0.56**
messidor	1151	19	9390	514	*TO*	∞	*TO*	**476***	174.59	**476***	**52.1**	**476***	62.41	**476***	174.47
monk1	124	6	18	**22***	0.57	**22***	0.004	**22***	0.23	**22***	**0.001**	**22***	**0.001**	**22***	**0.001**
monk2	169	6	18	**57***	2.51	**57***	0.004	**57***	0.23	**57***	**0.001**	**57***	**0.001**	**57***	**0.001**
monk3	122	6	18	**8***	0.37	**8***	0.03	**8***	0.28	**8***	0.001	**8***	0.0007	**8***	**0.0005**
seismic	2584	18	4285	169	*TO*	∞	*TO*	**165***	53.74	**165***	**14.62**	**165***	16.91	**165***	24.6
spambase	4601	57	15095	660	*TO*	∞	*TO*	**609**	*TO*	**609***	**250.37**	**609***	513.41	**609**	*TO*
Statlog	4435	36	2747	1035	*TO*	∞	*TO*	**950***	37.18	**950***	**8.63**	**950***	12.27	**950***	15.59
tic-tac-toe	958	18	37	**282***	5.95	**282***	0.08	**282***	0.34	**282***	0.01	**282***	**0.004**	**282***	0.009
wine	178	13	1277	**48**	*TO*	∞	*TO*	**48***	4.17	**48***	**0.53**	**48***	0.66	**48***	0.85

to represent paths, the use of a cache to store intermediate results, the use of bounds to prune the search space, the use of RSBS to efficiently count supports, the ability to use heuristics during the search, and the ability to return a result even when a time-out is reached.

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

In this chapter, 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, in the following chapters, we will show that DL8.5 can be modified for use in other constraint-based decision tree learning problems, using ideas from DL8 [NF10].

Learning ODTs under memory constraints

5

This chapter is based on the paper G. Aglin, S. Nijssen, and P. Schaus. “Learning Optimal Decision Trees Under Memory Constraints”. The paper is accepted at the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD) 2022 but is not yet published at the moment this thesis is written.

5.1 Introduction

As we saw in the previous chapter, approaches for finding ODTs can be put into two categories: (1) the category of approaches based on the use of mixed integer programming (MIP) solvers [VZ19; AGV21], and (2) the category of specialized search algorithms that use some form of *dynamic programming* (DP) such as DL8 and DL8.5 [NF07; NF10; ANS20; Ver+20; HRS19; Lin+20; Dem+22]. Among these two categories, we saw that DP approaches are significantly faster. Unfortunately, however, an important drawback of the DP approaches is their high memory consumption: in the worst case, their memory consumption is exponential in the number of features. MIP approaches, at least theoretically, do not have this weakness.

The key aim of this chapter is to address this situation. We propose a new DP-based approach that allows a user to determine the trade-off between time and space, by introducing a parameter that determines the amount of memory the algorithm is allowed to use.

The intuition for the good run time performance of DP approaches is that these approaches do not perform a search for *trees*, but a search for *paths*. As the number of paths is smaller than the number of trees, this reduces the size of search space in DP algorithms significantly. Their high memory consumption derives from the fact that the DP-based algorithms store information about paths in a *cache*, from which they construct ODTs.

The contributions of this chapter are the following:

C1 We study several caching strategies proposed in the literature, identifying which has the best characteristics when focusing on memory consumption.

C2 We propose a simple modification of existing DP-based approaches that amounts to removing, from time to time, elements from the cache if the cache becomes too large.

However, removing elements from the cache can have two undesirable consequences: (1) an ODT can no longer be recovered from the cache, and (2) search can become slower, as we can no longer use results in the cache. Our two next contributions address these critical problems:

C3 We propose strategies for the order in which elements are deleted from the cache.

C4 We present a strategy to recover deleted elements if these are required to output the ODT.

In the resulting approach, a parameter determines the trade-off between run time and memory consumption. Our final contribution (**C5**) is an experimental evaluation of this trade-off, as well as other dimensions important for the memory consumption of DP-based algorithms.

This chapter is structured as follows. Section 5.2 presents the state of the art of different caching architectures used for ODTs learning and some search techniques used to reduce memory consumption. Then, Section 5.3 presents the technical details of caching systems implemented as a trie and a hash table. Section 5.4 introduces our new size-limited cache. Finally, we present experimental results before concluding.

5.2 Related Work

There are two classes of approaches for finding ODTs: (1) approaches that rely on solvers, such as MIP solvers [VZ19; AGV21], and (2) approaches that rely on dynamic programming [NF07; NF10; ANS20; Ver+20; HRS19; Lin+20; Dem+22]. The first class of methods relies on solvers of which the memory use is bounded; however, recent studies on DP-based approaches showed that the run time performance of these methods is significantly better. On the other hand, they suffer from a memory problem related to the use of a cache. Indeed, the size of the cache increases with the number of features and the depth of the ODT that needs to be found. For these algorithms it is hence important to minimize the amount of memory used. In the literature, two types of memory-based optimization have been studied. The first one focuses on the caching system itself and concerns the data structure used to implement the cache and to represent the stored elements. The second optimization is related to the search, and mainly consists in reducing the number of cache entries that are stored. Below we provide a high-level perspective on these optimizations; their details will be discussed in the next section.

5.2.1 Caching optimizations

As stated in the previous chapter, some form of exhaustive search needs to be performed to learn an ODT. A core idea underlying the DP-based approaches is to store intermediate results during this exhaustive search. Using a cache key, these intermediate results are then used later on to avoid repeating the same search twice.

A major distinguishing factor between various approaches is the key that is used to associate intermediate results in the cache. The first algorithm to take this approach was DL8 [NF07; NF10]. In DL8, a cache is built in which *itemsets* serve as the key to which information is associated. Here, every path in a decision tree corresponds to an itemset: an itemset is essentially the set of conditions (tests) on a path. For trees of limited depths, these keys are short. This technique is also used in DL8.5, and works in the presence of depth constraints.

An optimization was presented in DL8, in which *closed itemsets* were used. A closed itemset in this context is obtained by adding to an itemset I all other conditions that hold in the same instances in which the conditions of I are true. It was shown that this leads to more cache reuse; however, calculating this key takes more time and the key cannot trivially be used in the presence of depth constraints, consequently, this optimization was not used in DL8.5.

Keys based on instances were used in the GOSDT [Lin+20] algorithm. Moreover, GOSDT relies on the use of hash tables to represent the cache structure; it uses bit vectors to represent the sets of identifiers. DL8 and DL8.5, on the other hand, use a *trie*, aiming to exploit the overlap between itemsets that are stored in the cache.

None of these systems allow the user to impose a limit on the size of the cache.

5.2.2 Search optimization

The higher the number of elements to be stored, the larger the cache. Hence it is important to minimize the number of elements that need to be stored as much as possible. Various improvements have been studied to reduce the size of the search space and hence the number of elements in the cache.

One core idea is to use a form of branch-and-bound search to limit which parts of the search need to be considered. We propose in Chapter 4 a hierarchical upper bound and an infeasibility lower bound to avoid exploring some nodes of a search tree over paths. [Lin+20] proposes a similarity lower bound. This bound is inspired by [Ang+17].

Using better bounds can effectively reduce the number of elements that need to be stored in the cache, and hence can have an impact on the memory used by the algorithm. However, while better bounds can make the search more feasible for larger datasets, they do not resolve the problem that, for some datasets, the search algorithms will run out of memory.

5.2.3 MurTree algorithm

After our work on DL8.5 was published, other researchers started working on different ways to improve the algorithm. To accomplish this task, the recursive character and the caching of solutions present in the DL8.5 algorithm are reused. Similarly, the branch-and-bound aspect is also reused. MurTree [Dem+22] is a recent algorithm that relies on DL8.5 and proposes some interesting improvements. Here is an overview of major improvements proposed by the MurTree algorithm.

Strong infeasibility lower bound The infeasibility lower bound used in DL8.5 is found when an upper bound is too restrictive to find a solution. The upper bound is therefore used as the minimum value of the expected error without taking into account the information found in the part of the search space explored before noticing the infeasibility. MurTree proposes a strong infeasibility lower bound that uses this information. Specifically, for each attribute explored, the error or the lower bound (when the error is not found) of its items are summed to compute the attribute lower bound. Then, in case of infeasibility, the minimum of all attribute lower bounds is used as the infeasibility lower bound. This value is guaranteed to be always better than using the restrictive upper bound.

Similarity lower bound This similarity lower bound used in MurTree is related to the one proposed in GOSDT. It is computed for a node based on the error already computed for a similar node. The similarity between two nodes is defined by the fact that the intersection of the set of instances covered by the two nodes is not an empty set. To compute the similarity lower bound of a node n_2 covering a set $\mathcal{D}_2$ using the error computed for a node n_1 covering $\mathcal{D}_1$, MurTree proves that the value $lb = error(n_1) - |\mathcal{D}_{out}|$ is a good candidate, where $\mathcal{D}_{out}$ is the set of instances in $\mathcal{D}_1$ and not in $\mathcal{D}_2$. This value is computed for each node and the best with respect to the infeasibility lower bound is used as the lower bound of the node.

Dynamic branching In the DL8.5 algorithm, different combinations of attributes are explored to evaluate the paths in the search space. For each attribute, two tests corresponding to items are evaluated. These two tests are considered one after the other in a fixed order. MurTree proposes a dynamic approach to decide the order in which to explore the two items produced by each attribute. Concretely, MurTree compares the real error or the pessimistic error (lower bound) of both items and decides to explore at first the item producing the higher lower bound. The idea is that the item with the higher error is more likely to produce a high error that may violate the upper bound constraint and lead to a pruning of the second.

Specialized algorithm for ODTs of depth 2 This feature is one of the main improvements of MurTree. The idea of this specialized algorithm relies on the reduction of the number of supports needed to be computed to find an ODT of depth 2. In fact, an exhaustive search for an ODT of depth 2 requires computing for each class, the support of each item and each pair of items. In MurTree, only a few of these supports are computed. The others are derived from the few computed. For instances, for any attributes I and J splittable respectively in $i, \neg i$ and $j, \neg j$, MurTree proves that the computation of supports of itemsets i and ij are sufficient to derive the support of $\neg i, i\neg j, \neg ij, \neg i\neg j$. As shown in the previous chapter, counting the supports is one of the major time-consuming operations of the DL8.5 algorithm. By using this technique, it reduces significantly the time needed to find an ODT of depth 2. Since this function is called often in the algorithm, it has a significant impact on reducing the run time.

Another important difference between MurTree and DL8.5 algorithm is the data structure used to implement the cache structure and the representation used as key for cache entries. In MurTree algorithm, the cache is not implemented as a trie. On the contrary, it is implemented as a hash table as in GOSDT. Regarding the representation used as key for the cache entries, MurTree proposes to use a combination of instance identifiers and path length (of which more details will be provided in the next section).

The various improvements proposed by MurTree lead to a reduction in the memory consumption of the algorithm. However, no technique for limiting the cache size is proposed. In this chapter, the memory reduction technique proposed is not a search algorithm optimization. Rather, we propose an improvement to the cache system of DP-based algorithms that allows to set an upper bound on the maximum number of cache entries. Hence, our optimizations are orthogonal to possible bounds used during the search.

5.3 Caching in DP-based ODT learning

In this section, we describe how the trie and hash table are used to cache subproblem solutions when learning ODTs in existing algorithms; while doing so, we also contribute to a comparison that will motivate our choices in this chapter.

5.3.1 Caching in DP-based algorithms

Let $\mathcal{D} = \{(\boldsymbol{x}, y)\}^n$ be a binary dataset of n instances and $\mathcal{F} = \{F_1, \ldots, F_m\}$ be the set of m features that describe $\mathcal{D}$. For each instance $\boldsymbol{x} = (x_1, \ldots, x_m)$ in $\mathcal{D}$, x_i takes value in $\{0, 1\}$. Because the approach of this chapter is generic to all DP-based algorithms, the paths are not mapped into itemsets like in Chapter 4. A decision tree recursively partitions a dataset into different groups following paths $p \in \mathcal{P}$. A decision tree can be seen as a collection $\mathcal{DT} \subseteq \mathcal{P}$ of paths, where each path starts at the root of the tree. While in decision tree, features are tested in a given order, it is the set of tests that determines which instances end up in a given node in the decision tree. For this reason, we will see a path as a *set* of tests on features. In the context of binary decision trees, a path p is a set of tests over binary features, $p \subseteq \bigcup_{F \in \mathcal{F}} tests(F)$, where $tests(F)$ returns the two possible tests for the feature F, $F = 1$ (abbreviated with f) and $F = 0$ (abbreviated with $\overline{f}$); we assume $|p \cap tests(F)| \leq 1$ for all F.

At a high level, dynamic programming-based approaches for solving the ODT problem are based on recursive equations. For approaches that use itemsets as keys, this is the recursive equation in its simplest form:

$$min\_error(p) = \begin{cases} \min_{F \in \mathcal{F}} \sum_{t \in tests(F)} min\_error(p \cup \{t\}) & \text{if } |p| < maxdepth; \\ leaf\_error(p) & \text{if } |p| = maxdepth, \end{cases} \tag{5.1}$$

where the recursion starts at $min\_error(\emptyset)$. In other words, to determine the error made by a decision tree of minimal error, we need to pick the feature in the root of the tree that minimizes error, when summing up the lowest possible errors for the left-hand and right-hand subtrees. Note that in a tree the tests are ordered. We can first test $A = 1$ followed by $B = 1$, or, alternatively, first test $B = 1$ and then $A = 1$. This order is not important when determining $min\_error(\{A = 1, B = 1\})$. DP approaches are based on the idea of storing information for p, such as $min\_error(p)$, so that we can reuse the information for all possible orders in which the tests can be put in a tree.

For the leafs of the tree, we assume a prediction is made, and a class is associated. In the case of our thesis, the associated class is the majority class (eq. 4.2), while the associated error is the misclassification rate defined by $leaf\_error(p) = |\mathcal{D}| - max_{c\in\mathcal{C}}|\mathcal{D}_c|$, where $\mathcal{D}$ is the set of instances falling in the leaf p and $\mathcal{D}_c$ is the subset of $\mathcal{D}$ associated with the class c.

In its more complex form, the recursive equation can take into account other constraints, and can be rephrased to return the optimal tree itself as well.

DP-based algorithms perform a depth-first search using the recursive equation, reusing information that is stored already, and using bounds to limit the cases for which the recursion is executed.

For approaches that use instances as keys, the recursive equation is slightly different:

$$min\_error(\mathcal{D}, d) = \begin{cases} \min_{F\in\mathcal{F}} \sum_{t\in tests(F)} min\_error(\sigma_t(\mathcal{D}), d+1) & \text{if } d < maxdepth; \\ leaf\_error(\mathcal{D}) & \text{if } d = maxdepth, \end{cases} \tag{5.2}$$

where $\mathcal{D}$ is a dataset, $\sigma_t(\mathcal{D})$ selects the instances of $\mathcal{D}$ which satisfy the condition in test t and d is the number of conditions in the path which is being considered and which covers $\mathcal{D}$. By using itemsets, different paths relying on the same tests are uniquely represented, and the error computed for one can be used for another. However, there are more paths having the same solution than those relying on the same tests. In fact, all the paths that cover the same set of instances have the same solution, as long the number of conditions to add to the paths to reach the maximum depth is the same. It is this property that is used in the recursive equation $min\_error(\mathcal{D}, d)$. The recursion starts for the full dataset at depth $d = 0$. In other words, to determine the error of an optimal tree for a dataset $\mathcal{D}$, we need to determine which test to put in the root of the tree, such that error for the datasets resulting from the split is minimal. Compared to using paths as keys, instances can allow for more reuse, as multiple paths may select the same set of instances.

In this chapter, our aim is to strictly monitor the memory consumption of DP-based algorithms. Hence it is important to understand which one of these two approaches leads to better memory use, as earlier studies did not address this question. We study this in the next subsection.

5.3.2 Comparison of Caching Strategies

Figure 5.1 shows a comparison of the memory consumption for different cache implementations for some datasets, when implemented in DL8.5 algorithm. The red curves represent the cache implementation that uses

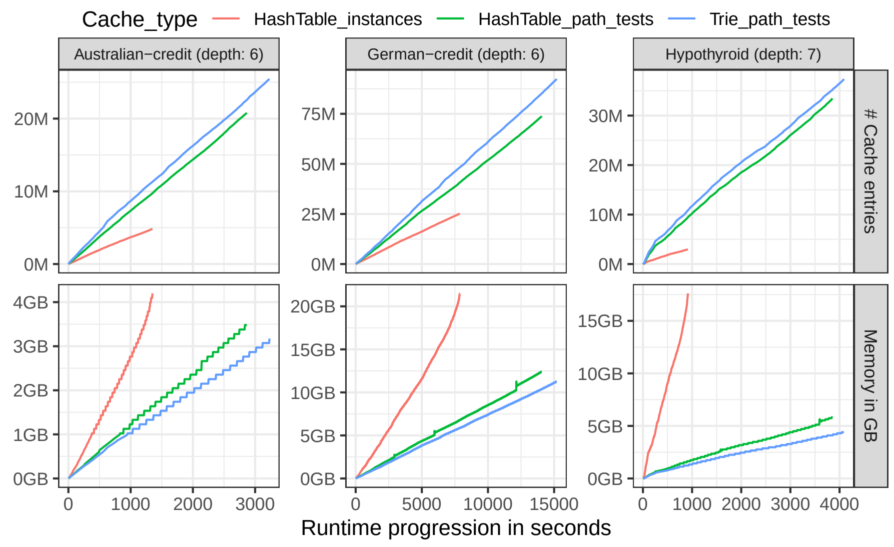


Figure 5.1: Comparison of memory use given the cache type

the set of instances as key. The other curves denote the cache implementations that use paths as key. The difference between the green and blue curves will be explained in the next subsection. Notice that the number of cache entries for the instance-based representation is indeed low compared to the other approach. At the same time, it consumes much more memory as a key that consists of instance identifiers is much longer than a key that consists of path tests. The rest of the paper focuses on the test-based representation, as it requires less memory than the instance-based one.

5.3.3 Caching data structure

To store the cache, DP-based algorithms require a data structure. Two implementations of the cache have been used in the literature. GOSDT and MurTree algorithms use a hash table, while DL8 and DL8.5 use a *trie* (or prefix tree).

The difference between these two data structures is illustrated in the following example, where a path is used as key; here we sort the tests in the path to obtain an ordered representation for the path. Let us assume

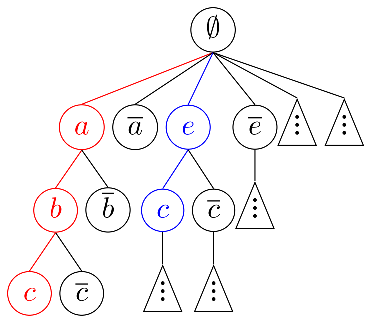


Figure 5.2: Search space

Table 5.1: Hash table storage

$\emptyset$		⁂	
a	⁂	$\overline{a}$	⁂
e	⁂	$\overline{e}$	⁂
ab	⁂	$a\overline{b}$	⁂
ce	⁂	$\overline{c}e$	⁂
abc	⁂	$ab\overline{c}$	⁂

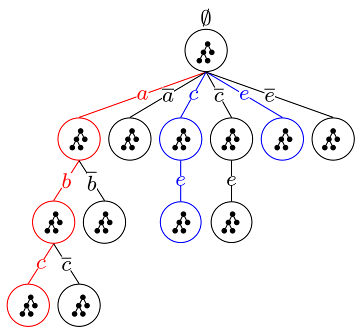


Figure 5.3: Trie storage

a dataset with at least 4 features and an ODT learning algorithm for trees of depth 3. Figure 5.2 shows a part of the search space to explore to find the ODT. Table 5.1 and Figure 5.3 show cache implementations based on a hash table and a trie, respectively. In the hash table data structure, the path is passed through a hash function, but to avoid collisions, every path needs to be stored with its associated information. In the trie, nodes are created for every prefix of a path.

The size of these data structures can differ, as illustrated by the example. Consider the path abc (in red) during the search. This path is reached in the search by first considering two other paths: $\{a, ab\}$. To save these three paths $\{a, ab, abc\}$ in the cache, in the case of a hash table, an entry is created for each path, and all tests in each path are saved. This is represented by red entries in Table 5.1: 6 tests must be saved to store the paths leading to abc. In the case of trie, the tests in common for parent paths are shared in such a way to avoid saving duplicate tests. To store abc and its parent paths, only 3 tests are necessary for a trie data structure, as the trie uses a compressed way to store paths.

Theoretically, the number of tests that need to be stored can be calculated for both data structures when no pruning strategy is used, and the ordered path representation is used as the key for the entries. In the case of a trie, this number is equal to the number of nodes. For each depth d, this number is expressed as $2^d \times C_d^{|\mathcal{F}|}$. The term $C_d^{|\mathcal{F}|}$ is the number of d-combinations in the set $\mathcal{F}$ and represents the number of different possible ordered paths of length d relying on the features in $\mathcal{F}$. The term 2^d represents the two tests (denoting the splits in a binary tree) of each feature in the path, given that it has a length d. On the other hand, each path stored for a certain depth d in a hash table, has the same length as d. Then the number of tests stored for each depth is $d \times 2^d \times C_d^{|\mathcal{F}|}$. Thus, the multiplication factor using a hash table is d. However, note that the expression $2^d \times C_d^{|\mathcal{F}|}$ is generally quite large (given d). Therefore, the impact of the multiplication factor d using a hash table is significant.

In practice, the memory consumption in Figure 5.1 of the trie data structure (in blue) is lower than that of the hash table (in green), and this motivates our choice for a trie in our experiments. However, note that in terms of the number of paths stored in the cache, the trie can be larger. For instance, this is the case for the paths leading to ec in Figure 5.2. While the hash table stores them using two cache entries, the trie needs three nodes for this operation. This explains why the number of cache entries in the trie is higher than in the hash table in Figure 5.1. However, as the hash table stores a complete path per entry, the memory consumption is higher. We thus choose the representation with the lower memory consumption.

5.4 Learning ODTs with limited memory resources

In this section, we present our proposal to limit the memory consumption of DP-based learning algorithms.

Our core objective is to make sure the size of the cache is limited, while making sure that the performance of the search is not affected too much. Moreover, we wish to do so in a manner that can be integrated in DP-based algorithms with minimal modifications.

The core idea of our approach is simple: instead of keeping all the elements in the cache, when necessary we will remove elements from the cache; when the search algorithm encounters these elements later, it will recalculate the results.

Hence, limiting the memory size will certainly impact the run time of the search algorithm, as its speed depends mainly on the reusability of the cache. The fewer entries in the cache, the less likely it is to find an existing solution. However, a good strategy specifying the order in which entries are deleted can limit the impact on the overall run time. In the next subsection, we will present our proposal for a bounded cache based on one deletion strategy. Subsequently, we will introduce additional deletion strategies and how to integrate them in the maintenance of the cache.

5.4.1 Implementation of a bounded cache

As the modifications of the DP-based systems we propose only concern the maintenance of the cache, we focus the description of our contribution to the maintenance of the cache. Pseudo-code for this can be found in Algorithm 8, for the trie data structure. Later, we explain how the pseudocode can be adapted to the simpler case of hash tables. Compared to the cache used by other systems, our new cache system requires two parameters. The first, `maxcachesize`, specifies the maximum number of entries that the cache can store. The second parameter; `wipeFactor` defines the percentage of the cache that will be cleared when the cache is full.

Any DP-based algorithm requires functionality that for a given key returns the data associated to the key, and if no such information is available, will add information to the cache; this is performed in Algorithm 8 by the function `insertOrGetEntry`. This function traverses the tests in a path, and either finds back the path in the trie, or adds the necessary nodes. Compared to the original trie-based system DL8.5 (Algorithm 5), there are two differences in the insertion method. First, there is a condition (line 14) that checks whether there is enough place in the cache to insert the path being created, otherwise the `wipe` function is called; this `wipe` function is responsible for removing elements from the cache. Second, each path added to the trie is also added to a deletion queue (line 17). This queue is maintained to keep track of the nodes in the cache, which is used when wiping the cache.

Since a node in a trie has a parent that links to it, when a path is deleted from the cache, it is necessary to delete the edge from its parent node. For this reason, the entry representing its parent node should be stored in order to perform the deletion of the edge.

A critical part of our contribution is the `wipe` function. At its core, this function sorts the nodes in the cache, and subsequently deletes the desired number of nodes from the cache according to this order. Of critical importance is here how the nodes are sorted for removal. To determine this order, our wipe function first calls a number of functions to compute necessary information for the entries in the cache. This information is subsequently used when sorting the nodes. The remaining

Algorithm 8: Class `Bounded_Cache(maxcachesize, wipeFactor)`

```
struct PathEntry{ useful: bool; curFeat: int; nDscPaths: int; solution:
  BestTree; childEntries: HashSet<int, PathEntry>}
struct BestTree {lb : float; feat : int; error : float }
// list of pairs <pathEntry, parentPathEntry>
deletionQueue ← pair<PathEntry, PathEntry> [maxcachesize]
rootEntry, cachesize ← newEntry(), 0
Method newEntry()
    return PathEntry(false, 0, −1, (0, NO_FEAT, +∞), {})
Method insertOrGetEntry(p : array of int) // p: path
    pEntry ← rootEntry
    for t ∈ p do // foreach test in path
        if t ∈ pEntry.childEntries then
            pEntry ← pEntry.childEntries.get(t)
        else // remainingTests() returns the number of tests in
            // the path which are not in the cache
            if cachesize + remainingTests() > maxcachesize then wipe()
            childEntry ← newEntry()
            pEntry.childEntries.push({t, childEntry})
            deletionQueue.push({childEntry, pEntry})
            cachesize, pEntry ← cachesize + 1, childEntry
    if pEntry.solution.error = +∞ then
        pEntry.solution.error ← leaf_error(p)
    return pEntry
Method wipe()
    // mark nodes that are required by the current state of the search
    setCurUsefulTag(rootEntry, {})
    countDscpaths(rootEntry)
    setOptiUsefulTag(rootEntry, {})
    sortDeletionQueue()// desc order with useful nodes at beginning
    nDel, counter ← (int)(maxcachesize ∗ wipeFactor), 0
    for path, parPath ∈ reverse(deletionQueue) do
        if counter = nDel or path.useful is true then break
        parPath.childEntries.remove(path)
        delete(path)
        counter, cachesize ← counter + 1, cachesize − 1
    unSetUsefulTag() // Remove tags
Method countDscpaths(pE : PathEntry)
    pE.nDscPaths ← 0
    for t ∈ pE.childEntries do
        pE.nDscPaths ← pE.nDscPaths + 1 +
          countDscpaths(pE.childEntries.get(t))
    return pE.nDscPaths
Method getChildPathEntries(p : array of int, feat : int)
    $t_1, t_2$ ← tests(feat)
    return {getEntry(sort(p∪$t_1$)), p∪$t_1$}, {getEntry(sort(p∪$t_2$)), p∪$t_2$}
```

```
Method setCurUsefulTag(pEntry : PathEntry, p : array of int)
    for pE, p ∈ getChildPathEntries(p, pEntry.curFeat) do
        pE.useful ← $true$; setCurUsefulTag(pE, p)

Method setOptiUsefulTag(pEntry : PathEntry, p : array of int)
    pEntry.useful ← $true$
    for pE, p ∈ getChildPathEntries(p, pEntry.solution.feat) do
        setOptiUsefulTag(pE, p)
```

steps of the `wipe` function are straightforward. The deletion queue is traversed from the end to the beginning, and each path entry is deleted (line 29) until the number of entries to be deleted is reached, or all possible paths have been deleted (line 30). Note that the number of entries to delete is calculated according to the percentage provided by the `wipeFactor` parameter (line 28). Before deleting a path, the edge that connects it to its parent path is deleted (line 31) to inform the parent path that its child path no longer exists.

The information that is collected for elements in the cache is the following.

(1) Information concerning which paths the search is currently considering. Please remember that DP-based approaches perform a recursive search over paths, by adding tests to paths. In the process of calculating the result for the path p, they assume that path p is present in the cache. The `setCurUsefulTag` method is used to mark the paths p that are currently under evaluation. These paths will be put first in the order, and will never be deleted. Please note that the number of such nodes is very small. The `setCurUsefulTag` method uses a `curFeat` field in the entries in the cache. This field is initialized by a small modification of the recursive search function. For DL8.5 this is illustrated in Algorithm 9, where in green the code is indicated that is added in the recursive search function of DL8.5; parts of the code of DL8.5 that are not modified are skipped as they are provided in Algorithm 7.

(2) Statistics concerning nodes in the cache. One such statistic which can be used to order elements, illustrated in our pseudocode, is the number of descendants a node has in the trie. In the trie case, when we delete a node from a trie, we also need to delete its children from the trie to preserve the consistency of the structure. This implies that each path must be deleted before its parent path. As a parent has more descendants than its children, by ordering nodes on the number of descendants, we can ensure children are deleted before their parents. Moreover, an intuition is that paths with numerous descendants are more ex-

pensive to evaluate than those with fewer descendant paths, and hence should be removed less quickly. To count the number of descendant paths per path, a simple post-order traversal is performed through the trie using the `countDscpaths` method. The result is stored in the variable `nDscPaths` (line 1).

(3) Information concerning which paths are part of the current optimal solution; we will return to this issue later on.

These three information (put in green on line 1) are new compared to information stored in the cache entry of DL8.5 (Algorithm 5).

5.4.2 Different wipe strategies

In the pseudo-code above, the statistic we used to order nodes was *the number of descendant path entries in the cache*. We also consider the following alternatives.

Number of reuses of solutions The intuition behind the number of reuses of elements is that a path that has been reused many times is more likely to be needed again. This strategy removes the less frequently reused solutions before the more frequently used ones. To calculate the number of times a path solution has been reused, a variable initialized to 0 must be added to the path entry structure. Whenever an existing path is returned from the method `insertOrGetEntry`, the variable must be incremented for the path. Note that this deletion criterion does not satisfy the requirement of deleting the deepest nodes first in a trie. To enforce this behavior, there are two possibilities. The first is to set the method `sortDeletionQueue` so that the sort is performed according to two parameters. First, the length of the path and then the number of reuses. Another possibility is to increment the number of reuses for each ancestor of a path as well. In this work, we use the second option because it is based solely on the number of reuses, rather than using an additional criterion. Note that for this strategy, the statistic is not computed in the `wipe` method.

All not required paths This strategy, as the name implies, deletes all paths except from the *useful* ones, as defined by criteria (1) and (3) above. No variable needs to be added to the path entry structure, and the deletion queue is not required. When the cache is full, after setting *useful* nodes, a post-order traversal is sufficient to delete all nodes without the *useful* tag.

5.4.3 Comparison with a priority queue

In our approach to bound the size of the caching data structures when learning ODTs, we propose the use of an additional array whose size is fixed to the maximum cache size allowed. This array is used to compute the order of deleting cache entries whenever the cache is full and we need to free up space for new entries. When the order is calculated, we delete a number of stored entries to make room for new entries. In contrast to this technique, in most cache size limitation approaches, a priority queue is used instead of an array with the goal of always deleting in $\mathcal{O}(1)$ the worst entry. In this priority queue situation, the worst entry is replaced by the new entry when there is no more space in the cache. However, for the criteria we use to order entries, this technique is not practical. Specifically, the value of the statistics we rely on to decide the order in which to delete entries changes as the search evolves. In this situation, using a priority queue will result in a permanent re-structuring operation to correct violated queue properties. Note that each re-structuring operation is performed in $\mathcal{O}(\log n)$ when the priority queue is implemented as a binary heap. This creates a considerable overhead, especially since some statistics (such as the number of reuses) lead to the modification of several priority values at once (given that all itemsets on a path need to be updated).

In addition to the above problem, the insertion and extraction complexities of a priority queue are each $\mathcal{O}(\log n)$ in the worst case, whereas these operations are performed in $\mathcal{O}(1)$ using an array. The real complexity to consider in our array case is the complexity of the sort that needs to compute the deletion order. This operation is performed in $\mathcal{O}(n \log n)$. Note also that this operation is performed only once when the cache is full.

Nevertheless, it is still possible to force the use of a priority queue, so that only the worst entry is replaced whenever an insertion occurs and the cache is full. However, the statistics to rely on must be static. An example of a statistic that might work with the priority queue is the depth of entries. It can also be coupled with a timestamp; whether such strategies are useful is future work.

5.4.4 Returning an ODT that relies on deleted elements

Until now, our recursive equations focused on returning the error of the most accurate tree. However, in practice we also wish to return the tree that obtains this error. As explained in Chapter 4, in DL8 an approach was proposed that allows to do so with minimal additional memory use:

Algorithm 9: Bounded-DL8.5(maxdepth, maxcachesize, wipeFactor)

```
cache ← Bounded_Cache(maxcachesize, wipeFactor)
DL8.5−Recurse({}, $+\infty$)
while cache.exploreTree() is incomplete do reconstituteWipedNodes({})

return cache.exploreTree()
// p is the path whose solution must be found
Procedure DL8.5−Recurse(p : array of int, ub : int)
    pEntry ← cache.insertOrGetEntry(sort(p))
    ... // if entry exists and has been solved, return its solution
    for each feature F in a well-chosen order and splittable in tests f and $\overline{f}$ do
        pEntry.curFeat ← F
        ... // compute error of best tree rooted by F
    pEntry.curFeat ← −1
    ... // return error and root of the tree with the lowest error
Procedure reconstituteWipedNodes(p : array of int, ub : int)
    pSol ← cache.insertOrGetEntry(sort(p)).solution
    if pSol.feat = NO_FEAT then return void
    ($pE_1, p_1$), ($pE_2, p_2$) ← cache.getChildPathEntries(p, pSol.feat)
    $found_{pE_1}$ ← $pE_1$ ≠ NULL and $pE_1$.solution.feat ≠ NO_FEAT
    $found_{pE_2}$ ← $pE_2$ ≠ NULL and $pE_2$.solution.feat ≠ NO_FEAT
    if $found_{pE_1}$ is false or $found_{pE_2}$ is false then
        if $found_{pE_1}$ is false and $found_{pE_2}$ is true then
            $pE_1$.solution.lb ← pSol.error − $pE_2$.solution.error
            DL8.5−Recurse($p_1$, $pE_1$.solution.lb + 1)
            reconstituteWipedNodes($p_2$)
        else if $found_{pE_2}$ is false and $found_{pE_1}$ is true then
            $pE_2$.solution.lb ← pSol.error − $pE_2$.solution.error
            DL8.5−Recurse($p_2$, $pE_2$.solution.lb + 1)
            reconstituteWipedNodes($p_1$)
        else
            $pE_1$.solution.lb ← 0
            DL8.5−Recurse($p_1$, pSol.error + 1)
            $pE_2$.solution.lb ← pSol.error − $pE_1$.solution.error
            DL8.5−Recurse($p_2$, $pE_2$.solution.error + 1)
    else
        reconstituteWipedNodes($p_1$)
        reconstituteWipedNodes($p_2$)
```

for every path p, only the feature F is stored in the cache that should be used to split optimally for p. The observation is that the optimal split for $p \cup \{f\}$ and $p \cup \{\overline{f}\}$ can also be found in the cache. Unfortunately, if we wipe part of the cache, this strategy can no longer be used: if the optimal tree relies on a path that is no longer in the cache, we can no longer recover this tree from the cache.

One solution could be to associate a complete optimal tree to every path in the cache, but this would blow up the memory required for the cache, which we would like to avoid. Hence, a critical contribution of this work is an alternative solution that works well in practice: it avoids the use of large amounts of memory, while being fast at the same time. This solution consists of two components.

First, at the moment that we wipe the cache, using the function `setOptiUsefulTag`, we determine which paths in the cache are part of the currently optimal solution; we do not remove these paths from the cache. In the best case, this optimal solution does not change any more and hence we can recover (most of) the solution from the cache.

Unfortunately it cannot be excluded that when the search continues, we find that a tree with a better quality exists that relies on paths that have been removed from the cache. In this case, we propose to *recalculate* the optimal tree for these paths. This algorithm is executed at the end of the original search to avoid making calculations for paths that later in the search may no longer be considered part of the final solution. This algorithm is described by the procedure `reconstituteWipedNodes` in the Algorithm 9. For each existing path, an attempt is made to obtain its two children paths from the cache (lines 16-18). If they exist (line 19), the DFS traversal continues (lines 34-35). Otherwise, the search is restarted. However, an important difference with the original search is that from the known errors (including the error of the optimal tree), much better bounds can be deduced than in the original search. In the case where a right child path exists but a search must be rerun for the left child path (lines 20-23), a simple subtraction between the errors of the parent path and the right child path provides the exact error of the left child path to be found. This error is used as a lower bound and a small value ϵ is added to it to define the upper bound. In the context of the misclassification rate, $\epsilon = 1$ is used. The same process is performed in the case of an existing left child path and a non-existing right child path (lines 24-27). When both child paths are nonexisting (lines 29-32), a search is performed for a first child path and the bounds for the second are derived from its solution. For the first path, the unknown lower bound is set to 0 while the upper bound is at most the error of the parent path added to ϵ. This procedure to reconstruct the wiped paths of the final ODT is very efficient in practice.

5.4.5 Adapting to hash tables

The implementation of the bounded trie-based cache can easily be adapted to a hash table and is even simpler in this case. In order to sort and delete paths in a specific order, the deletion queue no longer needs to store pointers towards parents. Moreover, non-useful cache entries can be deleted in any order without hierarchy constraints, leading to a greater freedom in the choice of deletion criteria. For example, for the *reuses number* strategy that we propose, in the case of a hash table, we can rely only on this number to define the deletion order of paths without having to increment the *reuses number* of ancestor paths.

5.5 Results

In this section we answer five main questions:

Q1 Which one of the wipe strategies proposed has the lowest impact on the run time?

Q2 What is the impact on the memory usage when using a bounded cache?

Q3 What is the impact of a bounded cache on the run time?

Q4 How fast is the algorithm to recover removed nodes from the ODT?

Q5 How does the use of a bounded cache compare to a MIP approach?

The implementation of the learning algorithm without cache size restriction that we use in our experiments is `DL8.5` [ANS20; ANS21b]. This means that the cache system used is a trie. However, we add some improvements to this `DL8.5` implementation to reduce the baseline memory consumption. These are the improvements proposed by MurTree and explained above in the chapter. We call the final algorithm `DL8.5` in our experiments because the main search features originate from `DL8.5`. `Bounded-DL8.5` is the `DL8.5` version using our bounded cache. Experiments were run on a Linux Rocky 8.4 server with an Intel Xeon Platinum 8160 CPU @ 2.10Ghz and 320GB of memory.

In the first experiment, we use 20 binary datasets from CP4IM[1]. We run `DL8.5` on these datasets with a time limit of 5 minutes. To avoid comparing too easy instances, we learned ODTs of depth 5. Then we run `Bounded-DL8.5` with the same parameters until we find the ODT, or we reach the same point in the search as `DL8.5` when the timeout was raised. We show in Figure 5.4 two ratios of `Bounded-DL8.5` over `DL8.5`: the run time and the total number of entries (including those created

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

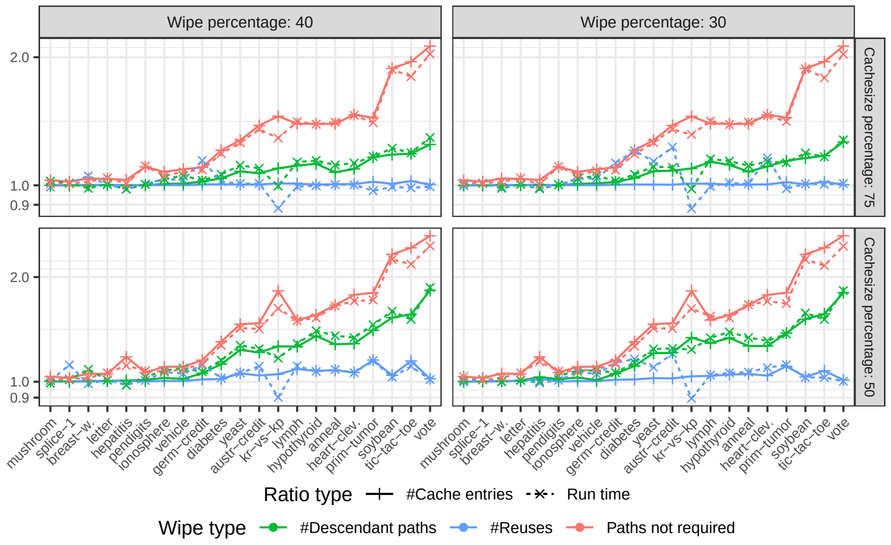


Figure 5.4: Time factor of `Bounded-DL8.5` per wipe strategy

after they have been removed). Each wipe strategy is considered for `Bounded-DL8.5`: the number of descendants (#DescendantsPaths), the number of reuses (#Reuses) and the paths not required. For the wipe parameters, we consider 75% and 50% of the memory as value for the `maxcachesize` parameter and a wipe factor of 40% and of 30%. Note that the results shown are representative for the results for other choices of the parameters. It is interesting to notice that the time spent by the algorithm is proportional to the number of cache entries created. It can also be seen that the number of cache entries and the run time increase as the bound of the cache is restricted. On the other hand, it is difficult to observe a concrete trend in these values when a change is applied to the percentage of memory to free at each cache wipe. Note, however, that it becomes easy to answer **Q1** thanks to Figure 5.4. As expected, the strategy of removing all non useful entries from the cache is the one increasing the most the run time of `Bounded-DL8.5`. Instead, the intuition of keeping as long as possible the entries often reused shows the best reduction of time impact. It performs better than removing the paths based on the number of descendants.

After this experiment, we select the best wipe strategy (*number of reuses*) to evaluate how `Bounded-DL8.5` can impact the memory usage on situations in which `DL8.5` requires a lot of memory to find the ODTs. To highlight these cases while ensuring reasonable run times, we experimentally select five specific datasets and depths. In the same way,

Table 5.2: Comparison of unbounded and bounded caches

Dataset	nFeats	nInsts	Depth	DL8.5			Bounded−DL8.5				BinOCT	
				|Cache|	Time(s)	Mem (GB)	Time(s)	Time factor	Mem (GB)	rTime (ms)	Time	Mem (GB)
Anneal	93	812	8	648M	21907	78.5	65453	2.99	**9**	2.63	TO	9.8
Diabetes	112	768	7	238M	18890	28	31534	1.67	**6.3**	0.21	TO	10.5
German-credit	112	1000	6	92M	15198	11	21898	1.44	**5.1**	0.10	TO	10.1
Kr-vs-Kp	73	3196	8	136M	3865	16.1	6365	1.65	**5.3**	4.68	TO	35.8
Yeast	89	1484	7	419M	34977	50.5	62483	1.79	**7.8**	4.70	TO	17.5

we limited the cache size of `Bounded-DL8.5` to 30 million (30M) entries at most and set the percentage of entries to wipe to 40%. The impact of these values is already discussed above. The results are reported in Table 5.2. The memory usages are obtained by using the program *top* available on Unix operating systems. Notice that the memory needed to solve the problems using `Bounded-DL8.5` never reaches 10GB while up to 78GB is required using `DL8.5`. Regarding the dataset *anneal*, more than 600 million of entries are created with `DL8.5` to find an ODT of depth 8. The advantage of using `Bounded-DL8.5` is that this number can be reduced to a desired quantity, here 30 million. As an answer to **Q2**, this reduces significantly the memory needed to find an ODT. Note, however, that this reduction depends on the limit on the number of cache entries. Moreover, it also has an impact on the run time. The time factor is recorded in Table 5.2 and is defined as the run time of `DL8.5` over `Bounded-DL8.5`. To answer **Q3**, notice that our strategy managed to achieve a time factor less than 1.5 on an instance that lasts over 3 hours with `DL8.5`. In the worst case, it almost reaches a time factor of 3 on an instance that originally required 78GB, which it reduces to 9GB. Regarding our algorithm ability to recover the nodes of the ODT that have been removed during the search, the time used by our algorithm is reported in milliseconds in the column *rTime*. Notice that for all instances, our algorithm uses less than 5 milliseconds to recover the removed nodes from the ODT. This answers the question **Q4**.

To answer **Q5**, we finally compare `Bounded-DL8.5` to a MIP approach. For this, we use `BinOCT`[2] model. The model is solved using CPLEX[3] 22.1.0. As MIP approaches are time consuming, we set a time limit to 70000 seconds, which is greater than the maximum time used by `Bounded-DL8.5` to solve an instance. Notice in Table 5.2 that `BinOCT` does not manage to solve any instance in the allocated time. This is represented by TO (timeout). Moreover, notice that the memory used

[2]https://github.com/SiccoVerwer/binoct
[3]https://www.ibm.com/analytics/cplex-optimizer

by `BinOCT` is greater than `Bounded-DL8.5` for all instances.

5.6 Conclusions

In this chapter, we address the problem of the huge memory consumption required by caching based algorithms for learning ODTs. We propose a technique that can be added to existing caches to wipe a desired number of entries from the cache. This leads to a significant reduction in memory consumption, with only a small impact on run time. We propose strategies to reduce the impact of the wipe on the overall run time. Finally, we show that our approach finds ODTs more quickly than MIP approaches, while also consuming less memory.

Part III

Applications of ODTs

Using ODTs to assess optimal forest criteria 6

This chapter is based on the paper G. Aglin, S. Nijssen, and P. Schaus. "Assessing Optimal Forests of Decision Trees". In: 2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI). IEEE Computer Society. 2021, pp. 32–39.

6.1 Introduction

Previous work in the literature showed that on many datasets optimal depth-limited decision trees perform well [BD17]. However, it is well-known in the machine learning literature that compared to other machine learning models, the predictive performance of decision trees is limited. One popular way to build stronger classifiers based on DTs, is the use of *ensemble learning*. The principle of *ensemble learning* is to combine many "weak" classifiers to build a stronger one. Ensemble learning techniques have proven for many years to be able to improve the generalization power of "weak" learners. Even though the ensemble learning concept may apply to any base classifier supporting sample weights, the most well-known use of ensemble learning is in combination with decision trees. In the remainder of this chapter, we will only consider ensemble learning models based on decision trees and call the resulting models *forests* or *decision forests*.

One of the most popular techniques for ensemble learning is boosting. Boosting algorithms learn successively a set of classifiers, each having a weight; the weighted prediction of the trees is the prediction of the overall model. One of the most well-known boosting algorithms is the Adaboost algorithm [FS97]. In Adaboost, heuristics are iteratively used to calculate weights for training instances; for each set of weights, a heuristic decision tree learning algorithm is used to learn a decision tree that takes these weights into account. In practice, this combination of heuristics works well.

The heuristic nature of Adaboost makes it very different from many other popular approaches in machine learning, such as deep learning, in which a global optimization criterion (or loss function) is defined, and subsequently an optimization algorithm is used to find a model that optimizes this criterion. For this reason, it is difficult to explain what the performance of Adaboost is due to.

To bridge this gap, and to get a better understanding in boosting in general, researchers have studied the definition of global optimization criteria for boosting. Accuracy by itself is not a good criterion: it could lead to very complex models that overfit training data. Hence, these studies focused on optimization criteria that could also reduce the risk of overfitting. This led to the development of new boosting techniques relying on mathematical models, of which LPBoost [DBS02] and MD-Boost [SL10] models are the prime examples that we will consider in this thesis. However, even though both techniques rely on a definition of a global optimization criterion, to optimize this criterion the authors of LPBoost and MDBoost still propose to use heuristic techniques; these techniques do not possess optimality guarantees, as the authors considered solving these criteria optimally infeasible. As a result, until now there is no approach to find provably optimal forests. Therefore, there is no concrete way to assess if the optimization criteria proposed by LP-Boost and MDBoost could explain the performance of boosting forests.

In this chapter, we will first show that by combining the algorithm used to solve LPBoost and MDBoost, and recent algorithms for finding optimal decision trees (ODTs), we can solve the boosting problems proposed in LPBoost and MDBoost till optimality. Hence, ODT techniques make it possible to evaluate the usefulness of the global optimization criteria without interference from the heuristics used in the underlying algorithms till now.

This development makes it possible to answer the following experimental questions:

1. What is the optimization gap between LPBoost and MDBoost using an optimal decision tree algorithm and a traditional heuristic algorithm?

2. Does it make a difference in practice on generalization error to find an optimal ensemble instead of a heuristic one?

3. is the generalization capacity of optimal LPboost and MDBoost forests similar to heuristic boosting forests?

The answers to these questions are not obvious. Heuristic decision tree learning algorithms are often used in boosting algorithms, and they are known to lead to a good performance; it is not obvious what ODT algorithms could add. Moreover, Reyzin and Schapire stated in [RS06] that it is important that the classification models used in boosting are not strong; otherwise the ability to generalize towards new data is compromised. It is not clear that ODTs or Optimal Decision Forests meet this requirement. Our experiments will help to answer these questions.

This chapter is organized as follows. Section 6.2 presents the state-of-the-art of boosting techniques as well as different intuitions about why Adaboost performs well. Section 6.3 presents the technical background related to learning optimal ensembles while Section 6.4 introduces our methodology to boost ODTs. Finally, section 6.5 presents a comparison of optimal forests, heuristic approaches and Adaboost.

6.2 Related work

In this section, we discuss previous work related to boosting. Adaboost [FS97] is one of the most popular boosting algorithms. It produces a weighted ensemble of several "weak" DTs to build a strong model. The different DTs are generated successively, each new tree focusing on the errors of previous trees in the ensemble. For this, the weight of each instance is re-evaluated after adding a new tree, in such a way that the next tree focuses on misclassified instances. The resulting approach is known to work very well in practice. Forests found by Adaboost show great performance on training set as well as on test set data. The resulting model is insensitive to overfitting on many datasets [Bre+98].

Even though Adaboost works well in practice, it is not clear why it is the case. Many studies have been done on Adaboost to find out the reason for its success. Breiman argued that Adaboost can be viewed as minimizing a loss function: *exponential loss* [Bre97b]. Moreover, Mason et al. [Mas+99] suggested that Adaboost performs a sort of gradient descent. This perspective on Adaboost has been shared by other researchers [FHT+00; ORM98; SS99; ROM01; FD98] . However, experiments conducted by Schapire and Freund [SF13] rejected the hypothesis that the success of Adaboost is only due to exponential loss minimization. Indeed, they implemented a gradient descent algorithm which minimizes the exponential loss, but that generalizes poorly compared to Adaboost.

Schapire et al. associated the success of Adaboost with a concept called *margin* [Sch+98]. This is a confidence score of an instance given an ensemble of classifiers. While a positive margin indicates a correct classification contrary to a negative one, a higher margin indicates a higher confidence in the prediction than a lower one. To support the margin argument, they showed the cumulative distribution plot of training instance margins obtained by Adaboost given different numbers of iterations. This experiment showed that as the number of iterations increases, Adaboost tends to maximize the *margin* of training instances: this explains why Adaboost continues adding new trees even when it reaches a perfect accuracy on a training set. Breiman [Bre97b] and Rätsch et al. [RW05] proposed respectively the Arc-gv and Adaboost* algorithms in order to maximize the minimum margin of training instances in a more fundamental manner than Adaboost. After Breiman [Bre97a] proved that the minimum margin maximization can be formulated as a linear program, Grove and Schuurmans [GS98] proposed two algorithms to maximize the minimum margin: LP-Adaboost and DualLPboost. LP-Adaboost is an Adaboost adaptation in which the tree weighting part is replaced by a linear program maximizing the minimum margin. DualLPboost takes this a step further, and is the first algorithm relying on a global optimization criterion based on margins; for the optimization of this criterion, this approach was the first to propose the use of *column generation* optimization technique in combination with a heuristic DT learning algorithm.

Despite an elegant mathematical formulation with a clear optimization criterion, these algorithms maximize the minimum margin slightly worse than Adaboost in practice. Reyzin and Schapire [RS06] looked at the Arc-gv algorithm to understand why it does not perform as well as Adaboost. They concluded first that the minimum margin maximization leads to the construction of too complex classifiers in the ensemble. This leads to the overfitting the training data. On the other hand, they found that the minimum margin maximization sacrifices the margin of the other instances. In their analysis of DualLPboost, Grove and Schuurmans [GS98] reached the conclusion that the hard maximisation of the minimum margin in DualLPboost could only work if data are linearly separable.

These observations led to the development of two boosting techniques based on column generation that are the basis of this chapter [DBS02; SL10]. Demiriz et al. [DBS02] proposed LPBoost, in which the minimum margin is maximized while also allowing for mistakes by means of a slack variable for each instance. The minimum margin maximization objective is then regularized by the minimization of the sum of misclassifications. When running their algorithm for a limited number of iterations, it was found to outperform DualLPboost, but it did not outperform Adaboost.

To reduce even more the impact of the minimum margin maximization on the other margins, Shen and Li [SL10] proposed MDBoost, a new column generation approach to maximize the margin distribution. Concretely, this approach maximizes the average of margins and minimizes their variance at the same time. The model includes a regularization parameter to decide whether the algorithm focuses more on the maximization of the margin distribution or on the minimization of their variance.

So far, within LPBoost and MDBoost heuristic DT algorithms were used; the resulting ensemble models are thus not guaranteed to be optimal given the optimization criterion proposed by the authors. Our work addresses this weakness.

IPboost [PP20] is a branch-and-price approach to learn boosted trees with respect to a misclassification rate optimization criterion. Also within this technique only heuristic DT learning methods are used. Another approach in the literature is [Miš20]. This work proposes a MIP model which, for given a predefined ensemble structure, fills in the features used in the ensemble, taking the misclassification rate as the optimization criterion. Contrary to our work, this approach limits the size of the ensemble and the structures of the trees in the ensemble.

6.3 Technical background

In this section, we present the technical background on which our work relies.

6.3.1 Boosting through margin optimization problem

As mentioned previously, Adaboost performs well in practice and many studies have been performed to find the reason for its success. Among the different research axes explored, the one on which we build in this chapter is the concept of *margin* maximization. Let $\{(\boldsymbol{x}_i, y_i)\}, i = 1, \ldots, M$ be the training data, where $\boldsymbol{x}_i$ is the feature vector of an instance, associated to a class $y_i \in \{-1, +1\}$. Let $h(\boldsymbol{x}) \in \mathcal{H}$ be a function representing a base classifier that associates to each instance $\boldsymbol{x}$ a class $y \in \{-1, +1\}$; in our case $\mathcal{H}$ represents the space of all possible decision trees for a given dataset, possibly under constraints, such as a constraint on the depth of the trees. The boosting problem consists in finding a weighted set $\mathcal{H}' = \{h_1(\boldsymbol{x}), \ldots, h_T(\boldsymbol{x})\} \subseteq \mathcal{H}$ which obtains good classification accuracy on *future* data. However, given that we do not have these instances at training time, we need to find a criterion that can be applied on training data to avoid overfitting.

There are some common ideas in how heuristic-based boosting approaches tackle this problem. Based on the prediction and the weight of each classifier in an ensemble, a prediction score is computed for each training instance. For a tree t predicting a class $h_t(\boldsymbol{x}) \in \{-1, +1\}$ and associated to a weight w_t, the score of a training instance $\boldsymbol{x}_i$ is equal to $\varrho_{ti} = y_i h_t(\boldsymbol{x}_i) w_t$, where y_i is the true label according to the training data. ϱ_{ti} takes values in $\{-w_t, w_t\}$. The sign of the score denotes whether the tree correctly classifies the instance or not while the value w_t denotes the confidence of the decision. For a set of T trees, let $H \in \mathbb{R}^{M \times T}$ be a matrix where the entry (i, t) of H is $H_{it} = h_t(\boldsymbol{x}_i)$. H_{it} is the output of the classifier t given the training instance $\boldsymbol{x}_i$. $H_{i:}$ is the vector of predictions for the instance $\boldsymbol{x}_i$ and $H_{:t}^{\top}$ is the vector of predictions for all of the training data given a tree t. $\boldsymbol{w} = (w_1, \ldots, w_T)$ is the vector of weights of different trees. The prediction score for an instance $\boldsymbol{x}_i$ given an ensemble of trees is computed by summing its score for each tree, $\varrho_{Ti} = \sum_{t=1}^{T} y_i h_t(\boldsymbol{x}_i) w_t$. This score is the unnormalized *margin* of an instance given an ensemble of classifiers. Its vector expression is $\varrho_{Ti} = y_i H_{i:} \boldsymbol{w}$. The normalized *margin* ensuring that the weights of all classifiers sum to 1 is $\rho_{Ti} = y_i H_{i:} \boldsymbol{w} / \|\boldsymbol{w}\|_1$. The *margin* score of an ensemble is a kind of balance between the incorrect decisions and the correct ones. Each decision having a weight, the final decision will be correct if trees predicting correctly have a higher weight than the others and vice versa. Concretely, an instance is predicted as 1 when its margin is positive and -1 when it is negative. As for a single tree, the final score expresses how confident the model is about a prediction. A model with higher margins on data is more confident than one with lower margins. Schapire et al. [Sch+98] stated that Adaboost performs a kind of margin maximization, arguing that it continues to increase the test score by adding new trees even when the training score reaches 100% of accuracy.

Many studies [GZ13; Sch+98] concluded that the success of Adaboost is due to an optimization of the margin distribution on the training data without concretely stating what a good margin distribution is. In this chapter, we present LPBoost and MDBoost, two boosting approaches optimizing the margins in training data but using different criteria.

The key idea underlying these boosting approaches, is to formalize the boosting problem as a mathematical programming problem in which we associate a Boolean variable to each possible decision tree that is allowed to be put in the ensemble; the optimization problem is to assign truth values to these Boolean variables. Due to the high number of possible tree given a depth, the model solving will be very time-consuming. An interesting solving technique to use in this context is the column generation [Du +99; DDS06].

6.3.2 Column generation

Column generation [Wol20] is a well-known approach to efficiently solve large-scale optimization problems; i.e., problems with numerous variables. Despite the large scale of these problems, a few variables are generally involved in their optimal solution; most variables are ignored in the optimal solution. Therefore, if we could instantiate a large-scale optimization problem only for variables that are part of the optimal solution and would solve the resulting optimization problem, the solution would still be optimal. It could not be possible to find a variable to add to the current solution in such a way as to obtain a better solution. Column generation is suitable for solving these problems. Instead of solving the main problem, which is difficult, the column generation process starts solving the problem by considering a small set of variables. It will lazily and iteratively add the most promising variable capable of improving the objective value until no variable can be found whose addition to the problem still leads to a better solution. The new variable iteratively added is also called *column*. To achieve the goal of column generation, the initial problem, also called the master problem, is split into 2 problems: *the primal* (or restricted master problem) and *the pricing*.

The primal problem is a restriction of the main problem with all variables to only some selected variables. *The primal* problem allows solving at each iteration of the column generation process, the master problem using selected variables. On the other hand, *the pricing* problem allows generating the new variable capable of improving the objective value of the current solution.

Concretely, to solve a linear programming problem, an iterative process is performed in which some variables are assigned (*basis variables*) while others are not (*non-basis variables*). From one iteration to the next, a non-basis variable moves to the basis while a basis variable leaves the basis. Through this process, a new admissible solution is found that improves the previous one. The process is repeated until the optimal solution is reached. To find a new variable to introduce in the basis, a coefficient is computed for each non-basis variable to assess how much the current objective value will change when its value is increased. This coefficient is the *reduced cost*. It should be negative to be a valid candidate. Refer to [IC94; Wol20] for further details about the reduced cost.

Solving the pricing problem in the column generation process allows the generation of a variable that improves the objective value of the master problem because it provides the variable with the lowest reduced cost. Note that finding a variable with a negative reduced cost is equivalent to finding a variable that violates a constraint of a companion problem of the primal: *the dual problem*.

Indeed, the dual problem is a companion problem derived from the primal problem. For each variable in the primal problem, there is a corresponding constraint in the dual problem. In the same way, there is a variable in the dual for each constraint in the primal. Further details about duality can also be found in [IC94; Wol20].

When no variable violates a constraint of the dual, then there is no variable with a negative reduced cost that can enter the basis to improve the objective value of the master problem. At this point, the optimal solution of the master problem is found. The pseudocode of the column generation process can be summarized in Algorithm 10.

Algorithm 10: Column generation

```
1 variables ← initialize a set of variables
2 solution ← {}
3 do
4     solution, dual_variables ← solve the primal problem
        using variables and get dual variables
5     new_variable ← solve the pricing problem using
        dual_variables
6     variables ← variables ∪ {new_variable}
7 while new_variable has an negative reduced cost ;
8 return solution
```

At the beginning of the algorithm, an initial set of variables is defined (line 1) as a starting point so that the other relevant variables will be added to it. At each iteration, the primal model solves the restricted master problem using the selected variables (line 4). The current solution is thus found. Then, the pricing is solved to find the next variable to add to the primal to improve the quality of its solution (line 5). When the reduced cost of the new variable leads to an improving the quality of the primal solution, it is added to the list of selected variables (lines 6,7). Otherwise, the iterative process stops, and the solution found at this stage is optimal and is returned by the algorithm (line 8).

6.3.2.1 Example case: Cutting Stock problem

The cutting stock problem [GG61] is a well-known optimization problem relevant to illustrate the column generation process. The problem is formulated as follows:

Given
- a number of large wood boards of a length L
- a number of shelves of various sizes that need to be cut from the boards
- the demand for each shelf size

Find
- the smallest number of boards to cut in order to meet the demand for shelves

Given a set B of all boards and a set S of shelves per board, an intuitive model to formalize this problem is expressed as follows.

$$\begin{aligned}
&\min_{y,x} \sum_{b \in B} y_b && && (6.1)\\
&\text{s.t. } My_b \geq x_{sb} && \forall s, b && (6.2)\\
&\quad \sum_{s \in S} l_s x_{sb} \leq L && \forall b && (6.3)\\
&\quad \sum_{b \in B} x_{sb} \geq d_b && \forall s && (6.4)\\
&\quad y_b \in \{0, 1\} && \forall b && (6.5)\\
&\quad x_{sb} \in \mathbb{N} && \forall s, b && (6.6)
\end{aligned}$$

Equation 6.1 expresses the minimization of the sum of the number y_b of each selected board b. The variable y_b is a 0/1 variable stating whether the board b is selected or not to meet the total demand. The variable x_{sb} represents the number of shelf s to cut from the board b. Equation 6.2 expresses thus that no shelf cannot be cut from a board if the board is not selected. The constraint represented in Equation 6.3 is a capacity constraint: the sum of the length l_s of each shelf s cut from a board b should not exceed the maximum length L of the board. The last constraint states that the total number of a shelf s cut from the selected boards should not be less than the demand d_s for the shelf.

This kind of model is easy to understand. However, it may be difficult to solve using linear programming solvers, especially when the number of available boards is large. A better solving technique for this problem is column generation. The optimal solution for this problem generally involves a few boards. The column generation process allows only solving the model for these relevant boards (variables). Solving this problem using column generation comes down to finding the right boards of specific cutting configurations — a way to cut the board into shelves — to meet the total demand. Therefore, a primal problem is defined as in Equation 6.7.

$$\begin{aligned}
&\min_{x} \sum_{c \in C} x_c\\
&\text{s.t. } \sum_{c \in C} n_{cs} x_c \geq d_s \quad \forall s\\
&\quad x_c \in \mathbb{N} \quad \forall c
\end{aligned} \qquad (6.7)$$

This primal problem assumes a selected set C of boards with specific cutting configurations. For each configuration c in C, the variable x_c denotes the number of selected boards of this configuration. The objective function minimizes the sum of this number for each selected configuration and hence the number of selected boards. The only constraint of this model is that for each shelf s, the sum of the number n_{cs} of shelf s in the boards of configuration c should be at least equal to the demand for this shelf. This model is good but does not state how to find the configurations. Instead of generating all of them manually, a pricing model using the dual variables is proposed to generate the configurations to consider in the set C. As stated in Algorithm 10, each time the pricing is solved, it provides a new configuration to add to C to improve the objective value of the primal problem. Concretely, the objective function of the pricing problem is formulated based on the reduced cost. It is designed to produce a configuration with a negative reduced cost. Without going into details about the calculation of the reduced cost, note that a configuration with a negative reduced cost is found for this problem when the configuration fulfills the condition $1 - \sum_{s \in S} l_s n_s < 0$ of the dual problem. In this expression, n_s represents the number of shelf s inside the configuration. The pricing problem is expressed by Equation 6.8.

$$\begin{aligned} \min_{n} \; & 1 - \sum_{s \in S} l_s n_s \\ \text{s.t.} \; & \sum_{s \in S} l_s n_s \leq L \end{aligned} \tag{6.8}$$

The new configuration to generate is described by the number n_s of shelf s inside the configuration. The pricing solving leads thus in setting the value n_s for each shelf in the board. In addition to the fact that the configuration should produce a negative reduced cost, it should fulfil the capacity constraint; the sum of the length of shelves of the configuration should not exceed the length of the board.

A new configuration that will improve the primal score is found by solving this pricing problem. Then, the iterative process of column generation continues until there is no configuration fulfilling the negative reduced cost condition. In this example, the column generation process stops when the new configuration found respects the condition $\sum_{s \in S} l_s n_s \geq 1$. At this point, the optimal solution is found.

6.3.3 Existing techniques to optimize margins

DualLPboost [GS98] was the first approach to show that it is possible to solve boosting problems using a column generation approach. In it, at each iteration, the *dual* of the optimization problem over selected variables is solved. The solution of the dual provides new weights for the samples; these are used to find new trees. The primal is used to compute the weights of the different trees. The process is repeated for a fixed number of iterations.

Other approaches aim to correct the earlier discussed weaknesses of DualLPboost.

6.3.3.1 LPBoost

While LPBoost essentially chooses to maximize the lowest margin value, similarly to DualLPboost, it uses a soft margin maximization of the minimum margin instead of a hard margin maximization:

$$\begin{aligned} \max_{\boldsymbol{w},\boldsymbol{\xi},\rho} \ & \rho - D\sum_{i=1}^{M} \xi_i \\ \text{s.t. } & y_i H_{i:}\boldsymbol{w} + \xi_i \geq \rho, \ i = 1, \dots, M \\ & \sum_{t=1}^{T} w_t = 1 \\ & \xi_i \geq 0, \ i = 1, \dots, M \\ & w_t \geq 0, \ t = 1, \dots, T \end{aligned} \tag{6.9}$$

In this model ρ represents the minimum margin of the solution. The model includes a slack variable ξ_i for each instance $\boldsymbol{x}_i$. This value represents a tolerance to misclassification for each instance because it impacts the true value of margins. If $\xi_i > 0$, an instance is located on the wrong side of the chosen margin. The sum of the slack variables is minimized in the objective function, while maximizing the margin. This minimization is regularized by a parameter $D \in [0; 1]$ in order to control the trade-off between the size of the margin and the extent to which instances are wrongly positioned w.r.t. this margin. A good value of D is determined by hyperparameter tuning.

Note that in this model the sum over variables t is over *all possible trees* that can be constructed for a given dataset. LPBoost addresses this using a column generation process. The corresponding dual model allows finding at each iteration of the column generation process, the new sample weights $\boldsymbol{u} = \{u_1, \dots, u_M\}$ to learn a new tree. The formulation of the dual is then:

$$\begin{aligned} &\min_{r,\boldsymbol{u}} r \\ &\text{s.t.} \sum_{i=1}^{M} u_i y_i H_{it} \le r, \ \forall t = 1, \dots, T \\ &\qquad \sum_{i=1}^{M} u_i = 1 \\ &\qquad 0 \le u_i \le D, \ i = 1, \dots, M \end{aligned} \tag{6.10}$$

Solving this dual model allows to set weights to the instances based on the prediction of the current trees. Concretely, an instance i that is misclassified by a tree t will produce a score -1 for the expression $y_i H_{it}$. Thus, its corresponding weight u_i must be set to a strictly positive value to reduce the value of the variable r. On the contrary, it must be set to 0 when the expression $y_i H_{it}$ is 1. On the other hand, considering all the trees in the model, some may misclassify an instance while others may classify it correctly. Their overall impact on r for this instance therefore depends on the number of trees that predict the instance correctly or not. The more trees misclassify an instance, the higher its weight will be. Otherwise, it will have a lower value. Note that the variable u_i can take a value in the interval $[0; 1]$ and that the sum of all variables u_i is equal to 1. The weight of an instance will therefore also depend on the number of misclassified instances. In practice, an instance correctly classified by all trees will have a weight equal to 0. Note also that r is an unbounded variable and will take a value less than 0 as long as there is a misclassified instance.

After solving this model, the weights representing the misclassification costs of instances at the current time are found. These weights are thus used to learn a new tree, mainly focusing on the misclassified instances. Unfortunately, a traditional decision tree learning algorithm is used in each iteration to find the decision tree that takes into account the weights computed for the instances. However, given that these algorithms are heuristic, LPBoost cannot decide at which moment an optimal ensemble has indeed been found.

6.3.3.2 MDBoost

Reyzin and Schapire [RS06] stated that the minimum margin maximization is performed at the expense of the other margins. Therefore MDBoost maximizes the unnormalized average margin and simultaneously minimizes the variance of the margin distribution under the assumption that the margin distribution follows a Gaussian distribution [SL10]. The primal objective function of MDBoost for T trees is $\max_{\boldsymbol{w}} \ \bar{\varrho} - \frac{1}{2}\sigma^2, \quad \text{s.t. } \boldsymbol{w} \geq \mathbf{0}, \mathbf{1}^\top \boldsymbol{w} = D$, where $\bar{\varrho}$ is the mean of the unnormalized margins over the training data, σ^2 is the variance and D is a regularization parameter which indicates whether the model mostly concentrates on the average margin maximization or the variance minimization. After some transformations, the primal can be rewritten as follows:

$$
\begin{aligned}
\max_{\boldsymbol{w},\boldsymbol{\varrho}} \ & \mathbf{1}^\top \boldsymbol{\varrho} - \frac{1}{2}\boldsymbol{\varrho}^\top A \boldsymbol{\varrho} \\
\text{s.t. } & \boldsymbol{\varrho}_i = y_i H_{i:} \boldsymbol{w}, \ \forall i = 1, \dots, M \\
& \mathbf{1}^\top \boldsymbol{w} = D \\
& \boldsymbol{w} \geq \mathbf{0}
\end{aligned}
\tag{6.11}
$$

A is a symmetric matrix with all entries on the main diagonal set to 1, and the other values are set to $\frac{-1}{M-1}$. This matrix allows the variance calculation and is positive semidefinite. The model is hence a convex quadratic problem in $\boldsymbol{\varrho}$. The corresponding dual model to reweigh the training samples is then:

$$
\begin{aligned}
\min_{r,\boldsymbol{u}} \ & r + \frac{1}{2D}(\boldsymbol{u} - \mathbf{1})^\top A^{-1} (\boldsymbol{u} - \mathbf{1}) \\
\text{s.t. } & \sum_{i=1}^{M} u_i y_i H_{it} \leq r, \ \forall t = 1, \dots, T
\end{aligned}
\tag{6.12}
$$

Both LPBoost and MDBoost require solving a same pricing problem. Indeed one can see in their dual formulation (6.10) and (6.12) that identifying a violated constraint corresponds to finding a decision tree that minimizes the weight (given by the dual variables considered as fixed) of correctly classified instances minus the weights of incorrectly classified ones. The column generation alternates the two steps of 1) solving the primal problem and collecting the dual values and 2) solving the pricing problem which amounts of finding an optimal weighted decision tree that is then added to the set of trees considered by the primal.

6.4 Learning optimal forests

This section describes our approach to learn provably optimal decision forests.

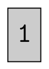
6.4.1 Problems in the existing approaches

The different approaches we described in the previous sections were all proposed to achieve better performances than Adaboost by optimizing a global criterion. Crucially, although the column generation process used by these approaches allows solving a problem until optimality, two elements in the solving process annihilate the optimality of solutions found.

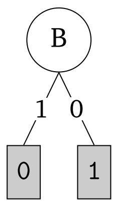
Table 6.1: Example database

A	B	C	class
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

Heuristic-based learners The algorithms used by LPBoost and MD-Boost for the pricing problem at each iteration of the column generation process are heuristic-based, that is, the algorithm does not provide any guarantee that it finds a tree to improve the objective, even when one exists. Consider the example dataset in Table 6.1 and a sample weight distribution $\boldsymbol{u} = (0.05, 0.06, 0.33, 0.02, 0.09, 0.02, 0.22, 0.04, 0.02, 0.08, 0.07)$. The tree found by the CART implementation of Scikit-learn [Ped+11] for a maximum depth equal to 1 is represented in Figure 6.1a. It always predicts the class 1 and thus misclassifies 5 instances. The tree minimizing the misclassification rate for these weights the most is represented in Figure 6.1b and misclassifies 4 instances. Therefore, using a heuristic-based tree in a column generation process does not guarantee optimality. In this case, if the next tree to be added to the ensemble must have at least the quality of the optimal tree, the CART algorithm cannot find it and will stop the process without reaching optimality. To solve this problem, we have to use an algorithm to learn an optimal decision tree and taking into account the sample weights.



(a) CART tree



(b) Optimal tree

Figure 6.1: Tree found for a max depth = 1, given the database of Table 6.1 and the sample weight u.

The number of iterations In the different approaches, a maximum number of iterations is set after which the column generation process is stopped and the result at this stage is used. At that moment we have no guarantee that the result found is optimal. To deal with this problem, we need to let the algorithm run until no tree can be added, i.e. no tree can be found violating a dual constraint.

6.4.2 Optimal weighted decision trees (OWDT)

We build on the recent techniques for finding ODTs. However, to the best of our knowledge, there does not exist any efficient algorithm to learn ODTs in which instances are weighted. Therefore we adapted the DL8.5 algorithm from Chapter 4 as it is the most efficient up-to-date algorithm to learn ODTs. The only important thing to handle in order to integrate the sample weights in DL8.5 is the way the score is computed in each leaf. Basically, the formula used in DL8.5 to compute the error in a leaf covering a set ℓ of instances is:

$$error(\ell) = \mid \ell \mid - \max_{c \in \mathcal{C}} \mid \{i; i \in \ell \wedge y_i = c\} \mid \tag{6.13}$$

To handle sample weights, the error function must integrate the weight of each instance falling in a leaf. The new error is the difference between the sum of weights of instances in a leaf and the maximum sum of weights of instances grouped by class. It is expressed as:

$$weighted\_error(\ell) = \sum_{i \in \ell} w_i - \max_{c \in \mathcal{C}} \sum_{i \in \ell \wedge y_i = c} w_i \tag{6.14}$$

The error in each leaf of DL8.5 is computed based on counting. This is efficiently implemented using bit-wise operations and the *Reversible Sparse bitset* [Dem+16] data structure (Chapter 3), where each bit in the structure represents an instance. Unfortunately, the reversible sparse bitset data structure is more appropriate for counting problems than enumeration ones. However, to handle the weighted error, it is important to loop on each weight. This data structure needs to be adapted in our case to iterate over the true bits of the bitsets and collect the weight of the corresponding instance. Instead of checking each bit of the reversible sparse bitset, we implement this efficiently using a bitwise operation to retrieve the index of the rightmost non-zero bit. This operation coupled to *shift* operation allows to iterate only over the covered instances.

6.4.3 Our approach to learn optimal forests

We summarize our generic algorithm for learning optimal decision forests, *Optiboost*, in Algorithm 11. This algorithm can be used for various optimization criteria, such as the ones proposed in LPBoost and MDBoost. We call it *OptiLPBoost* or *OptiMDBoost* when it is used respectively with the LPBoost or MDBoost model. The distinguishing feature of *Optiboost* is that we use an optimal weighted decision tree learning algorithm for finding the new trees to add in the column generation process that respects the customized sample weights. Moreover, we continue the search until no new tree can be found to improve the value of the optimization criterion.

Algorithm 11: Optiboost($\{(\boldsymbol{x}_i, y_i)\}, depth, D$)

```
1  $\boldsymbol{w}, \mathcal{H}', r \leftarrow \mathbf{0}, \{\}, -\infty$
2  $u_i \leftarrow 1, i = 1, \ldots, M$
3  $h \leftarrow$ OWDT($\{(\boldsymbol{x}_i, y_i)\}, depth, \boldsymbol{u}$)
4  do
5      $\mathcal{H}' \leftarrow \mathcal{H}' \cup h$
6      $r, \boldsymbol{u} \leftarrow$ solve the dual (6.10) for LPBoost and (6.12) for MDBoost
        using $D$
7      $h \leftarrow$ OWDT($\{(\boldsymbol{x}_i, y_i)\}, depth, \boldsymbol{u}$)
8  while $\sum_{i=1}^{M} u_i y_i h_{\boldsymbol{x}_i} > r$;
9  $\boldsymbol{w} \leftarrow$ solve the primal (6.9) for LPBoost and (6.11) for MDBoost using
    $D$
10 return $\mathcal{H}', \boldsymbol{w}$
```

A second distinguishing feature of the algorithm is that we do not fix the size of the ensemble in advance. Instead, we repeat the column generation process until the OWDT learning algorithm concludes that no tree can be added to the ensemble that violates a constraint of the dual. At that moment, the optimality of the ensemble has been proven and the weight of each decision tree is returned.

We start the process with an OWDT with all equal sample weights $\boldsymbol{u}$ set to 1 (line 2) which means that the first tree added to the ensemble optimizes its accuracy. The dual is then solved to update the variables r and the vector $\boldsymbol{u}$. The next iteration restarts with the new weights. If the new tree added does not violate any dual constraint, the iteration stops. At this point the optimal forest is found. The primal model needs to be solved to find the weight of each tree of the final forest. However, these weights can be found using a by-product if the solver provides this feature. The final trees and their weights are returned.

6.5 Experiments

In this section, we answer the two questions:

Q1 What is the impact on the objective value when embedding an ODT algorithm rather than a greedy one such as CART for solving the pricing problem in LPBoost and MDBoost?

Q2 Does it make a difference in practice on the generalization error to use an ODT learning algorithm?

To solve the linear and quadratic models of LPBoost and MDBoost, we use Gurobi [Gur21] Optimizer 9.1.0 as solver. For the different base learners, we used the implementation of CART available in Scikit-learn [Ped+11] as heuristic DT algorithm. For the optimal DT algorithm, we used our customized DL8.5 [ANS20] algorithm. Experiments were run on a Red Hat 4.8.5-16 server with an Intel Xeon E5-2640 CPU and 128GB of memory.

To obtain the results in this section, we performed a 5-fold cross validation. As the different models use a regularization parameter, for each fold, we used grid search coupled to a 4-fold cross validation as hyperparameter tuning technique to find the best parameter. The different regularization parameters tested for LPBoost and MDBoost are respectively {0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1} and {0.5, 1, 3, 6, 12, 25, 50, 100}. The complexity of MDBoost increases as the number of instances increases. Indeed, the size of the matrix A depends on the number M of instances. Considering the number of iterations and the fact that the model must be solved at each iteration and also considering that the complexity of the OWDT algorithm increases depending on the size of the dataset, we performed the experiments on 10 well-known UCI[1] datasets having reasonable size. The datasets were already prepro-

[1] https://archive.ics.uci.edu/ml/datasets.php

cessed and available on the CP4IM[2] project website. The different base learners used for the experiments have a maximum depth equal to 3.

To answer question **Q1**, we present in Table 6.2 the optimization gap between ODT (OptiLPBoost, OptiMDBoost) and heuristic (LPBoost and MDBoost) based algorithms. The first columns of the table describe the different datasets. *nFeat* stands for the number of features and *nTrans* represents the number of instances in the dataset. To determine the best value for the hyperparameter D, we performed hyperparameter tuning. A hyperparameter tuning has been performed for bold columns. The same value of the hyperparameter is then used for the other approach, shown in the adjacent column. We notice clearly in Table 6.2 that the optimal value reached by our approach is better than classic LPBoost and MDBoost. The optimization value reached is on average improved by an order of 2.

Table 6.2: Comparison of the objective reached when using the same hyperparameters

Dataset	nFeat	nTrans	**LPBoost**	OptiLPBoost	**OptiLPBoost**	LPBoost	**MDBoost**	OptiMDBoost	**OptiMDBoost**	MDBoost
anneal	93	812	**0.0**	**0.0**	**0.0**	**0.0**	44.57	**82.15**	**165.43**	103.4
audiology	148	216	0.4	**0.62**	**0.62**	0.4	103.57	**135.94**	**142.73**	118.32
heart-clev.	95	296	0.11	**0.34**	**0.34**	0.11	89.93	**122.06**	**133.29**	104.31
hepatitis	68	137	0.29	**0.58**	**0.58**	0.29	38.12	**69.07**	**71.11**	42.01
lymph	68	148	0.28	**0.55**	**0.55**	0.28	46.28	**77.57**	**77.08**	47.54
prim.-tum.	31	336	**0.0**	**0.0**	**0.0**	**0.0**	59.08	**79.75**	**90.1**	67.37
soybean	50	630	0.01	**0.13**	**0.13**	0.01	104.06	**189.68**	**274.37**	171.06
tic-tac-toe	27	958	0.11	**0.28**	**0.28**	0.1	105.39	**196.59**	**190.45**	104.1
vote	48	435	0.23	**0.46**	**0.46**	0.23	154.89	**223.69**	**239.1**	179.92
zoo-1	36	101	**1.0**	**1.0**	**1.0**	**1.0**	**80.8**	**80.8**	**80.8**	**80.8**

We can also notice in Figure 6.2 and Figure 6.3 how the objective value evolves as the iterations increase in the boosting process on different folds. We show these plots for the dataset *tic-tac-toe* as it has the highest number of instances in our datasets and the resulting graphs are representative for the results on other datasets as well. For space reasons, we only show the first three folds of our 5 folds. The same regularization parameter is used for the different approaches. On these plots, the parameters used have been tuned for LPBoost and MDBoost. Notice that the number of iterations reached by our optimal approach is always greater than the classic approaches. This indicates that the

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

heuristic approaches stop without proving the inexistence of trees. Note that already from the first iterations, the objective value of our approach is better than the classic LPBoost and MDBoost. Moreover, we can notice that, similarly to Adaboost, the optimization criterion of LPBoost and MDBoost does not focus on accuracy. The number of trees increases even when the accuracy on the training set is 100%, trying to increase the margin of instances.

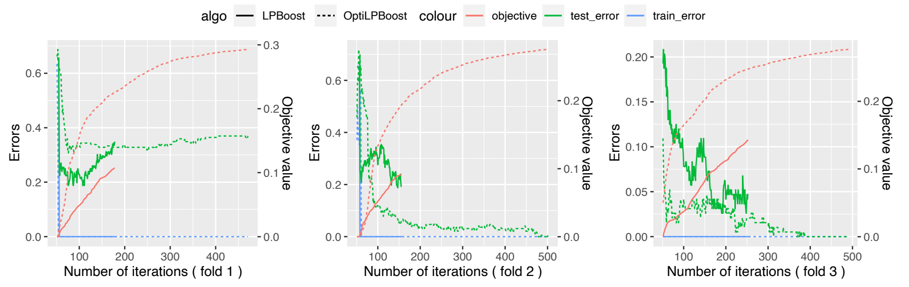


Figure 6.2: Evolution of objective value with LPBoost on *tic-tac-toe* dataset

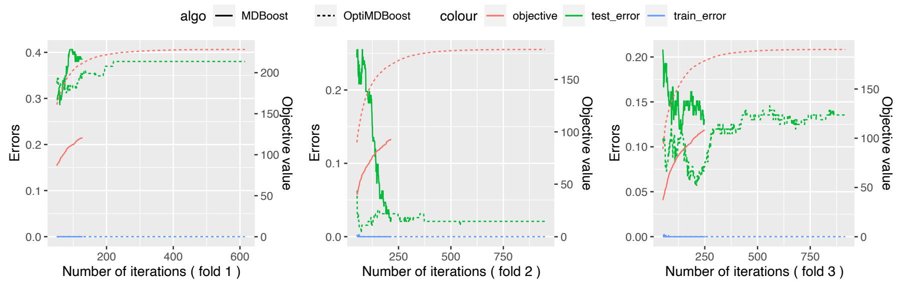


Figure 6.3: Evolution of objective value with MDBoost on *tic-tac-toe* dataset

To answer question **Q2**, we show in Table 6.3 a performance comparison of classic LPBoost and MDBoost against our optimal forest approaches, where the regularization value is tuned for each approach. For each approach, we present the number of trees found, the accuracy on training set and test set. The best accuracy on the test set is in bold. The column Adaboost will be discussed later. Not surprisingly, and as explained above, the number of trees found with OptiLPBoost and OptiMDBoost are higher than LPBoost and MDBoost. However, this is not always the case because this number also depends on the value of the regularization parameter. According to the authors of LPBoost and MDBoost, increasing the margin of instances should also reduce the generalization error of the model on unseen instances. The results in Table

Table 6.3: Comparison of generalization performance

Dataset	OptiLPBoost			LPBoost			Adaboost	OptiMDBoost			MDBoost			Adaboost
	n_trees	train_acc	test_acc	n_trees	train_acc	test_acc		n_trees	train_acc	test_acc	n_trees	train_acc	test_acc	
anneal	14.4	61.43	**61.43**	2.0	27.15	27.22	88.79	205.0	92.98	**88.92**	75.8	91.62	87.93	88.79
audiology	31.6	100.0	**95.82**	21.2	100.0	93.96	93.53	43.2	99.77	93.5	25.2	99.08	**94.89**	93.53
heart-clev.	119.2	100.0	78.02	55.8	100.0	**78.36**	79.39	89.4	96.03	78.36	23.2	93.41	**81.41**	79.39
hepatitis	55.2	100.0	**82.49**	29.2	100.0	80.32	82.46	188.6	99.63	**83.23**	30.2	99.45	81.83	82.46
lymph	64.2	100.0	**89.15**	31.6	100.0	85.82	89.79	114.2	99.66	**89.17**	21.8	97.47	85.13	89.79
prim.-tum.	29.0	88.02	**71.46**	29.4	78.12	69.33	77.37	110.2	92.04	**82.74**	27.8	90.48	82.14	77.37
soybean	58.6	98.97	93.81	26.4	99.17	**94.13**	95.71	129.4	98.13	95.71	53.4	97.86	**96.03**	95.71
tic-tac-toe	153.2	100.0	**89.47**	98.8	100.0	82.89	84.99	875.8	100.0	**87.49**	204.0	100.0	87.28	84.99
vote	58.4	100.0	**94.25**	28.8	100.0	**94.25**	95.63	163.6	99.48	**96.09**	40.6	98.85	95.63	95.63
zoo-1	1.0	100.0	**97.0**	1.0	100.0	**97.0**	100.0	1.0	100.0	**97.0**	1.0	100.0	**97.0**	100.0

6.3 show that this is not always the case. Although the generalization error of LPBoost and MDBoost decreases on most datasets with OptiLP-Boost and OptiMDBoost, there are still some datasets on which boosting ODTs perform worse. This is confirmed by the Figures 6.2 and 6.3, where the test error obtained by OptiLPBoost and OptiMDBoost on some folds is higher than errors obtained by classic approaches, although the same regularization values are used and the objective reached by the optimal approaches are higher. In conclusion, there is a difference between boosting a traditional heuristic algorithm compared to ODTs. Concretely, boosting ODTs can increase performance, but not on all datasets. The result also depends on the optimization criterion. Notice that the results of MDBoost are better than LPBoost. This implies that it is more interesting to maximize all the margins than maximizing the lowest one.

Due to the fact that LPBoost and MDBoost are not optimal, it was not possible to assess their optimization criterion against Adaboost. In Table 6.3, we also compare the different approaches to Adaboost with 100 trees. The accuracy found on test sets are put in the column *Adaboost*. We compare the results according to whether it is LPBoost or MDBoost. In each case, we compare the classic approach, the optimal approach, as well as Adaboost; the best one is underlined. Notice that the performance of LPBoost models is lower than Adaboost even when the optimal forest approach managed to outperform Adaboost on 3 instances. On the other hand, concerning the MDBoost model, the optimal forest approach outperforms the classic approach as well as Adaboost on many datasets. We can conclude that it is a good approach to infer optimal forests by boosting ODTs. However, the performance is mainly ensured by the optimization criterion. A good optimization criterion will result in a good generalization.

6.6 Conclusions

In this chapter, we explored the question of learning optimal forests. We presented existing techniques to boost DTs and the intuition on which they rely. We demonstrated that the results found by these techniques are not optimal. We then built on recent works on learning ODTs to propose an ODT algorithm supporting sample weights. We built on this OWDT algorithm to propose a way to find optimal forests. For the first time, this allowed us to infer optimal decision forests given a global optimization criterion. We then performed a number of experiments to (1) show that there is a gap between optimal forests and existing approaches. (2) We compare optimal forests to existing approaches and found that optimal forests increase the generalization character of these models. (3) We found that in addition to optimality, the optimization criterion is important to learn ensembles with a good generalization character. Although we noticed that maximizing all margins is preferable over maximizing the lowest one and can outperform Adaboost, it appears that the existing criteria are not yet very satisfactory. We hope this work could launch the study of other objective functions, as we made our implementation easy to support other optimization criteria.

PyDL8.5: an extendable tool for learning ODTs 7

This chapter is based on the paper Aglin, S. Nijssen, and P. Schaus. “PyDL8.5: a library for learning optimal decision trees”. In: Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence. 2021, pp. 5222–5224.

7.1 Introduction

Machine learning models are increasingly used in a wide range of applications. However, an increasing concern is the interpretability of machine learning models. Whether or not human experts can understand a model can for instance be important to avoid ethical problems [CS14; RSG18; RSG16a; RSG16b; INM19]. Decision trees have gained interest in recent years for this reason, since they can be interpreted as rules that are interpretable by domain experts.

An important parameter for the interpretability of a decision tree is the depth of the tree. Deeper trees contain more tests and more rules, and hence are often harder to interpret.

As explained in the previous chapters, the most well-known algorithms such as CART [Bre+84] and C4.5 [Qui93] use a greedy approach to learn DTs. They select iteratively the best feature to split on based on a heuristic (information gain, gini index) and continue this splitting process until a desired quality (eg., classification accuracy) is obtained or a desired depth is reached. However, this greedy process has disadvantages. Greedy trees that are sufficiently accurate are sometimes unnecessarily deep, and depth-constrained greedy trees are not sufficiently accurate [BD17]. Moreover, to learn trees for other tasks than classification, it is often necessary to develop new heuristics [BRR98], making it harder to solve such tasks.

The main challenge in finding ODTs is the NP-hardness of finding ODTs. Good search algorithms are needed to make solving this task feasible. We showed that DL8.5 obtains the best performance on a wide range of test cases [ANS20]. Indeed, on commonly used UCI datasets DL8.5 computes ODTs within seconds, making its use in an interactive demo possible. Moreover, we show in the previous chapter that it can be used to solve another problem than the one of misclassification rate minimization.

In this chapter, we introduce PyDL8.5, an open source Python library that implements DL8.5 in an efficient and extendible manner. Compared to other systems for finding ODTs, it offers these advantages: (1) it offers better computational performance, as shown in chapter 4 [ANS20]; (2) it is easy to use, as it is fully compatible with scikit-learn; (3) it extends

ODTs to several well-known problems; (4) it can easily be extended to solve other tree learning problems than classification problems.

To deal with other learning problems, we implemented DL8.5 such that it works for any optimization criterion that is *additive*; i.e. the error produced by a node is the sum of errors produced by its children. Using our library, a user can express her optimization criterion using any Python library of choice; the optimization criterion does not need to be linear or integer, as required in alternative methods based on MIP and SAT solvers.

This chapter is organized as follows. In the next section, we remind the essentials of DL8.5 by proposing a summary of the algorithm. Then, we present a bit the architecture of PyDL8.5 library. Finally, we propose a non-exhaustive list about examples of how to use PyDL8.5 on some ML tasks.

7.2 Learning ODTs using DL8.5

Given a binary training set $\mathcal{D}$, and two parameters $maxdepth$ and $minsup$, the problem that is solved by the DL8.5 algorithm is to find the decision tree $\arg\min_{T \in \mathcal{DT}} f(T)$, where

- $\mathcal{DT}$ represents all decision trees of depth lower than or equal to $maxdepth$ in which each leaf covers at least $minsup$ examples of the training data $\mathcal{D}$;

- $f(T)$ is a scoring function that is additive over the leafs of the tree T, that is, f can be written as $f(T) = \sum_{\ell \in leafs(T)} g(\ell)$, where $g(\ell)$ is a function that evaluates the quality of each leaf ℓ, and $g(\ell)$ is independent of the order of the tests leading to leaf ℓ.

In [ANS20] DL8.5 was introduced for the problem of classification; a scoring function $g(\ell)$ was used that evaluates the misclassification rate of a leaf, that is, the number of instances covered by a leaf that do not belong to the majority class of that leaf. However, the algorithm is correct for other additive scoring functions as well.

DL8.5's search algorithm relies on a mix of Dynamic Programming and Branch-and-Bound search. It recursively explores all possible splits and selects the split with the lowest score. Since different sequences of splits may select the same set of instances in the data, the same subset of data may be encountered multiple times during the search. DL8.5 uses a caching system to reuse results of subsets already assessed. To prune the search space, DL8.5 uses a bounding system. When a subtree is found, its score is used as an upper bound to restrict the quality of future subtrees. Here, DL8.5 exploits the additive nature of the scoring function to prune a right-hand subtree when the left-hand subtree provides an error greater than or equal to the upper-bound. On the contrary, when the upper bound is too restrictive to find a solution, it is considered as a lower bound.

7.3 Architecture of PyDL8.5

Given the popularity of Python in data science and AI, we implemented PyDL8.5 as a Python 3 library. The core of the algorithm is written in C++ so PyDL8.5 is featured as a Python wrapper over the C++ implemetation of DL8.5 as well as its extensions proposed in the previous chapters. However, PyDL8.5 also provides numerous features that extend the DL8.5 algorithm. PyDL8.5 implements the `fit/predict` interface of the popular scikit-learn library [Ped+11] to make it easy to use in combination with scikit-learn. An important component of DL8.5 is the scoring function used to evaluate the leafs of a tree. For the most common scoring functions, a fast implementation in C++ is provided. This is important as the function is called very often. Thanks to DL8.5, users can also write a scoring function in Python, although such functions may slow down the execution.

The library is hosted on PyPI[1] and the source code is available at https://github.com/aia-uclouvain/pydl8.5 under MIT license. It can be installed by running the command `pip install pydl8.5`. The documentation of the library is available at https://pydl85.readthedocs.io/en/latest/.

The following examples demonstrate how easily PyDL8.5 can be used to implement many different ODT learning tasks.

7.4 Examples of PyDL8.5 use

In this section, we present how PyDL8.5 can be used to learn many types of ODTs based on the DL8.5 algorithm.

The first example we show is the basic implementation of the DL8.5 algorithm.

Example Task 1: Shallow Classifiers. Listing 7.1 shows the code needed to train an ODT classifier under a maximum depth and a minimum support constraints. The code also allows predicting on unseen

[1]https://pypi.org/project/pydl8.5/

data. Note the simple integration in scikit-learn. Indeed, a scikit-learn function is used to split the dataset into a training and test sets. Moreover, the outputs of the code are compatible with scikit-learn metric functions. Notice that the lines from 1 to 9 are only for preprocessing purpose. Only the three lines (12-14) are really important to learn the ODT and predict on unseen data. The score minimized by default by `DL85Classifier` is the misclassification rate. Notice the presence of parameters mentioned in the previous chapters. `max_depth`, `min_sup` and `time_limit` are parameters mentioned in Chapter 4. `max_depth`, `min_sup` are constraints set on the tree to be found. `time_limit` is the parameter denoting the anytime behavior of the DL8.5 algorithm. In the code proposed, it is to 60, meaning that the search will be interrupted after 60 seconds if the ODT is not found. In this case, the current ODT found is returned. The parameters `maxcachesize` and `wipe_factor` are introduced in Chapter 5. They are related to the reduction of the memory consumption. `maxcachesize` denotes the maximum size of the cache, and `wipe_factor` indicates the percentage of the cache to remove when the cache is full. There are also possibilities to change the type of the cache (trie or hash table) and the representation used as the key in the cache (paths or set of instances). These options can be found in the online documentation.

```python
1 import numpy as np
2 from sklearn.model_selection import train_test_split
3 from sklearn.metrics import confusion_matrix, accuracy_score
4 from pydl85 import DL85Classifier
5 # read the dataset and split into features and targets
6 dataset = np.genfromtxt("anneal.txt")
7 X = dataset[:, 1:] # get the features matrix
8 y = dataset[:, 0] # get the class vector
9 # split the dataset into training and test sets
10 X_train,X_test,y_train,y_test = train_test_split(X, y, test_size=0.2)
11 # initialize the classifier, train and predict
12 clf = DL85Classifier(max_depth=3, min_sup=5, time_limit=60,
       maxcachesize = 10000, wipe_factor = 0.3)
13 clf.fit(X_train, y_train) # train the model
14 y_pred = clf.predict(X_test) # predict on new data
15 print("Accuracy on test set =", accuracy_score(y_test, y_pred))
16 print("Confusion Matrix\n", confusion_matrix(y_test, y_pred))
```

Listing 7.1: Code snippet to train a classifier

Given the branch-and-bound nature of DL8.5, it can be interesting to provide an initial upper-bound on the quality of the tree to find, to help the search. A concrete example is to run a heuristic tree learning algorithm, for instance CART, to quickly find a tree and to use its error to bound the search. This feature isn provided by PyDL8.5. It can be easily used by providing the parameter `max_error` to the `DL8Classifier`. This feature can also be used to solve some ML tasks. An example of a problem that can be solved using this feature is the problem of finding the shallowest tree with perfect accuracy that has been studied recently. The problem was solved using a SAT solver [Nar+18; Ave20].

Example Task 2: 100% Accurate Classifiers. The following code shows how this problem can be solved using PyDL8.5. The idea consists in searching the ODT for successive depths starting from 1 until an error of 0 is met. To speed up the search, an upper bound representing a maximum error not reachable is set using the parameter `max_error`. In this case, a good upper bound is 1, denoting that we want an error lower than 1, i;e. 0. The fact that the upper bound is very restrictive, induces a huge number of pruning. We found that this simple piece of code typically solves the same problem more rapidly than SAT-based approaches.

```python
import numpy as np
from sklearn.model_selection import train_test_split
from pydl85 import DL85Classifier
# read the dataset and split into features and targets
dataset = np.genfromtxt("anneal.txt")
X = dataset[:, 1:] # get the features matrix
y = dataset[:, 0] # get the class vector
# split the dataset into training and test sets
X_train,X_test,y_train,y_test = train_test_split(X, y, test_size=0.2)
# loop for each depth
depth = 1
while True:
  clf = DL85Classifier(max_depth=depth, max_error=1)
  clf.fit(X_train, y_train)
  if clf.error_ == 0:
    break
  maxdepth += 1
y_pred = clf.predict(X_test)
```

Listing 7.2: Code snippet to train the shallowest 100% accurate DT

As explained above, the DL8.5 algorithm is suitable for finding trees for which the total error can be expressed as a sum over the errors of all leafs. As many ML problems meet this requirement, we provide through PyDL8.5 an interface to allow users to define their own optimization criteria. An example of a task fulfilling this requirement is predictive clustering [BRR98]. This is an unsupervised task in which one aims to identify clusters of good quality in the leafs of the tree and the tree can be interpreted as a description of the clusters. To handle this problem, PyDL8.5 provides a `DL85Predictor` class in addition to the DL85Classifier class, each of them having an interesting parameter: error_function. This input provides the function used by DL8.5 to compute the error produced by the different paths assessed during the search. To easily define this function, DL8.5 provides to the function the list of instances covered by the path. An additional parameter is added to the DL85Predictor class: leaf_value_function. It is used to define a function to set a label to the leaf assessed. In DL85Classifier class, the error_function can directly return the class in addition to the error value. It can be a class for a classification problem or a label for a clustering problem. This function also receives from DL8.5, the list of transactions falling in the leaf to label. Note that since the error computation is performed in Python, it slows down the execution compared to if it was done in C++.

Example Task 3: Predictive Clustering. Listing 7.3 shows how to use the features of PyDL8.5 to implement predictive clustering using a custom scoring function error that calculates the sum of Euclidean distances from each point in a cluster to the centroid. Note that no heuristic is needed, that this function is nonlinear, and is written using NumPy code itself. A leaf_value function is provided to determine the labels put in the leafs of the tree. The centroids of clusters are used as labels. This feature provided by PyDL8.5 can be used to learn regression trees optimizing a criterion like the mean absolute error (MAE). This can also be used to learn weighted instances trees mentioned in Chapter 6 even if the resolution of this problem is directly provided by the core implementation of DL8.5 written in C++. For this, the parameter sample_weights should be used with the DL85Classifier class.

```python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neighbors import DistanceMetric
from pydl85 import DL8Predictor
eucl_dist = DistanceMetric.get_metric("euclidean")
# user-defined scoring function
def error(tids):
  X_subset = X_train[list(tids),:]
  centroid = np.mean(X_subset, axis=0)
  distances = eucl_dist.pairwise(X_subset, [centroid])
  return float(sum(distances))
# user-defined labels in the leaf
def leaf_value(tids):
  return np.mean(X_train.take(list(tids)))
# read the dataset and split into features and targets
dataset = np.genfromtxt("anneal.txt")
X, y = dataset[:, 1:], dataset[:, 0]
# split the dataset into training and test sets
```

```
X_train,X_test,y_train,y_test = train_test_split(X, y, test_size=0.2)
# initialize the search and run it
clf = DL85Predictor(max_depth=3, min_sup=5, error_function=error,
    leaf_value_function=leaf_value)
clf.fit(X_train)
predicted = clf.predict(X_test)
```

Listing 7.3: Code snippet of user-defined clustering

To speed up a bit the learning of user-specific trees, PyDL8.5 offers a special interface for an error function that relies on the number of instances per class. For this, a parameter `fast_error_function` is provided through the `DL85Predictor` and `DL85Classifier` classes. This parameter allows the user to pass a function to compute the error value of paths assessed. In contrast to the parameter `error_function` that relies on the list of instances covered by the path, the `fast_error_function` parameter relies on the numbers of instances per class related to the path. For instance, to compute the misclassification rate, the complexity of using the supports per class depends on the number of classes, which is lower than the number of instances covered by the path. This feature can be used to learn cost sensitive trees. The problem of cost sensitive learning is similar to misclassification rate minimization, except the fact that the cost of predicting a class in place of another one is different for each pair of classes.

Example Task 4: Cost-sensitive Learning. Listing 7.4 shows how to use the `fast_error_function` provided by PyDL8.5 to learn cost-sensitive trees. The function `error` passed as input to the `DL85Predictor` class (line 17), takes as input for each path, the supports per class (line 7). The classic misclassification rate is computed, and it is timed by the cost of predicting the minority class as the majority class (line 10). Notice also that using the `DL85Predictor` class, the class of the leaf error is directly provided using the user-defined function; here `maxindex` (line 10). This feature of PyDL8.5 can also be used to solve many other tasks, such as fair tree learning.

```
1 import numpy as np
2 from sklearn.model_selection import train_test_split
3 from pydl85 import DL85Predictor
4 # misclassification costs
5 cost = [[1, 3], [2, 1]]
6 # user scoring function: weighted misclassification rate
7 def error(sup_iter):
8   supports = list(sup_iter)
9   maxindex = np.argmax(supports)
```

```
  return (sum(supports) - supports[maxindex]) *
      cost[maxindex][abs(maxindex-1)], maxindex
# read the dataset and split into features and targets
dataset = np.genfromtxt("anneal.txt")
X, y = dataset[:, 1:], dataset[:, 0]
# split the dataset into training and test sets
X_train,X_test,y_train,y_test = train_test_split(X, y, test_size=0.2)
# initialize the search and run it
clf = DL85Predictor(max_depth=3, fast_error_function=error)
clf.fit(X_train)
predicted = clf.predict(X_test)
```

Listing 7.4: Code snippet of user-defined clustering

Many other tasks such as the boosting discussed in Chapter 7 are made available through the PyDL8.5 library. The online documentation can be consulted to get a further insight on the use case of PyDL8.5. However, a last interesting feature of PyDL8.5 is worth to be mentioned here: it is the printing of the learnt ODTs.

Example Task 5: Visual representation of learnt ODT. Listing 7.5 shows how to learn an ODT and display its visual representation. A function `export_graphiz` is provided by the `DL85Classifier` class. It returns a .dot string that represents the learned tree. It can be used by the `graphviz` library to convert the string to an image.

```
import numpy as np
from sklearn.model_selection import train_test_split
from pydl85 import DL85Classifier
import graphviz
# read the dataset and split into features and targets
dataset = np.genfromtxt("anneal.txt")
X, y = dataset[:, 1:], dataset[:, 0]
# split the dataset into training and test sets
X_train,X_test,y_train,y_test = train_test_split(X, y, test_size=0.2)
# instance the classifier, fit and predict
clf = DL85Classifier(max_depth=2, min_sup=1)
clf.fit(X, y)
y_pred = clf.predict(X_test)
# print the tree
dot = clf.export_graphviz()
graph = graphviz.Source(dot, format="png")
graph.render("anneal_odt")
```

Listing 7.5: Code snippet of user-defined clustering

Figure 7.1 shows the visual representation of the tree learned in the Listing 7.5.

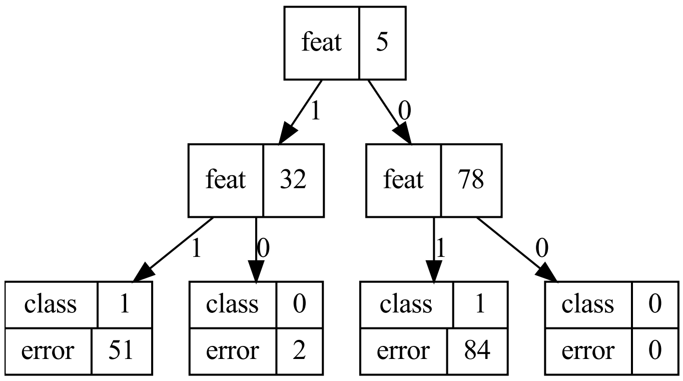


Figure 7.1: Visual representation of an ODT

Part IV

Conclusion

Conclusion 8

The conclusion of this thesis consists of 3 parts. Section 8.1 presents a summary of the thesis. In Section 8.2, an overview of DL8.5 applications in real-world situations is shown. Finally, Section 8.3 presents a discussion of some future works.

8.1 Summary

In this thesis, we studied four main problems:

1. The learning of optimal decision trees under a maximum depth and a minimum support constraints,
2. The learning of optimal decision trees under memory constraints,
3. The use of optimal decision trees for solving and assessing boosting models,
4. The learning of optimal decision trees optimizing user-defined error criteria.

The starting point is two earlier works:

(1) The study performed by Bertsimas and Dunn [BD17] showed experimentally that the optimal decision trees under a maximum depth constraint generalize better than the heuristic decision trees. However, [BD17] had not managed to propose an approach to learn ODTs within a reasonable time. This is mainly due to the use of a MIP solver that is generic and cannot take into account the specificity of the decision tree learning problem. Moreover, Laurent and Rivest [LR76] have shown that it is NP-complete to learn optimal decision trees under a size constraint.

(2) To learn optimal decision trees in a shorter period of time, we mainly relied on an earlier work DL8 [NF07; NF10]. This algorithm is a dynamic programming algorithm based on the caching and the reuse of solutions of subproblems encountered multiple times during the search. It has provided the essential algorithmic basics that have directed our work.

In this thesis, we created a new algorithm, DL8.5, which relied on DL8 and introduced a branch-and-bound behavior in the dynamic programming process. We proposed the use of the best so far error found in the search space as hierarchical upper bound and a restrictive bound as lower bound. The combination of these bounds showed impressive results compared to earlier work. However, the solution caching inherited from DL8 has an important drawback: the memory consumption.

Despite the choice in DL8 to represent its cache as a prefix tree, the number of elements that must in general be stored is huge and this led to a high memory consumption. To mitigate this problem, we proposed a bounded cache to replace the standard cache used in the DL8 algorithm. The new cache we proposed is still a prefix tree because its compression capacity is an asset. However, we provided the cache with the ability to limit its size. Therefore, the number of solutions that can be stored is bounded. This allowed to reduce the memory consumption of the algorithm at the expense of a longer run time. To reduce the impact of the cache bounding on the run time of the algorithm, we proposed different strategies to remove whenever the cache is full, some specific elements, which ensures a lower impact of their recomputation on the run time.

After solving the memory consumption problem of DL8.5, we showed that it can be used to solve many other problems. We used the DL8.5 algorithm to solve till optimality existing optimization problems that could not be solved beforehand. Specifically, we solved some boosting optimization problems that have been proposed earlier to explain the performance of traditional boosting models. Despite the specification of these models, it was not possible to solve them because there was no algorithm able to learn optimal decision trees given a dataset of weighted instances, whereas this is required to accomplish a boosting task. Based on the DL8.5 algorithm, we proposed an algorithm able to learn optimal decision trees taking the weight of instances into account. Thanks to this algorithm, we were able to solve the models and assess their relevance.

Finally, we proposed a new library, PyDL8.5, as a way to use the numerous features provided by DL8.5 or implemented on top of the DL8.5 algorithm. Despite the fact that the core source code of DL8.5 is written in C++, the library is provided in Python to follow the trend of most of the libraries in data science and to ease its use. Moreover, the library is compatible with the well-know scikit-learn library [Ped+11]. It can solve many machine learning problems and we showed some examples of how this can be done. Note also that in addition to the optimization criterion implemented in the library, it allows its user to define her own optimization criterion as long as it is additive. One can also print a visual representation of the learned trees. Using the PyDL8.5 library is as simple as installing it by running the command `pip install pydl8.5`.

8.2 DL8.5 in real-world applications

During this thesis, we directed some master theses in collaboration with companies. In this section, we concisely present the ones that are related to the use of DL8.5 algorithm on real-world data. Note that these works were not published, but they demonstrate the usefulness of our approach in real-world situations.

8.2.1 Distribution similarity

This work was done in collaboration with a major Belgian company that provides services to the population throughout the country. For each service that they provide to the population, they keep track of information about the status of the service. Thus, the company has a dataset that contains for each customer, the different information related to the status of the services provided to the customer. The status of these services is a numerical data type. The objective of this work was to optimally find groups of customers with similar profiles so that the intra-cluster distance is minimized. There are several clustering algorithms to find these groups. However, they are all heuristic unlike DL8.5. In addition, the company also wanted to understand the similarities. A decision tree is an interesting model to accomplish this task. The dataset provided by the company was collected over twenty months and contains information about 120 million services provided to over 3.2 million customers.

To perform this task, we used the predictive clustering technique introduced in section 7.4. Prior to this, we performed a preprocessing task to convert the dataset of individual services into a dataset of the distribution of service statuses per customer. The number of instances was thus equal to the number of clients. Different versions of the dataset were generated using different levels of aggregation (days, weeks or months). The size of the distribution per customer then depends on this level of aggregation. In order to perform predictive clustering, we need, in addition to the distributions, some features on which the decision tree will rely to explain the predicted clusters. Since we did not have enough information about the customers and in order to reduce the size of the dataset, we transformed the dataset into a new one by grouping the customers into 19781 geographic units provided by the Belgian statistical office (Statbel[1]). These groups bring together several addresses belonging to the same socio-economic status and are a good representation of the clients in them. In addition, the Belgian statistical office provides interesting socio-economic characteristics on these groups that

[1]https://statbel.fgov.be/en

might be interesting to explain the similarities between the customers. Our final dataset consisted of 19781 instances. We were able to perform the predictive clustering task by minimizing the Euclidean distance. The results showed some interesting clusters, mainly explained by the job function of customers. These were validated by a visual representation on a map that confirms the difference between some well-known regions in Belgium, known to be regions with strong economic activities such as airports, ports and industrial areas.

8.2.2 Quantile regression trees

In this work done in collaboration with a large international retail company, the objective was to learn quantile regression trees. Specifically, the company uses an optimization framework to define how they replenish stores and warehouses for each product. To execute this framework, they provide as input, for each product, the prediction of demands in the future. These predictions could be provided by a linear regression model that would learn them based on previous sales records. Unfortunately, the distribution of past product sales contains many outliers and the average value predicted by linear regression models is not representative of the true distribution. In this situation, it is more interesting to predict the demands, for many quantile values, so that the optimization framework will base on these quantile-based predictions that take outliers into account. Moreover, the company was interested in knowing the features explaining the predictions.

In order to learn quantile regression trees, the error function used in DL8.5 must be replaced by the quantile loss error. Fortunately, this error function satisfies the additivity condition described in Chapter 7; that is, the quantile loss produced by a node in a binary decision tree is the sum of the quantile losses produced by its two children. By replacing the error function of DL8.5, one would be able to learn an optimal regression tree for a specific quantile value. However, since the dataset provided was large, this operation takes a considerable amount of time and it becomes very time consuming to successively learn 100 trees for a percentile case, for example. Concretely, the dataset had 70000 instances and 46 features. To mitigate the iterative tree learning process, we proposed in this work, a modification of the DL8.5 algorithm to learn several quantile regression trees at once. The idea is to share the information of the search space between the different trees being searched rather than starting the search from the beginning for each tree. The bounds are therefore adapted. There is one per node and per tree. An example of a modification made is that the upper bound pruning only occurs when the bounds of all trees violate the least restrictive upper bound of a node. The results show an impressive gap between the runtime of our new algorithm and the iterative tree learning process. A future work is to publish these results.

8.3 Perspectives

Concerning the perspectives of improvement of DL8.5, many directions are possible, either in terms of the efficiency of the algorithm in terms of time and memory or in the different problems solved by the algorithm. In this thesis, we present some of them.

8.3.1 Parallelization

One important direction we can consider to make the DL8.5 algorithm more efficient is parallelization of the algorithm. As a recursive algorithm that performs an independent search each time a branching occurs, it is intuitive to consider the algorithm as a parallel algorithm. However, an important aspect to consider when implementing the parallel version of DL8.5 is cache access. Since the algorithm is based on a dynamic programming concept, the cache will be regularly used by each process to store or get the solution of subproblems. It is therefore important to find a way to coordinate how the cache will be accessed to avoid collisions. This problem can be solved by implementing a dedicated protocol deciding which process and when a process can use the cache, while ensuring that this does not produce an overhead. For example, a priority metric could be computed based on the importance of the solution to be stored/retrieved by each process in order to decide the order of access to the cache.

8.3.2 Model distillation

Regarding the future features that can be covered by DL8.5, model distillation is an interesting one. The idea is to find a decision tree that can explain the decisions of a complex model. This allows to find explanations for well-performing models but which lack of interpretability. The problem has already been studied by some researchers [RSG16a; RSG16b; RSG18], but it has not been treated as an optimal decision tree learning problem yet. In the context of optimal decision trees, the model distillation problem can be considered as the problem of finding a decision tree for which the difference between the prediction of the tree and that of the model is minimal. Being aware that a decision tree explaining all the decisions of a complex model can be huge, it is possible to consider finding the best tree minimizing this error under a maximum depth constraint. Another important aspect is that the dataset used to learn the complex model will be needed by the decision tree learning algorithm. In the context of trained models for which the input dataset is not existing, an approach to generate a large dataset with different examples can be considered. Otherwise, models that can be used to generate datasets will be preferred.

8.3.3 Continuous features

Another interesting feature is the handling of numerical attributes. Currently, DL8.5 as well as other existing approaches for learning ODTs only support binary datasets as input. When datasets with numeric attributes are provided, they are converted to a binary dataset as shown in Section 4.3.1. Then, the binary dataset is passed to algorithms designed for binary datasets. Unfortunately, the binarization of numerical features leads to a dataset with numerous Boolean features, whereas the run time of the learning algorithms increase with the number of features. In practice, there are many real-world applications in which it is not feasible to learn an optimal decision tree because the datasets contain mostly numerical features with a large number of distinct real values per feature. In this context, a possible approach could be to perform discretization with a fixed number of values (equal width or equal frequency) before performing binarization. However, the guarantee of optimality is lost.

An attempt to learn optimal decision trees when processing numerical attributes could be a better handling of the feature columns generated after the binarization process. Since the features are generated from the same original feature, some preprocessing operations before or during the learning phase could allow, after exploring some features, to identify other features from the same original feature, which could become irrelevant and be removed from the list of features to explore.

Another aspect that might be interesting to study is the distribution of feature values originating from the same feature. We might expect them to share some similarities. Therefore, we might expect some kind of similarity bound to be derived for one feature based on another.

The use of instances and path length as a representation of subproblems will also lead to more reuse of solutions and thus a reduction in execution time. However, the closed itemsets concept introduced in DL8 could be explored to create a more compact form of this representation and therefore avoid the risk of too much memory consumption.

Bibliography

[AAP00] R. C. Agarwal, C. C. Aggarwal, and V. Prasad. “Depth first generation of long patterns”. In: *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2000, pp. 108–118.

[AAP01] R. C. Agarwal, C. C. Aggarwal, and V. Prasad. “A tree projection algorithm for generation of frequent item sets”. In: *Journal of parallel and Distributed Computing* 61.3 (2001), pp. 350–371.

[AAV19] S. Aghaei, M. J. Azizi, and P. Vayanos. “Learning Optimal and Fair Decision Trees for Non-Discriminative Decision-Making”. In: *arXiv preprint arXiv:1903.10598* (2019).

[Agr+96] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. I. Verkamo, et al. “Fast discovery of association rules.” In: *Advances in knowledge discovery and data mining* 12.1 (1996), pp. 307–328.

[AGV21] S. Aghaei, A. Gómez, and P. Vayanos. “Strong optimal classification trees”. In: *arXiv preprint arXiv:2103.15965* (2021).

[AIS93] R. Agrawal, T. Imieliński, and A. Swami. “Mining association rules between sets of items in large databases”. In: *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. 1993, pp. 207–216.

[Ang+17] E. Angelino, N. Larus-Stone, D. Alabi, M. Seltzer, and C. Rudin. “Learning certifiably optimal rule lists for categorical data”. In: *arXiv preprint arXiv:1704.01701* (2017).

[AS+94] R. Agrawal, R. Srikant, et al. “Fast algorithms for mining association rules”. In: *Proc. 20th int. conf. very large data bases, VLDB*. Vol. 1215. Citeseer. 1994, pp. 487–499.

[Ave20] F. Avellaneda. “Efficient inference of optimal decision trees”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 3195–3202.

[AW10] H. Abdi and L. J. Williams. “Principal component analysis”. In: *Wiley interdisciplinary reviews: computational statistics* 2.4 (2010), pp. 433–459.

[BB96] K. P. Bennett and J. A. Blue. “Optimal decision trees”. In: *Rensselaer Polytechnic Institute Math Report* 214 (1996), p. 24.

[BCG01] D. Burdick, M. Calimlim, and J. Gehrke. “Mafia: A maximal frequent itemset algorithm for transactional databases”. In: *Proceedings 17th international conference on data engineering*. IEEE. 2001, pp. 443–452.

[BD17] D. Bertsimas and J. Dunn. “Optimal classification trees”. In: *Machine Learning* 106.7 (2017), pp. 1039–1082.

[BG98] S. Balakrishnama and A. Ganapathiraju. “Linear discriminant analysis-a brief tutorial”. In: *Institute for Signal and information Processing* 18.1998 (1998), pp. 1–8.

[BHO09] C. Bessiere, E. Hebrard, and B. O’Sullivan. “Minimising decision tree size as combinatorial optimisation”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2009, pp. 173–187.

[BR97] C. Bessiere and J.-C. Régin. “Arc consistency for general constraint networks: preliminary results”. In: (1997).

[Bre+84] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks, 1984.

[Bre+98] L. Breiman et al. “Arcing classifier (with discussion and a rejoinder by the author)”. In: *The annals of statistics* 26.3 (1998), pp. 801–849.

[Bre97a] L. Breiman. *Arcing the edge*. Tech. rep. Technical Report 486, Statistics Department, University of California at . . ., 1997.

[Bre97b] L. Breiman. “Prediction games and arcing classifiers”. In: *Technical Report 504, Statistics Department, University of California at Berkeley* (1997).

[BRR98] H. Blockeel, L. D. Raedt, and J. Ramon. “Top-Down Induction of Clustering Trees”. In: *Proceedings of the Fifteenth International Conference on Machine Learning (ICML 1998), 1998*. 1998, pp. 55–63.

[CS14] M. W. Craven and J. W. Shavlik. “Learning symbolic rules using artificial neural networks”. In: *Proceedings of the Tenth International Conference on Machine Learning*. 2014, pp. 73–80.

[DBS02] A. Demiriz, K. P. Bennett, and J. Shawe-Taylor. “Linear programming boosting via column generation”. In: *Machine Learning* 46.1-3 (2002), pp. 225–254.

[DDS06] G. Desaulniers, J. Desrosiers, and M. M. Solomon. *Column generation*. Vol. 5. Springer Science & Business Media, 2006.

[Dem+16] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J.-C. Régin, and P. Schaus. “Compact-table: efficiently filtering table constraints with reversible sparse bit-sets”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2016, pp. 207–223.

[Dem+22] E. Demirović, A. Lukina, E. Hebrard, J. Chan, J. Bailey, C. Leckie, K. Ramamohanarao, and P. J. Stuckey. “MurTree: Optimal Decision Trees via Dynamic Programming and Search”. In: *Journal of Machine Learning Research* 23.26 (2022), pp. 1–47.

[Du +99] O. Du Merle, D. Villeneuve, J. Desrosiers, and P. Hansen. “Stabilized column generation”. In: *Discrete Mathematics* 194.1-3 (1999), pp. 229–237.

[FD98] M. Frean and T. Downs. *A simple cost function for boosting*. Tech. rep. Technical report, Dep. of Computer Science and Electrical Engineering . . ., 1998.

[FHT+00] J. Friedman, T. Hastie, R. Tibshirani, et al. “Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors)”. In: *The annals of statistics* 28.2 (2000), pp. 337–407.

[FS97] Y. Freund and R. E. Schapire. “A decision-theoretic generalization of on-line learning and an application to boosting”. In: *Journal of computer and system sciences* 55.1 (1997), pp. 119–139.

[Gen+07] I. P. Gent, C. Jefferson, I. Miguel, and P. Nightingale. “Data structures for generalised arc consistency for extensional constraints”. In: *AAAI*. Vol. 7. 2007, pp. 191–197.

[GG61] P. C. Gilmore and R. E. Gomory. “A linear programming approach to the cutting-stock problem”. In: *Operations research* 9.6 (1961), pp. 849–859.

[GR71] G. H. Golub and C. Reinsch. “Singular value decomposition and least squares solutions”. In: *Linear algebra*. Springer, 1971, pp. 134–151.

[GS98] A. J. Grove and D. Schuurmans. “Boosting in the limit: Maximizing the margin of learned ensembles”. In: *AAAI/IAAI*. 1998, pp. 692–699.

[Gur21] L. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2021.

[GZ13] W. Gao and Z.-H. Zhou. “On the doubt about margin explanation of boosting”. In: *Artificial Intelligence* 203 (2013), pp. 1–18.

[Hol+95] M. Holsheimer, M. L. Kersten, H. Mannila, and H. Toivonen. “A perspective on databases and data mining”. In: *KDD*. Vol. 95. 1995, pp. 150–155.

[HRS19] X. Hu, C. Rudin, and M. Seltzer. “Optimal sparse decision trees”. In: *Advances in Neural Information Processing Systems* 32 (2019).

[IC94] J. P. Ignizio and T. M. Cavalier. *Linear programming*. Prentice-Hall, Inc., 1994.

[INM19] A. Ignatiev, N. Narodytska, and J. Marques-Silva. “Abduction-based explanations for machine learning models”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 1511–1519.

[Lin+20] J. Lin, C. Zhong, D. Hu, C. Rudin, and M. Seltzer. “Generalized and scalable optimal sparse decision trees”. In: *ICML*. PMLR. 2020, pp. 6150–6160.

[LR76] H. Laurent and R. L. Rivest. “Constructing optimal binary decision trees is NP-complete”. In: *Information processing letters* 5.1 (1976), pp. 15–17.

[Mas+99] L. Mason, J. Baxter, P. L. Bartlett, M. Frean, et al. “Functional gradient techniques for combining hypotheses”. In: *Advances in Neural Information Processing Systems* (1999), pp. 221–246.

[Miš20] V. V. Mišić. “Optimization of tree ensembles”. In: *Operations Research* 68.5 (2020), pp. 1605–1624.

[MSV17] L. Michel, P. Schaus, and P. Van Hentenryck. *Mini-CP: A Minimalist Open-Source Solver to Teach Constraint Programming*. 2017.

[Nar+18] N. Narodytska, A. Ignatiev, F. Pereira, J. Marques-Silva, and I. RAS. “Learning Optimal Decision Trees with SAT.” In: *IJCAI*. 2018, pp. 1362–1368.

[NF07] S. Nijssen and E. Fromont. “Mining optimal decision trees from itemset lattices”. In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2007, pp. 530–539.

[NF10] S. Nijssen and E. Fromont. “Optimal constraint-based decision tree induction from itemset lattices”. In: *Data Mining and Knowledge Discovery* 21.1 (2010), pp. 9–51.

[Nib87] T. Niblett. “Constructing decision trees in noisy domains”. In: *Proceedings of the 2nd European Conference on European Working Session on Learning*. 1987, pp. 67–78.

[ORM98] T. Onoda, G. Rätsch, and K.-R. Müller. “An asymptotic analysis of AdaBoost in the binary classification case”. In: *International Conference on Artificial Neural Networks*. Springer. 1998, pp. 195–200.

[Pas+99] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. “Discovering frequent closed itemsets for association rules”. In: *International Conference on Database Theory*. Springer. 1999, pp. 398–416.

[Ped+11] 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”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[PHM+00] J. Pei, J. Han, R. Mao, et al. “CLOSET: An efficient algorithm for mining frequent closed itemsets.” In: *ACM SIGMOD workshop on research issues in data mining and knowledge discovery*. Vol. 4. 2. 2000, pp. 21–30.

[PP20] M. Pfetsch and S. Pokutta. “IPBoost–non-convex boosting via integer programming”. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 7663–7672.

[QR89] J. R. Quinlan and R. L. Rivest. “Inferring decision trees using the minimum description lenght principle”. In: *Information and computation* 80.3 (1989), pp. 227–248.

[Qui86] J. R. Quinlan. “Induction of decision trees”. In: *Machine learning* 1.1 (1986), pp. 81–106.

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 1558602402.

[Ris78] J. Rissanen. “Modeling by shortest data description”. In: *Automatica* 14.5 (1978), pp. 465–471.

[Ris87] J. Rissanen. “Minimum-description-length principle”. In: *Encyclopedia of Statistic Sciences* (1987).

[ROM01] G. Rätsch, T. Onoda, and K.-R. Müller. “Soft margins for AdaBoost”. In: *Machine learning* 42.3 (2001), pp. 287–320.

[RS06] L. Reyzin and R. E. Schapire. “How boosting the margin can also boost classifier complexity”. In: *Proceedings of the 23rd international conference on Machine learning*. 2006, pp. 753–760.

[RSG16a] M. T. Ribeiro, S. Singh, and C. Guestrin. “" Why should i trust you?" Explaining the predictions of any classifier”. In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2016, pp. 1135–1144.

[RSG16b] M. T. Ribeiro, S. Singh, and C. Guestrin. “Model-agnostic interpretability of machine learning”. In: *arXiv preprint arXiv:1606.05386* (2016).

[RSG18] M. T. Ribeiro, S. Singh, and C. Guestrin. “Anchors: High-precision model-agnostic explanations”. In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[RW05] G. Rätsch and M. K. Warmuth. “Efficient margin maximizing with boosting”. In: *Journal of Machine Learning Research* 6.Dec (2005), pp. 2131–2152.

[RW88] J. I. Rissanen and M. Wax. *Algorithm for constructing tree structured classifiers*. US Patent 4,719,571. Jan. 1988.

[SAG17] P. Schaus, J. O. Aoga, and T. Guns. “Coversize: A global constraint for frequency-based itemset mining”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2017, pp. 529–546.

[Sch+98] R. E. Schapire, Y. Freund, P. Bartlett, W. S. Lee, et al. "Boosting the margin: A new explanation for the effectiveness of voting methods". In: *The annals of statistics* 26.5 (1998), pp. 1651–1686.

[SF13] R. E. Schapire and Y. Freund. "Boosting: Foundations and algorithms". In: *Kybernetes* (2013).

[Sha48] C. E. Shannon. "A mathematical theory of communication". In: *The Bell system technical journal* 27.3 (1948), pp. 379–423.

[SL10] C. Shen and H. Li. "Boosting through optimization of margin distributions". In: *IEEE Transactions on Neural Networks* 21.4 (2010), pp. 659–666.

[SS99] R. E. Schapire and Y. Singer. "Improved boosting algorithms using confidence-rated predictions". In: *Machine learning* 37.3 (1999), pp. 297–336.

[Ver+19] H. Verhaeghe, S. Nijssen, G. Pesant, C.-G. Quimper, and P. Schaus. "Learning optimal decision trees using constraint programming". In: *The 25th International Conference on Principles and Practice of Constraint Programming (CP2019)*. 2019.

[Ver+20] H. Verhaeghe, S. Nijssen, G. Pesant, C.-G. Quimper, and P. Schaus. "Learning optimal decision trees using constraint programming". In: *Constraints* 25.3 (2020), pp. 226–250.

[VW94] P. Volf and F. M. Willems. "Context maximizing: Finding MDL decision trees". In: *15th Symp. Inform. Theory Benelux*. 1994, pp. 192–200.

[VZ19] S. Verwer and Y. Zhang. "Learning optimal classification trees using a binary linear program formulation". In: *Proceedings of AAAI*. Vol. 33. 01. 2019, pp. 1625–1632.

[Wan+05] J. Wang, J. Han, Y. Lu, and P. Tzvetkov. "TFP: An efficient algorithm for mining top-k frequent closed itemsets". In: *IEEE Transactions on Knowledge and Data Engineering* 17.5 (2005), pp. 652–663.

[Wol20] L. A. Wolsey. *Integer programming*. John Wiley & Sons, 2020.

[WP93] C. S. Wallace and J. D. Patrick. "Coding decision trees". In: *Machine Learning* 11.1 (1993), pp. 7–22.

[Zak+97] M. J. Zaki, S. Parthasarathy, M. Ogihara, W. Li, et al. “New algorithms for fast discovery of association rules.” In: *KDD*. Vol. 97. 1997, pp. 283–286.

[ZG03] M. J. Zaki and K. Gouda. “Fast vertical mining using diffsets”. In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2003, pp. 326–335.

[ZH02] M. J. Zaki and C.-J. Hsiao. “CHARM: An efficient algorithm for closed itemset mining”. In: *Proceedings of the 2002 SIAM international conference on data mining*. SIAM. 2002, pp. 457–473.